Juan de Lara
Andrea Zisman (Eds.)

ARCoSS
LNCS 7212

Fundamental Approaches to Software Engineering

15th International Conference, FASE 2012
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2012
Tallinn, Estonia, March/April 2012, Proceedings

Springer

Lecture Notes in Computer Science 7212

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

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

Juan de Lara Andrea Zisman (Eds.)

Fundamental Approaches to Software Engineering

15th International Conference, FASE 2012
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012
Tallinn, Estonia, March 24 – April 1, 2012
Proceedings

Volume Editors

Juan de Lara
Universidad Autónoma de Madrid
School of Computer Science
Campus Cantoblanco
28049 Madrid, Spain
E-mail: juan.delara@uam.es

Andrea Zisman
City University
School of Informatics
Northampton Square
London EC1V 0HB, UK
E-mail: a.zisman@soi.city.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-28871-5 e-ISBN 978-3-642-28872-2
DOI 10.1007/978-3-642-28872-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012932857

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

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 2012 is the fifteenth 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 six sister conferences (CC, ESOP, FASE, FOSSACS, POST, TACAS), 21 satellite workshops (ACCAT, AIPA, BX, BYTECODE, CMCS, DICE, FESCA, FICS, FIT, GRAPHITE, GT-VMT, HAS, IWIGP, LDTA, LINEARITY, MBT, MSFP, PLACES, QAPL, VSSE and WRLA), and eight invited lectures (excluding those specific to the satellite events).

The six main conferences received this year 606 submissions (including 21 tool demonstration papers), 159 of which were accepted (6 tool demos), giving an overall acceptance rate just above 26%. 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 to participate in this exciting event, and that you will all continue to submit to ETAPS and contribute to making 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, security and improvement. The languages, methodologies and tools that 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.

This year, ETAPS welcomes a new main conference, *Principles of Security and Trust*, as a candidate to become a permanent member conference of ETAPS. POST is the first addition to our main programme since 1998, when the original five conferences met in Lisbon for the first ETAPS event. It combines the practically important subject matter of security and trust with strong technical connections to traditional ETAPS areas.

A step towards the consolidation of ETAPS and its institutional activities has been undertaken by the Steering Committee with the establishment of *ETAPS e.V.*, a non-profit association under German law. ETAPS e.V. was founded on April 1st, 2011 in Saarbrücken, and we are currently in the process of defining its structure, scope and strategy.

ETAPS 2012 was organised by the *Institute of Cybernetics at Tallinn University of Technology*, 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)

and with support from the following sponsors, which we gratefully thank:

Institute of Cybernetics at TUT; Tallinn University of Technology (TUT); Estonian Centre of Excellence in Computer Science (EXCS) funded by the European Regional Development Fund (ERDF); Estonian Convention Bureau; and Microsoft Research.

The organising team comprised:

General Chair:	*Tarmo Uustalu*
Satellite Events:	*Keiko Nakata*
Organising Committee:	*James Chapman, Juhan Ernits, Tiina Laasma, Monika Perkmann* and their colleagues in the *Logic and Semantics* group and *administration* of the *Institute of Cybernetics*

The ETAPS portal at http://www.etaps.org is maintained by *RWTH Aachen University*.

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

Vladimiro Sassone (Southampton, Chair), Roberto Amadio (Paris 7), Gilles Barthe (IMDEA-Software), David Basin (Zürich), Lars Birkedal (Copenhagen), Michael O'Boyle (Edinburgh), Giuseppe Castagna (CNRS Paris), Vittorio Cortellessa (L'Aquila), Koen De Bosschere (Gent), Pierpaolo Degano (Pisa), Matthias Felleisen (Boston), Bernd Finkbeiner (Saarbrücken), Cormac Flanagan (Santa Cruz), Philippa Gardner (Imperial College London), Andrew D. Gordon (MSR Cambridge and Edinburgh), Daniele Gorla (Rome), Joshua Guttman (Worcester USA), Holger Hermanns (Saarbrücken), Mike Hinchey (Lero, the Irish Software Engineering Research Centre), Ranjit Jhala (San Diego), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Barbara König (Duisburg), Juan de Lara (Madrid), Gerald Lüttgen (Bamberg), Tiziana Margaria (Potsdam), Fabio Martinelli (Pisa), John Mitchell (Stanford), Catuscia Palamidessi (INRIA Paris), Frank Pfenning (Pittsburgh), Nir Piterman (Leicester), Don Sannella (Edinburgh), Helmut Seidl (TU Munich),

Scott Smolka (Stony Brook), Gabriele Taentzer (Marburg), Tarmo Uustalu (Tallinn), Dániel Varró (Budapest), Andrea Zisman (London), and Lenore Zuck (Chicago).

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

Finally, I would like to thank the Organising Chair of ETAPS 2012, Tarmo Uustalu, and his Organising Committee, for arranging to have ETAPS in the most beautiful surroundings of Tallinn.

January 2012

Vladimiro Sassone
ETAPS SC Chair

Preface

This volume contains the papers accepted for FASE 2012, the 15th International Conference on Fundamental Approaches to Software Engineering, which was held in Tallinn, Estonia, in March 2012 as part of the annual European Joint Conference on Theory and Practice of Software (ETAPS). FASE is concerned with the foundations on which software engineering is built. It focusses on novel techniques and the way in which they contribute to make software engineering a more mature and soundly based discipline.

This year we solicited two kinds of contributions: research papers and tool demonstration papers. We received 134 submissions from 39 countries around the world, of which 5 were tool demonstrations. After a rigorous selection process, the Programme Committee accepted 33 submissions (2 of which were tool demonstrations), corresponding to an acceptance rate of approximately 24.6%. Each paper received at least three reviews, and four in some cases. The acceptance decisions were made after exhaustive and careful online discussions by the members of the Programme Committee.

The accepted papers cover several aspects of software engineering, including verification, slicing and refactoring, testing, model transformations, components, software architecture, product lines, and empirical aspects of the development process. We believe that the accepted papers made a scientifically strong and exciting programme, which triggered interesting discussions and exchange of ideas among the FASE participants.

This year, we were honoured to host an invited talk by Wil van der Aalst from Eindhoven University of Technology (The Netherlands) and Queensland University of Technology (Australia) entitled "Distributed Process Discovery and Conformance Checking". Professor van der Aalst is internationally recognised by his pioneering work on workflow management, process mining, and Petri nets. The presentation discussed the challenges for distributed process mining in the context of both procedural and declarative process models.

We would like to thank all authors who submitted their work to FASE 2012. Without their excellent contributions we would not have managed to prepare a strong programme. We would also like to thank the Programme Committee members and external reviewers for their high-quality reviews and the effort and time they dedicated to the review and discussion processes. Finally, we wish to express our sincere gratitude to the Organizing and Steering Committees for their continuous support. The logistics of our job as Programme Chairs were facilitated by the EasyChair system, and supported by Andrei Voronkov.

We sincerely hope that you will enjoy reading these proceedings.

January 2012

Juan de Lara
Andrea Zisman

Organization

Programme Committee

Luciano Baresi	Politecnico di Milano, Italy
Don Batory	University of Texas at Austin, USA
Artur Boronat	University of Leicester, UK
Paolo Bottoni	University of Rome, Italy
Marsha Chechik	University of Toronto, Canada
Shing-Chi Cheung	Hong Kong University of Science and Technology, Hong Kong, SAR China
Luca De Alfaro	University of California, Santa Cruz, USA
Jurgen Dingel	Queen's University, Canada
Gregor Engels	University of Paderborn, Germany
Claudia Ermel	Technische Universität Berlin, Germany
Dimitra Giannakopoulou	Carnegie Mellon University/NASA Ames, USA
Holger Giese	Hasso Plattner Institute, Germany
Esther Guerra	Universidad Autónoma de Madrid, Spain
Reiko Heckel	University of Leicester, UK
John Hosking	University of Auckland, New Zealand
Christos Kloukinas	City University London, UK
Alexander Knapp	University of Augsburg, Germany
Jeff Kramer	Imperial College London, UK
Luis Lamb	Federal University of Rio Grande do Sul, Brazil
Yngve Lamo	Bergen University College, Norway
Tiziana Margaria	University of Potsdam, Germany
Fernando Orejas	Universidad Politécnica Catalunya, Spain
Richard Paige	The University of York, UK
Alfonso Pierantonio	Università degli Studi dell'Aquila, Italy
Andy Schürr	Technische Universität Darmstadt, Germany
George Spanoudakis	City University London, UK
Jesús Sánchez Cuadrado	Universidad Autónoma de Madrid, Spain
Gabriele Taentzer	Philipps-Universität Marburg, Germany
Daniel Varro	Budapest University of Technology and Economics, Hungary

Additional Reviewers

Albarghouthi, Aws	Anjorin, Anthony	Apel, Sven
Ardagna, Danilo	Arendt, Thorsten	Arifulina, Svetlana
Bals, Jan-Christopher	Bapodra, Mayur	Becker, Basil
Berger, Thorsten	Bergmann, Gábor	Bianculli, Domenico

Biermann, Enrico
Bisztray, Denes
Bordihn, Henning
Borges, Rafael
Braatz, Benjamin
Brooke, Phil
Bruch, Marcel
Bucchiarone, Antonio
Christ, Fabian
Cicchetti, Antonio
Cichos, Harald
Cota, Erika
Di Ruscio, Davide
Diaz, Oscar
Doedt, Markus
Duarte, Lucio Mauro
Ehrig, Hartmut
Fazal-Baqaie, Masud
Foster, Howard
Franch, Xavier
Gabriel, Karsten
Gabrysiak, Gregor
Galloway, Andy
Garcez, Artur
Geisen, Silke
Gerth, Christian
Golas, Ulrike
Guinea, Sam
Gurfinkel, Arie
Gönczy, László
Güldali, Baris
Haneberg, Dominik
Hebig, Regina
Hegedüs, Ábel
Hermann, Frank
Hildebrandt, Stephan
Horváth, Ákos
Huang, Jeff
Khan, Tamim A.
Kincaid, Zachary
Kocsis, Imre
Kovi, Andras
Krause, Christian
Lambers, Leen
Lauder, Marius
Liebig, Jörg
Liu, Yepang
Luckey, Markus
Machado, Rodrigo
Mahbub, Khaled
Mantz, Florian
Matragkas, Nikos
Mezei, Gergely
Monga, Mattia
Morasca, Sandro
Moreira, Alvaro
Mühlberger, Heribert
Naeem, Muhammad
Nagel, Benjamin
Nejati, Shiva
Neumann, Stefan
Ortega, Alfonso
Oster, Sebastian
Patzina, Lars
Patzina, Sven
Pelliccione, Patrizio
Polack, Fiona
Posse, Ernesto
Qayum, Fawad
Radjenovic, Alek
Rakamaric, Zvonimir
Raman, Vishwanath
Rose, Louis
Rossi, Matteo
Rubin, Julia
Rungta, Neha
Rutle, Adrian
Ráth, István
Rüthing, Oliver
Sabetzadeh, Mehrdad
Salay, Rick
Saller, Karsten
Schaefer, Ina
Soltenborn, Christian
Spijkerman, Michael
Steffen, Bernhard
Tavakoli Kolagari, Ramin
Thomas, Stephen
Tkachuk, Oksana
Varro, Gergely
Vazquez-Salceda, Javier
Vogel, Thomas
Waez, Md Tawhid Bin
Wagner, Christian
Wahl, Thomas
Wang, Xiaoliang
Wang, Xinming
Wieber, Martin
Williams, James
Wimmer, Manuel
Wonisch, Daniel
Wätzoldt, Sebastian
Ye, Chunyang
Zhang, Zhenyu
Zurowska, Karolina

Table of Contents

Intermodelling and Model Transformations

Modelling and Adaptation

Product Lines and Feature-Oriented Programming

Development Process

Verification and Synthesis

Testing and Maintenance

Slicing and Refactoring

Distributed Process Discovery and Conformance Checking

Wil M.P. van der Aalst[1,2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] Queensland University of Technology, Brisbane, Australia
www.vdaalst.com

Abstract. Process mining techniques have matured over the last decade and more and more organization started to use this new technology. The two most important types of process mining are *process discovery* (i.e., learning a process model from example behavior recorded in an event log) and *conformance checking* (i.e., comparing modeled behavior with observed behavior). Process mining is motivated by the availability of event data. However, as event logs become larger (say terabytes), performance becomes a concern. The only way to handle larger applications while ensuring acceptable response times, is to *distribute* analysis over a network of computers (e.g., multicore systems, grids, and clouds). This paper provides an overview of the different ways in which process mining problems can be distributed. We identify three types of distribution: *replication*, a *horizontal partitioning* of the event log, and a *vertical partitioning* of the event log. These types are discussed in the context of both *procedural* (e.g., Petri nets) and *declarative* process models. Most challenging is the horizontal partitioning of event logs in the context of procedural models. Therefore, a new approach to decompose Petri nets and associated event logs is presented. This approach illustrates that process mining problems can be distributed in various ways.

Keywords: process mining, distributed computing, grid computing, process discovery, conformance checking, business process management.

1 Introduction

Digital data is everywhere – in every sector, in every economy, in every organization, and in every home – and will continue to grow exponentially [22]. Some claim that all of the world's music can be stored on a $600 disk drive. However, despite Moore's Law, storage space and computing power cannot keep up with the growth of event data. Therefore, analysis techniques dealing with "big data" [22] need to resort to distributed computing.

This paper focuses on *process mining*, i.e., the analysis of processes based on *event data* [3]. Process mining techniques aim to *discover, monitor, and improve processes using event logs*. Process mining is a relatively young research discipline that sits between machine learning and data mining on the one hand, and process analysis and formal methods on the other hand. The idea of process mining is

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 1–25, 2012.

to discover, monitor and improve real processes (i.e., not assumed processes) *by extracting knowledge from event logs* readily available in today's (information) systems. Process mining includes (automated) process discovery (i.e., extracting process models from an event log), conformance checking (i.e., monitoring deviations by comparing model and log), social network/organizational mining, automated construction of simulation models, model extension, model repair, case prediction, and history-based recommendations.

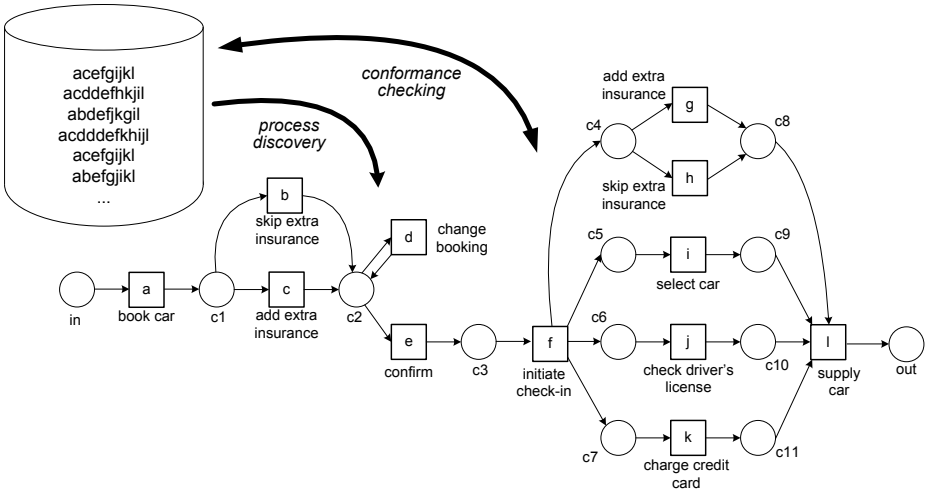


Fig. 1. Example illustrating two types of process mining: process discovery and conformance checking

Figure 1 illustrates the two most important types of process mining: process discovery and conformance checking. Starting point for process mining is an event log. Each event in such a log refers to an *activity* (i.e., a well-defined step in some process) and is related to a particular *case* (i.e., a *process instance*). The events belonging to a case are *ordered* and can be seen as one "run" of the process. For example, the first case in the event log shown in Fig. 1 can be described by the trace $\langle a, c, e, f, g, i, j, k, l \rangle$. This is the scenario where a car is booked (activity a), extra insurance is added (activity c), the booking is confirmed (activity e), the check-in process is initiated (activity f), more insurance is added (activity g), a car is selected (activity i), the license is checked (activity j), the credit card is charged (activity k), and the car is supplied (activity l). The second case is described by the trace $\langle a, c, d, d, e, f, h, k, j, i, l \rangle$. In this scenario, the booking was changed two times (activity d) and no extra insurance was taken at check-in (activity h). It is important to note that an event log contains only example behavior, i.e., we cannot assume that all possible runs have been observed. In fact, an event log often contains only a fraction of the possible behavior [3].

Process discovery techniques automatically create a model based on the example behavior seen in the event log. For example, based on the event log shown

in Fig. 1 the corresponding Petri net is created. Note that the Petri net shown in Fig. 1 is indeed able to generate the behavior in the event log. The model allows for more behavior, but this is often desirable as the model should generalize the observed behavior.

Whereas process discovery constructs a model without any a priori information (other than the event log), conformance checking uses a model and an event log as input. The model may have been made by hand or discovered through process discovery. For conformance checking, the modeled behavior and the observed behavior (i.e., event log) are compared. There are various approaches to diagnose and quantify conformance. For example, one can measure the fraction of cases in the log that can be generated by the model. In Fig. 1, all cases fit the model perfectly. However, if there would have been a case following trace $\langle a, c, f, h, k, j, i, l \rangle$, then conformance checking techniques would identify that in this trace activity e (the confirmation) is missing.

Given a small event log, like the one shown in Fig. 1, analysis is simple. However, in reality, process models may have hundreds of different activities and there may be millions of events and thousands of unique cases. In such cases, process mining techniques may have problems to produce meaningful results in a reasonable time. This is why we are interested in *distributed process mining*, i.e., decomposing challenging process discovery and conformance checking problems into smaller problems that can be distributed over a network of computers.

Today, there are many different types of distributed systems, i.e., systems composed of multiple autonomous computational entities communicating through a network. Multicore computing, cluster computing, grid computing, cloud computing, etc. all refer to systems where different resources are used concurrently to improve performance and scalability. Most data mining techniques can be distributed [16], e.g., there are various techniques for distributed classification, distributed clustering, and distributed association rule mining [13]. However, in the context of process mining only distributed genetic algorithms have been examined in detail [15]. Yet, there is an obvious need for distributed process mining. This paper explores the different ways in which process discovery and conformance checking problems can be distributed. We will not focus on the technical aspects (e.g., the type of distributed system to use) nor on specific mining algorithms. Instead, we systematically explore the different ways in which event logs and models can be partitioned.

The remainder of this paper is organized as follows. First, in Section 2, we discuss the different ways in which process mining techniques can be distributed. Besides *replication*, we define two types of distribution: *vertical distribution* and *horizontal distribution*. In Section 3 we elaborate on the representation of event logs and process models. Here, we also discuss the differences between procedural models and declarative models. We use Petri nets as typical representatives of conventional procedural models. To illustrate the use of declarative models in the context of distributed process mining, we elaborate on the *Declare* language [8]. Section 4 discusses different ways of measuring conformance while zooming in on the notion of fitness. The horizontal distribution of process mining tasks

is promising, but also particularly challenging for procedural models. Therefore, we elaborate on a particular technique to decompose event logs and processes (Section 5). Here we use the notion of *passages* for Petri nets which enables us to split event logs and process models horizontally. Section 6 concludes the paper.

2 Distributed Process Mining: An Overview

This section introduces some basic process mining concepts (Section 2.1) and based on these concepts it is shown that event logs and process models can be distributed in various ways (Section 2.2).

2.1 Process Discovery and Conformance Checking

As explained in the introduction there are two basic types of process mining: *process discovery* and *conformance checking*.[1] Figure 2 shows both types.

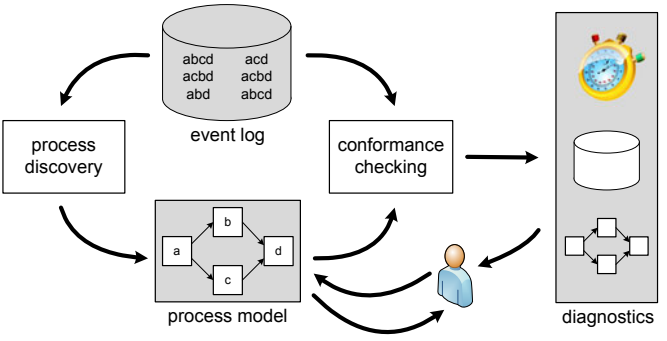


Fig. 2. Positioning process mining techniques

Process discovery techniques take an event log and produce a process model in some notation. Figure 1 already illustrated the basic idea of discovery: learn a process model from example traces.

Conformance checking techniques take an event log and a process model and compare the observed behavior with the modeled behavior. As Fig. 2 shows the process model may be the result of process discovery or made by hand. Basically, three types of conformance-related diagnostics can be generated. First of all, there may be overall metrics describing the degree of conformance, e.g., 80% of all cases can be replayed by the model from begin to end. Second, the non-conforming behavior may be highlighted in the event log. Third, the non-conforming behavior may be revealed by annotating the process model. Note that

[1] As described in [3], process mining is not limited to process discovery and conformance checking and also includes enhancement (e.g., extending or repairing models based on event data) and operational support (on-the-fly conformance checking, prediction, and recommendation). These are out-of-scope for this paper.

conformance can be viewed from two angles: (a) the model does not capture the real behavior ("the model is wrong") and (b) reality deviates from the desired model ("the event log is wrong"). The first viewpoint is taken when the model is supposed to be descriptive, i.e., capture or predict reality. The second viewpoint is taken when the model is normative, i.e., used to influence or control reality.

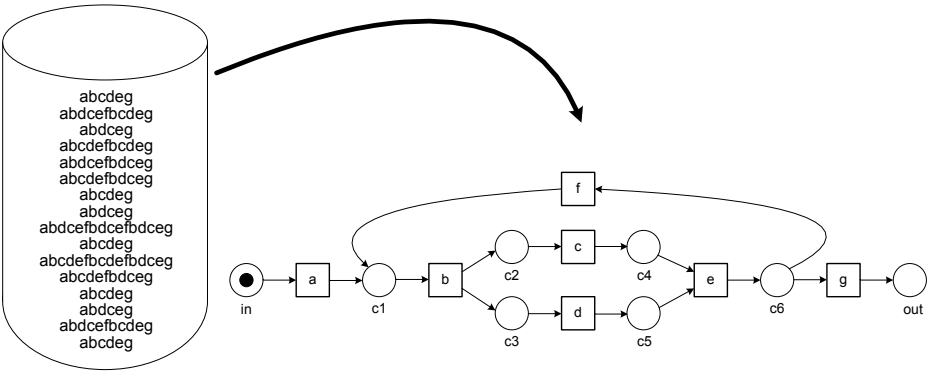


Fig. 3. Example illustrating process discovery

To further illustrate the notion of process discovery consider the example shown in Fig. 3. Based on the event log shown, a Petri net is learned. Note that all traces in the event log start with activity a and end with activity g. This is also the case in the Petri net (consider all full firing sequences starting with a token in place *in* and ending with a token in *out*). After a, activity b can be executed. Transition b in the Petri net is a so-called AND-split, i.e., after executing b, both c and d can be executed concurrently. Transition e is a so-called AND-join. After executing e a choice is made: either g occurs and the case completes or f is executed and the state with a token in place $c1$ is revisited. Many process discovery algorithms have been proposed in literature [9, 10, 12, 17–19, 23, 28–30]. Most of these algorithms have no problems dealing with this small example.

Figure 4 illustrates conformance checking using the same example. Now the event log contains some traces that are not possible according to the process model shown in Fig. 4. As discussed in the context of Fig. 2, there are three types of diagnostics possible. First of all, we can use metrics to describe the degree of conformance. For example, 10 of the 16 cases (i.e., 62.5 percent) in Fig. 4 are perfectly fitting. Second, we can split the log into two smaller event logs: one consisting of conforming cases and one consisting of non-conforming cases. These logs can be used for further analysis, e.g., discover commonalities among non-conforming cases using process discovery. Third, we can highlight problems in the model. As Fig. 4 shows, there is a problem with activity b: according to the model b should be executed before c and d but in the event log this is not always the case. There is also a problem with activity f: it should only be executed after e, but in the log it also appears at other places.

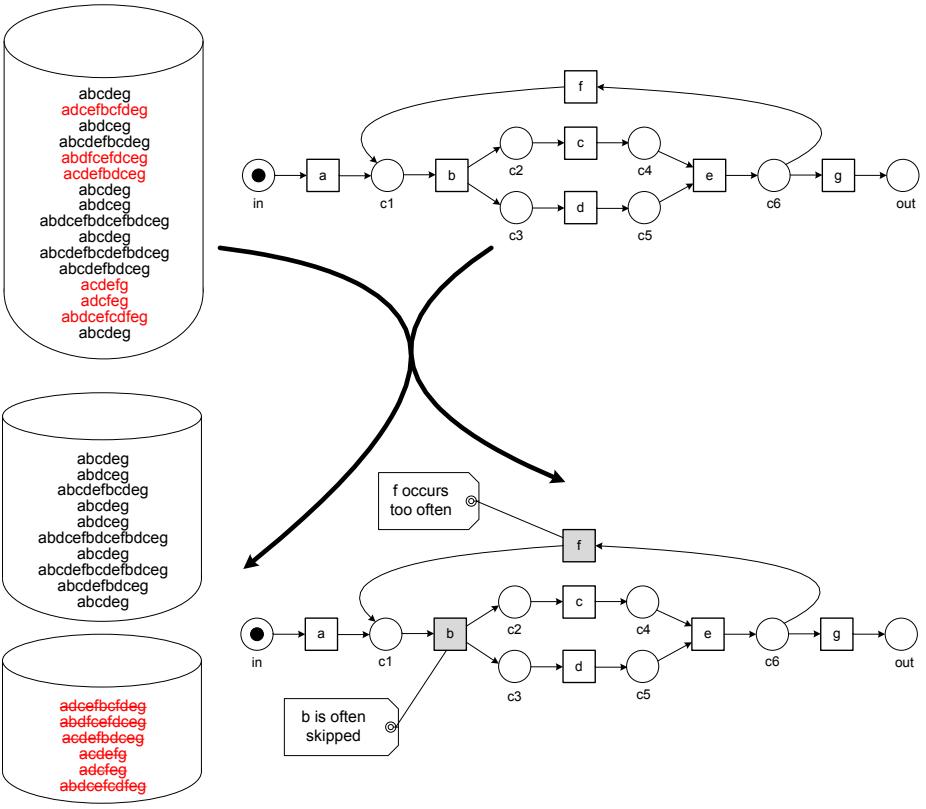


Fig. 4. Example illustrating conformance checking

Figures 3 and 4 show the basic idea of process mining. Note that the example is oversimplified. For example, most event logs contain much more information. In the example log an event is fully described by an activity name. However, often there is additional information about an event such as the *resource* (i.e., person or device) executing or initiating the activity, the *timestamp* of the event, or *data elements* recorded with the event (e.g., the size of an order).

The process models shown thus far are all Petri nets (WF-nets [1, 6] to be precise). Different process mining algorithms may use different representations. Moreover, the notation used to visualize the result may be very different from the representation used during the actual discovery process. All mainstream BPM notations (Petri nets, EPCs, BPMN, YAWL, UML activity diagrams, etc.) can be used to show discovered processes [3, 31]. In fact, later we will also elaborate on so-called *declarative* process models. However, to explain the concept of distributed process mining, such differences are less relevant. Therefore, we defer a discussion on the difference between procedural models and declarative models to Section 3.4.

2.2 Distributing Event Logs and Process Models

New computing paradigms such as cloud computing, grid computing, cluster computing, etc. have emerged to perform resource-intensive IT tasks. Modern computers (even lower-end laptops and high-end phones) have multiple processor cores. Therefore, the distribution of computing-intensive tasks, like process mining on "big data", is becoming more important.

At the same time, there is an exponentially growing torrent of event data. MGI estimates that enterprises globally stored more than 7 exabytes of new data on disk drives in 2010, while consumers stored more than 6 exabytes of new data on devices such as PCs and notebooks [22]. A recent study in *Science* suggests that the total global storage capacity increased from 2.6 exabytes in 1986 to 295 exabytes in 2007 [20]. These studies illustrate the growing potential of process mining.

Given these observations, it is interesting to develop techniques for *distributed process mining*. In recent years, distributed data mining techniques have been developed and corresponding infrastructures have been realized [16]. These techniques typically partition the input data over multiple computing nodes. Each of the nodes computes a local model and these local models are aggregated into an overall model.

In [15], we showed that it is fairly easy to distribute genetic process mining algorithms. In this paper (i.e., [15]), we replicate the entire log such that each node has a copy of all input data. Each node runs the same genetic algorithm, uses the whole event log, but, due to randomization, works with different individuals (i.e., process models). Periodically, the best individuals are exchanged between nodes. It is also possible to partition the input data (i.e., the event log) over all nodes. Experimental results show that distributed genetic process mining significantly speeds-up the discovery process. This makes sense because the fitness test is most time-consuming. However, individual fitness tests are completely independent. Although genetic process mining algorithms can be distributed easily, they are not usable for large and complex data sets. Other process mining algorithms tend to outperform genetic algorithms [3]. Therefore, we also need to consider the distribution of other process mining techniques.

To discuss the different ways of distributing process mining techniques we approach the problem from the viewpoint of the event log. We consider three basic types of distribution:

- *Replication.* If the process mining algorithm is non-deterministic, then the same task can be executed on all nodes and in the end the best result can be taken. In this case, the event log can be simply replicated, i.e., all nodes have a copy of the whole event log.
- *Vertical partitioning.* Event logs are composed of cases. There may be thousands or even millions of cases. These can be distributed over the nodes in the network, i.e., each case is assigned to one computing node. All nodes work on a subset of the whole log and in the end the results need to be merged.

– *Horizontal partitioning.* Cases are composed of multiple events. Therefore, we can also partition cases, i.e., part of a case is analyzed on one node whereas another part of the same case is analyzed on another node. In principle, each node needs to consider all cases. However, the attention of one computing node is limited to a particular subset of events per case.

Of course it is possible to combine the three types of distribution.

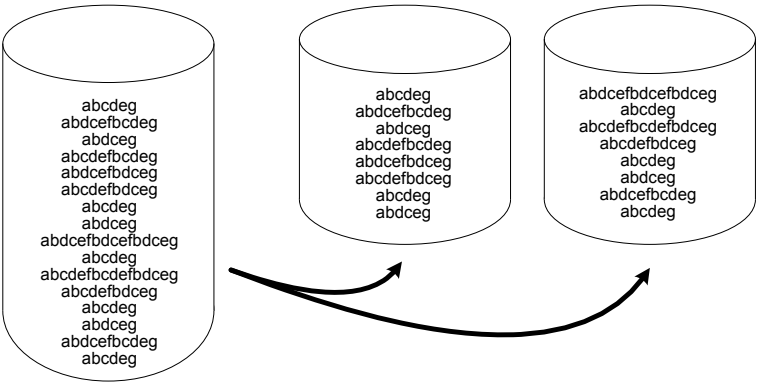


Fig. 5. Partitioning the event log *vertically*: cases are distributed arbitrarily

Figure 5 illustrates the vertical partitioning of an event log using our running example. The original event log contained 16 cases. Assuming that there are two computing nodes, we can partition the cases over these two nodes. Each case resides in exactly one location, i.e., the nodes operate on disjoint sublogs. Each node computes a process mining result for a sublog and in the end the results are merged. Depending on the type of process mining result, merging may be simple or complex. For example, it we are interested in the percentage of fitting cases it is easy to compute the overall percentage. Suppose there are n nodes and each node $i \in \{1 \ldots n\}$ reports on the number of fitting cases (x_i) and non-fitting cases (y_i) in the sublog. The fraction of fitting cases can be computed easily: $(\sum_i \ x_i)/(\sum_i \ x_i + y_i)$. When each node produces a process model, it is more difficult to produce an overall result. However, by using lower-level output such as the dependency matrices used by mining algorithms like the heuristic miner and fuzzy miner [3], one can merge the results.

In Fig. 5 the cases are partitioned over the logs without considering particular features, i.e., the first eight cases are assigned to the first node and the remaining eight cases are assigned to the second node. As Fig. 6 shows, one can also distribute cases based on a particular feature. In this case all cases of length 6 are moved to the first node, cases of length 11 are moved to the second node, and cases of length 16 are moved to the third node. Various features can be used, e.g., the type of customer (one node analyzes the process for gold customers, one for silver customers, etc.), the flow time of the case, the start time of the case,

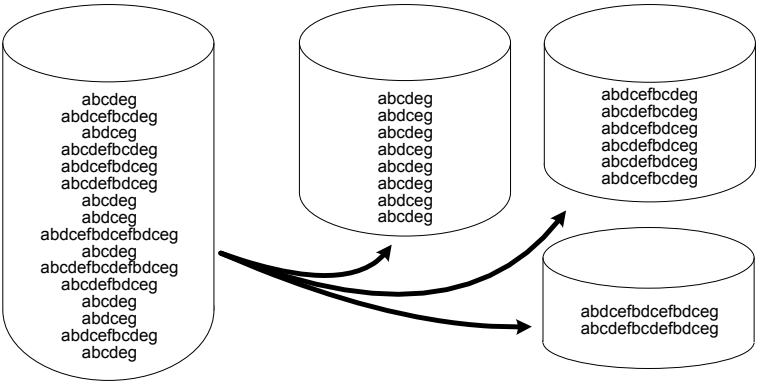


Fig. 6. Partitioning the event log vertically: cases are distributed based on a particular feature (in this case the length of the case)

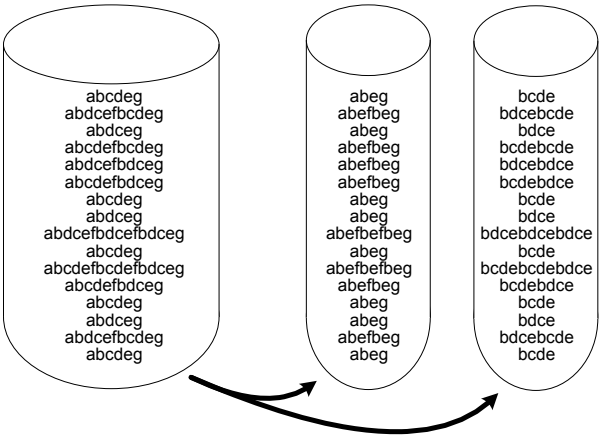


Fig. 7. Partitioning the event log *horizontally*

the monetary value of the case, etc. Such a vertical partitioning may provide additional insights. An example is the use of the start time of cases when distributing the event log. Now it is interesting to see whether there are significant differences between the results. The term *concept drift* refers to the situation in which the process is changing while being analyzed [14]. For instance, in the beginning of the event log two activities may be concurrent whereas later in the log these activities become sequential. Processes may change due to periodic/seasonal changes (e.g., "in December there is more demand" or "on Friday afternoon there are fewer employees available") or due to changing conditions (e.g., "the market is getting more competitive"). A vertical partitioning based on the start time of cases may reveal concept drift or the identification of periods with severe conformance problems.

Figure 7 illustrates the *horizontal* partitioning of event logs. The first sublog contains all events that correspond to activities a, b, e, f, and g. The second sublog contains all events that correspond to activities b, c, d, and e. Note that each case appears in each of the sublogs. However, each sublog contains only a selection of events per case. In other words, events are partitioned "horizontally" instead of "vertically". Each node computes results for a particular sublog. In the end, all results are merged. Figure 8 shows an example of two process fragments discovered by two different nodes. The process fragments are glued together using the common events. In Section 5 we will further elaborate on this.

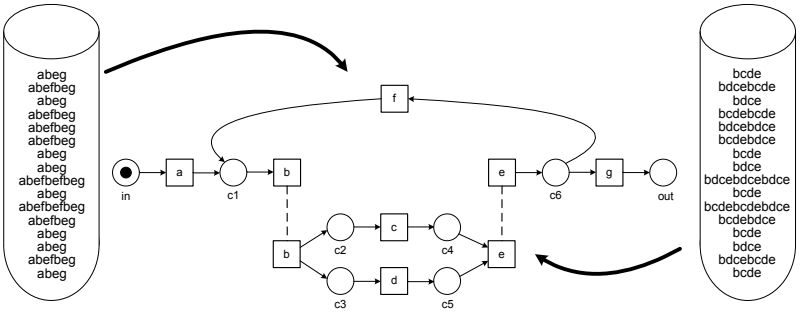


Fig. 8. Horizontally partitioned event logs are used to discover process fragments that can be merged into a complete model.

3 Representation of Event Logs and Process Models

Thus far, we have only discussed things informally. In this section, we formalize some of the notions introduced before. For example, we formalize the notion of an event log and provide some Petri net basics. Moreover, we show an example of a declarative language (*Declare* [8]) grounded in LTL.

3.1 Multisets

Multisets are used to represent the state of a Petri net and to describe event logs where the same trace may appear multiple times.

$\mathcal{B}(A)$ is the set of all multisets over some set A. For some multiset $b \in \mathcal{B}(A)$, $b(a)$ denotes the number of times element $a \in A$ appears in b. Some examples: $b_1 = [\,]$, $b_2 = [x, y, y]$, $b_3 = [x, y, z]$, $b_4 = [x, x, y, x, y, z]$, $b_5 = [x^3, y^2, z]$ are multisets over $A = \{x, y, z\}$. b_1 is the empty multiset, b_2 and b_3 both consist of three elements, and $b_4 = b_5$, i.e., the ordering of elements is irrelevant and a more compact notation may be used for repeating elements.

The standard set operators can be extended to multisets, e.g., $x \in b_2$, $b_2 \uplus b_3 = b_4$, $b_5 \setminus b_2 = b_3$, $|b_5| = 6$, etc. $\{a \in b\}$ denotes the set with all elements a for which $b(a) \geq 1$. $[f(a) \mid a \in b]$ denotes the multiset where element $f(a)$ appears $\sum_{x \in b | f(x) = f(a)} b(x)$ times.

3.2 Event Logs

As indicated earlier, *event logs* serve as the starting point for process mining. An event log is a multiset of *traces*. Each trace describes the life-cycle of a particular *case* (i.e., a *process instance*) in terms of the *activities* executed.

Definition 1 (Trace, Event Log). *Let A be a set of activities. A trace $\sigma \in A^*$ is a sequence of activities. $L \in \mathcal{B}(A^*)$ is an event log, i.e., a multiset of traces.*

An event log is a *multiset* of traces because there can be multiple cases having the same trace. In this simple definition of an event log, an event refers to just an *activity*. Often event logs may store additional information about events. For example, many process mining techniques use extra information such as the *resource* (i.e., person or device) executing or initiating the activity, the *timestamp* of the event, or *data elements* recorded with the event. In this paper, we abstract from such information. However, the results presented in this paper can easily be extended to event logs with more information.

An example log is $L_1 = [\langle a,b,c,d,e,g\rangle^{30}, \langle a,b,d,c,e,g\rangle^{20}, \langle a,b,c,d,e,f,b,c,d,e,g\rangle^{5}, \langle a,b,d,c,e,f,b,c,d,e,g\rangle^{3}, \langle a,b,c,d,e,f,b,d,c,e,g\rangle^{2}]$. L_1 contains information about 60 cases, e.g., 30 cases followed trace $\langle a,b,c,d,e,g\rangle$.

Definition 2 (Projection). *Let A be a set and $X \subseteq A$ a subset. $\restriction_X \in A^* \to X^*$ is a projection function and is defined recursively: (a) $\langle\,\rangle\restriction_X = \langle\,\rangle$ and (b) for $\sigma \in A^*$ and $a \in A$: $(\sigma;\langle a\rangle)\restriction_X = \sigma\restriction_X$ if $a \notin X$ and $(\sigma;\langle a\rangle)\restriction_X = \sigma\restriction_X;\langle a\rangle$ if $a \in X$.*

The projection function is generalized to event logs, i.e., for some event log $L \in \mathcal{B}(A^*)$ and set $X \subseteq A$: $L\restriction_X = [\sigma\restriction_X \mid \sigma \in L]$. For event log L_1 define earlier: $L_1\restriction_{\{a,f,g\}} = [\langle a,g\rangle^{50}, \langle a,f,g\rangle^{10}]$.

3.3 Procedural Models

A wide variety of process modeling languages are used in the context of process mining, e.g., Petri nets, EPCs, C-nets, BPMN, YAWL, and UML activity diagrams [3, 31]. Most of these languages are *procedural* languages (also referred to as *imperative* languages). In this paper, we use Petri nets as a typical representative of such languages. However, the ideas can easily be adapted to fit other languages. Later we will formalize selected distribution concepts in terms of Petri nets. Therefore, we introduce some standard notations.

Definition 3 (Petri Net). *A Petri net is tuple $PN = (P,T,F)$ with P the set of places, T the set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ the flow relation.*

Figure 9 shows an example Petri net. The state of a Petri net, called *marking*, is a multiset of places indicating how many *tokens* each place contains. $[in]$ is the initial marking shown in Fig. 9. Another potential marking is $[c2^{10}, c3^5, c5^5]$. This is the state with ten tokens in $c2$, five tokens in $c3$, and five tokens in $c5$.

Definition 4 (Marking). *Let $PN = (P,T,F)$ be Petri net. A marking M is a multiset of places, i.e., $M \in \mathcal{B}(P)$.*

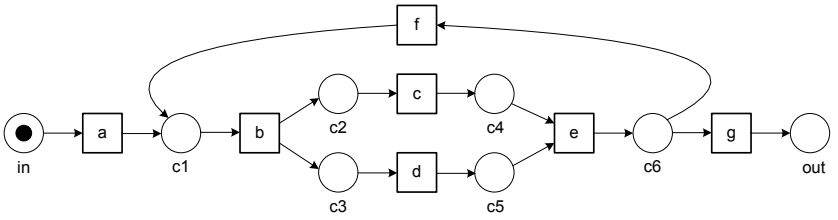


Fig. 9. A Petri net $PN = (P, T, F)$ with $P = \{in, c1, c2, c3, c4, c5, c6, out\}$, $T = \{a, b, c, d, e, f, g\}$, and $F = \{(in, a), (a, c1), (c1, b), \ldots, (g, out)\}$

As usual we define the preset and postset of a node (place or transition) in the Petri net graph. For any $x \in P \cup T$, $\bullet x = \{y \mid (y, x) \in F\}$ (input nodes) and $x\bullet = \{y \mid (x, y) \in F\}$ (output nodes).

A transition $t \in T$ is *enabled* in marking M, denoted as $M[t\rangle$, if each of its input places $\bullet t$ contains at least one token. Consider the Petri net in Fig. 9 with $M = [c3, c4]$: $M[e\rangle$ because *both* input places are marked.

An enabled transition t may *fire*, i.e., one token is removed from each of the input places $\bullet t$ and one token is produced for each of the output places $t\bullet$. Formally: $M' = (M \setminus \bullet t) \uplus t\bullet$ is the marking resulting from firing enabled transition t in marking M. $M[t\rangle M'$ denotes that t is enabled in M and firing t results in marking M'. For example, $[in][a\rangle[c1]$ and $[c1][b\rangle[c2, c3]$ for the net in Fig. 9.

Let $\sigma = \langle t_1, t_2, \ldots, t_n\rangle \in T^*$ be a sequence of transitions. $M[\sigma\rangle M'$ denotes that there is a set of markings $M_0, M_1, \ldots, M_n$ such that $M_0 = M$, $M_n = M'$, and $M_i[t_{i+1}\rangle M_{i+1}$ for $0 \leq i < n$. A marking M' is *reachable* from M if there exists a σ such that $M[\sigma\rangle M'$. For example, $[in][\sigma\rangle[out]$ for $\sigma = \langle a, b, c, d, e, g\rangle$.

Definition 5 (Labeled Petri Net). *A labeled Petri net $PN = (P, T, F, T_v)$ is a Petri net (P, T, F) with visible labels $T_v \subseteq T$. Let $\sigma_v = \langle t_1, t_2, \ldots, t_n\rangle \in T_v^*$ be a sequence of visible transitions. $M[\sigma_v \triangleright M'$ if and only if there is a sequence $\sigma \in T^*$ such that $M[\sigma\rangle M'$ and the projection of σ on T_v yields σ_v (i.e., $\sigma_v = \sigma{\restriction_{T_v}}$).*

If we assume $T_v = \{a, e, f, g\}$ for the Petri net in Fig. 9, then $[in][\sigma_v \triangleright [out]$ for $\sigma_v = \langle a, e, f, e, f, e, g\rangle$ (i.e., b, c, and d are invisible).

In the context of process mining, we always consider processes that start in an initial state and end in a well-defined end state. For example, given the net in Fig. 9 we are interested in firing sequences starting in $M_i = [in]$ and ending in $M_o = [out]$. Therefore, we define the notion of a *system net*.

Definition 6 (System Net). *A system net is a triplet $SN = (PN, M_i, M_o)$ where $PN = (P, T, F, T_v)$ is a Petri net with visible labels T_v, $M_i \in \mathcal{B}(P)$ is the initial marking, and $M_o \in \mathcal{B}(P)$ is the final marking.*

Given a system net, $\tau(SN)$ is the set of all possible visible full traces, i.e., firing sequences starting in M_i and ending in M_o projected onto the set of visible transitions.

Definition 7 (Traces). *Let $SN = (PN, M_i, M_o)$ be a system net. $\tau(SN) = \{\sigma_v \mid M_i[\sigma_v\rhd M_o\}$ is the set of visible traces starting in M_i and ending in M_o.*

If we assume $T_v = \{a, e, g\}$ for the Petri net in Fig. 9, then $\tau(SN) = \{\langle a, e, g\rangle, \langle a, e, e, g\rangle, \langle a, e, e, e, g\rangle, \ldots\}$.

The Petri net in Fig. 9 has a designated source place (*in*), a designated source place (*out*), and all nodes are on a path from *in* to *out*. Such nets are called *WF-nets* [1, 6].

Definition 8 (WF-net). *$WF = (PN, in, T_i, out, T_o)$ is a workflow net (WF-net) if*

- *$PN = (P, T, F, T_v)$ is a labeled Petri net,*
- *$in \in P$ is a source place such that $\bullet in = \emptyset$ and $in\bullet = T_i$,*
- *$out \in P$ is a sink place such that $out\bullet = \emptyset$ and $\bullet out = T_o$,*
- *$T_i \subseteq T_v$ is the set of initial transitions and $\bullet T_i = \{in\}$,*
- *$T_o \subseteq T_v$ is the set of final transitions and $T_o\bullet = \{out\}$, and*
- *all nodes are on some path from source place in to sink place out.*

WF-nets are often used in the context of business process modeling and process mining. Compared to the standard definition of WF-nets [1, 6] we added the requirement that the initial and final transitions need to be visible.

A WF-net $WF = (PN, in, T_i, out, T_o)$ defines the system $SN = (PN, M_i, M_o)$ with $M_i = [in]$ and $M_o = [out]$. Ideally WF-nets are also *sound*, i.e., free of deadlocks, livelocks, and other anomalies [1, 6]. Formally, this means that it is possible to reach M_o from any state reachable from M_i.

Process models discovered using existing process mining techniques may be unsound. Therefore, we cannot assume/require all WF-nets to be sound.

3.4 Declarative Models

Procedural process models (like Petri nets) take an "inside-to-outside" approach, i.e., all execution alternatives need to be specified explicitly and new alternatives must be explicitly added to the model. Declarative models use an "outside-to-inside" approach: anything is possible unless explicitly forbidden. Declarative models are particularly useful for conformance checking. Therefore, we elaborate on *Declare*. Declare is both a language (in fact a family of languages) and a fully functional WFM system [8, 24].

Declare uses a graphical notation and its semantics are based on LTL (Linear Temporal Logic) [8]. Figure 10 shows a Declare specification consisting of eight constraints. The construct connecting activities b and c is a so-called *non-coexistence constraint*. In terms of LTL this constraint means "$\neg((\Diamond b) \wedge (\Diamond c))$"; $\Diamond b$ and $\Diamond c$ cannot both be true, i.e., it cannot be the case that both b and c happen for the same case. There is also a non-coexistence constraint preventing the execution of both g and h for the same case. There are three *precedence constraints*. The semantics of the precedence constraint connecting a to b can also be expressed in terms of LTL: "$(\neg b)\ W\ a$", i.e., b should not happen before a has

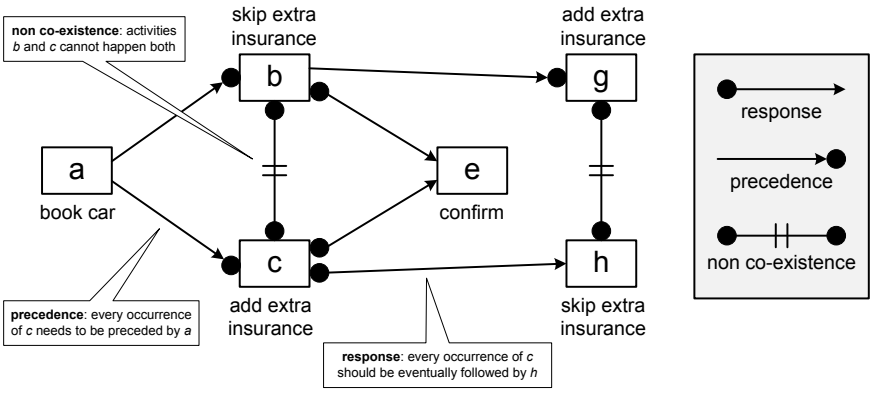


Fig. 10. Example of a Declare model consisting of six activities and eight constraints

happened. Since the weak until (W) is used in "$(\neg b)\ W\ a$", traces without any a and b events also satisfy the constraint. Similarly, g should not happen before b has happened: "$(\neg g)\ W\ b$". There are three *response constraints*. The LTL formalization of the precedence constraint connecting b to e is "$\Box(b \Rightarrow (\Diamond e))$", i.e., every occurrence of b should eventually be followed by e. Note that the behavior generated by the WF-net in Fig. 1 satisfies all constraints specified in the Declare model, i.e., none of the eight constraints is violated by any of the traces. However, the Declare model shown in Figure 10 allows for all kinds of behaviors not possible in Fig. 1. For example, trace $\langle a, a, b, e, e, g, g \rangle$ is allowed. Whereas in a procedural model, everything is forbidden unless explicitly enabled, a declarative model allows for anything unless explicitly forbidden. For processes with a lot of flexibility, declarative models are more appropriate [8, 24].

In [5] it is described how Declare/LTL constraints can be checked for a given log. This can also be extended to the on-the-fly conformance checking. Consider some running case having a partial trace $\sigma_p \in A^*$ listing the events that have happened thus far. Each constraint c is in one of the following states for σ_p:

- *Satisfied*: the LTL formula corresponding to c evaluates to true for the partial trace σ_p.
- *Temporarily violated*: the LTL formula corresponding to c evaluates to false for σ_p, however, there is a longer trace σ'_p that has σ_p as a prefix and for which the LTL formula corresponding to c evaluates to true.
- *Permanently violated*: the LTL formula corresponding to c evaluates to false for σ_p and all its extensions, i.e., there is no σ'_p that has σ_p as a prefix and for which the LTL formula evaluates to true.

These three notions can be lifted from the level of a *single constraint* to the level of a *complete Declare specification*, e.g., a Declare specification is *satisfied* for a case if all of its constraints are satisfied. This way it is possible to check conformance on-the-fly and generate warnings the moment constraints are permanently/temporarily violated [3].

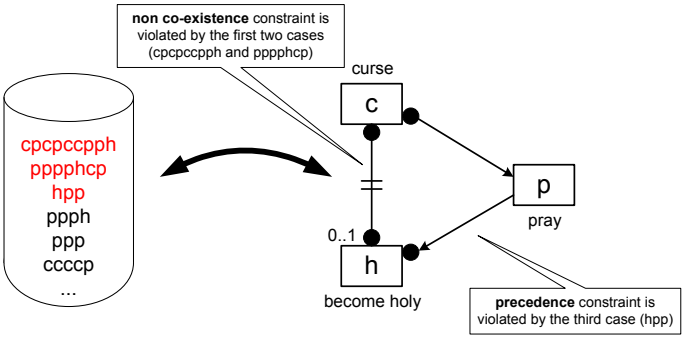


Fig. 11. Conformance checking using a declarative model.

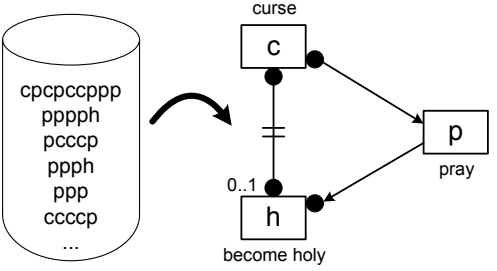


Fig. 12. Discovering a declarative model

We use the smaller example shown in Fig. 11 to illustrate conformance checking in the context of Declare. The process model shows four constraints: the same person cannot "curse" and "become holy" (non-coexistence constraint), after one "curses" one should eventually "pray" (response constraint), one can only "become holy" after having "prayed" at least once (precedence constraint), and activity h ("become holy") can be executed at most once (cardinality constraint).

Two of the four constraints are violated by the event log shown in Fig. 11. The first two traces/persons cursed and became holy at the same time. The third trace/person became holy without having prayed before.

Conformance checking can be distributed easily for declarative models. One can partition the log *vertically* and simply check per computing node all constraints on the corresponding sublog. One can also partition the set of constraints. Each node of the computer network is responsible for a subset of the constraints and uses a log projected onto the relevant activities, i.e., the event log is distributed *horizontally*. In both cases, it is easy to aggregate the results into overall diagnostics.

Figure 12 illustrates the discovery of Declare constraints from event logs [21]. A primitive discovery approach is to simply investigate a large collection of candidate constraints using conformance checking. This can be distributed vertically

or horizontally as just described. It is also possible to use smarter approaches using the "interestingness" of potential constraints. Here ideas from distributed association rule mining [13] can be employed.

4 Measuring Conformance

Conformance checking techniques can be used to investigate how well an event log $L \in \mathcal{B}(A^*)$ and the behavior allowed by a model fit together. Figure 4 shows an example where deviations between an event log and Petri net are diagnosed. Figure 11 shows a similar example but now using a Declare model. Both examples focus on a particular conformance notion: *fitness*. A model with good fitness allows for most of the behavior seen in the event log. A model has a *perfect fitness* if all traces in the log can be replayed by the model from beginning to end. This notion can be formalized as follows.

Definition 9 (Perfectly Fitting Log). *Let $L \in \mathcal{B}(A^*)$ be an event log and let $SN = (PN, M_i, M_o)$ be a system net. L is perfectly fitting SN if and only if $\{\sigma \in L\} \subseteq \tau(SN)$.*

The above definition assumes a Petri net as process model. However, the same idea can be operationalized for Declare models [5], i.e., for each constraint and every case the corresponding LTL formula should hold.

Consider two event logs $L_1 = [\langle a,c,d,g\rangle^{30}, \langle a,d,c,g\rangle^{20}, \langle a,c,d,f,c,d,g\rangle^{5}, \langle a,d,c,f,c,d,g\rangle^{3}, \langle a,c,d,f,d,c,g\rangle^{2}]$ and $L_2 = [\langle a,c,d,g\rangle^{8}, \langle a,c,g\rangle^{6}, \langle a,c,f,d,g\rangle^{5}]$ and the system net SN of the WF-net depicted in Fig. 9 with $T_v = \{a,c,d,f,g\}$. Clearly, L_1 is perfectly fitting SN whereas L_2 is not. There are various ways to quantify fitness [3, 4, 11, 19, 23, 25–27], typically on a scale from 0 to 1 where 1 means perfect fitness. To measure fitness, one needs to *align* traces in the event log to traces of the process model. Some example alignments for L_2 and SN:

$$\gamma_1 = \begin{array}{|c|c|c|c|}\hline a & c & d & g \\ \hline a & c & d & g \\ \hline \end{array} \quad \gamma_2 = \begin{array}{|c|c|c|c|}\hline a & c & \gg & g \\ \hline a & c & d & g \\ \hline \end{array} \quad \gamma_3 = \begin{array}{|c|c|c|c|c|}\hline a & c & f & d & g \\ \hline a & c & \gg & d & g \\ \hline \end{array} \quad \gamma_4 = \begin{array}{|c|c|c|c|c|c|c|}\hline a & c & \gg & f & d & \gg & g \\ \hline a & c & d & f & d & c & g \\ \hline \end{array}$$

The top row of each alignment corresponds to "moves in the log" and the bottom row corresponds to "moves in the model". If a move in the log cannot be mimicked by a move in the model, then a "$\gg$" ("no move") appears in the bottom row. For example, in γ_3 the model is unable to do f in-between c and d. If a move in the model cannot be mimicked by a move in the log, then a "$\gg$" ("no move") appears in the top row. For example, in γ_2 the log did not do a d move whereas the model has to make this move to enable g and reach the end. Given a trace in the event log, there may be many possible alignments. The goal is to find the alignment with the least number of $\gg$ elements, e.g., γ_3 seems better than γ_4. Finding a optimal alignment can be viewed as an optimization problem [4, 11]. After selecting an optimal alignment, the number of $\gg$ elements can be used to quantify fitness.

Fitness is just one of the four basic conformance dimensions defined in [3]. Other quality dimensions for comparing model and log are *simplicity*, *precision*, and *generalization*.

The *simplest* model that can explain the behavior seen in the log is the best model. This principle is known as Occam's Razor. There are various metrics to quantify the complexity of a model (e.g., size, density, etc.).

The *precision* dimension is related to the desire to avoid "underfitting". It is very easy to construct an extremely simple Petri net ("flower model") that is able to replay all traces in an event log (but also any other event log referring to the same set of activities). See [4, 25–27] for metrics quantifying this dimension.

The *generalization* dimension is related to the desire to avoid "overfitting". In general it is undesirable to have a model that only allows for the exact behavior seen in the event log. Remember that the log contains only example behavior and that many traces that are possible may not have been seen yet.

Conformance checking can be done for various reasons. First of all, it may be used to audit processes to see whether reality conforms to some normative of descriptive model [7]. Deviations may point to fraud, inefficiencies, and poorly designed or outdated procedures. Second, conformance checking can be used to evaluate the performance of a process discovery technique. In fact, genetic process mining algorithms use conformance checking to select the candidate models used to create the next generation of models [23].

5 Example: Horizontal Distribution Using Passages

The vertical distribution of process mining tasks is often fairly straightforward; just partition the event log and run the usual algorithms on each sublog residing at a particular node in the computer network. The horizontal partitioning of event logs is more challenging, but potentially very attractive as the focus of analysis can be limited to a few activities per node. Therefore, we describe a generic distribution approach based on the notion of *passages*.

5.1 Passages in Graphs

A *graph* is a pair $G = (N, E)$ comprising a set N of *nodes* and a set $E \subseteq N \times N$ of *edges*. A Petri net (P, T, F) can be seen as a particular graph with nodes $N = P \cup T$ and edges $E = F$. Like for Petri nets, we define preset $\bullet n = \{n' \in N \mid (n', n) \in E\}$ (direct predecessors) and postset $n\bullet = \{n' \in N \mid (n, n') \in E\}$ (direct successors). This can be generalized to sets, i.e., for $X \subseteq N$: $\bullet X = \cup_{n \in X} \bullet n$ and $X\bullet = \cup_{n \in X} n\bullet$.

To decompose process mining problems into smaller problems, we partition process models using the notion *passages*. A passage is a pair of non-empty sets of nodes (X, Y) such that the set of direct successors of X is Y and the set of direct predecessors of Y is X.

Definition 10 (Passage). *Let $G = (N, E)$ be a graph. $P = (X, Y)$ is a passage if and only if $\emptyset \neq X \subseteq N$, $\emptyset \neq Y \subseteq N$, $X\bullet = Y$, and $X = \bullet Y$. $pas(G)$ is the set of all passages of G.*

Consider the sets $X = \{a, b, c, e, f, g\}$ and $Y = \{c, d, g, h, i\}$ in the graph fragment shown in Fig. 13. (X, Y) is a passage. As indicated, there may be no edges leaving from X to nodes outside Y and there may be no edges into Y from nodes outside X.

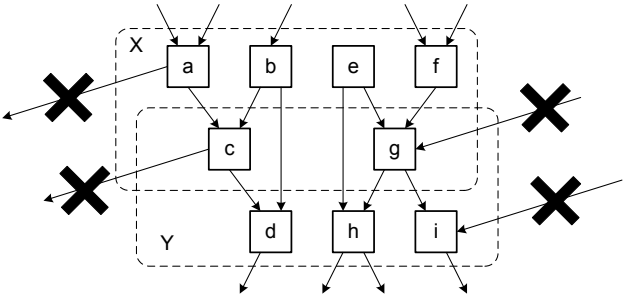


Fig. 13. (X, Y) is a passage because $X\bullet = \{a, b, c, e, f, g\}\bullet = \{c, d, g, h, i\} = Y$ and $X = \{a, b, c, e, f, g\} = \bullet\{c, d, g, h, i\} = \bullet Y$

Definition 11 (Operations on Passages). *Let $P_1 = (X_1, Y_1)$ and $P_2 = (X_2, Y_2)$ be two passages.*

- *$P_1 \leq P_2$ if and only if $X_1 \subseteq X_2$ and $Y_1 \subseteq Y_2$,*
- *$P_1 < P_2$ if and only if $P_1 \leq P_2$ and $P_1 \neq P_2$,*
- *$P_1 \cup P_2 = (X_1 \cup X_2, Y_1 \cup Y_2)$,*
- *$P_1 \setminus P_2 = (X_1 \setminus X_2, Y_1 \setminus Y_2)$.*

The union of two passages $P_1 \cup P_2$ is again a passage. The difference of two passages $P_1 \setminus P_2$ is a passage if $P_2 < P_1$. Since the union of two passages is again a passage, it is interesting to consider *minimal passages*. A passage is *minimal* if it does not "contain" a smaller passage.

Definition 12 (Minimal Passage). *Let $G = (N, E)$ be a graph with passages $pas(G)$. $P \in pas(G)$ is* minimal *if there is no $P' \in pas(G)$ such that $P' < P$. $pas_{min}(G)$ is the set of minimal passages.*

The passage in Figure 13 is not minimal. It can be split into the passages $(\{a, b, c\}, \{c, d\})$ and $(\{e, f, g\}, \{g, h, i\})$. An edge uniquely determines one minimal passage.

Lemma 1. *Let $G = (N, E)$ be a graph and $(x, y) \in E$. There is precisely one minimal passage $P_{(x,y)} = (X, Y) \in pas_{min}(G)$ such that $x \in X$ and $y \in Y$.*

Passages define an equivalence relation on the edges in a graph: $(x_1, y_1) \sim (x_2, y_2)$ if and only if $P_{(x_1,y_1)} = P_{(x_2,y_2)}$. For any $\{(x, y), (x', y), (x, y')\} \subseteq E$: $P_{(x,y)} = P_{(x',y)} = P_{(x,y')}$, i.e., $P_{(x,y)}$ is uniquely determined by x and $P_{(x,y)}$ is also uniquely determined by y. Moreover, $pas_{min}(G) = \{P_{(x,y)} \mid (x, y) \in E\}$.

5.2 Distributed Conformance Checking Using Passages

Now we show that it is possible to decompose and distribute conformance checking problems using the notion of *passages*. In order to do this we focus on the visible transitions and create the so-called *skeleton* of the process model. To define skeletons, we introduce the notation $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$ which states that there is a non-empty path σ from node x to node y where the set of intermediate nodes visited by path σ does not include any nodes in Q.

Definition 13 (Path). *Let $G = (N, E)$ be a graph with $x, y \in N$ and $Q \subseteq N$. $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$ if and only if there is a sequence $\sigma = \langle n_1, n_2, \ldots n_k \rangle$ with $k > 1$ such that $x = n_1$, $y = n_k$, for all $1 \leq i < k$: $(n_i, n_{i+1}) \in E$, and for all $1 < i < k$: $n_i \notin Q$. Derived notations:*

- *$x \overset{E\#Q}{\rightsquigarrow} y$ if and only if there exists a path σ such that $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$,*
- *$nodes(x \overset{E\#Q}{\rightsquigarrow} y) = \{n \in \sigma \mid \exists_{\sigma \in N^*}\ x \overset{\sigma:E\#Q}{\rightsquigarrow} y\}$, and*
- *for $X, Y \subseteq N$: $nodes(X \overset{E\#Q}{\rightsquigarrow} Y) = \cup_{(x,y)\in X\times Y}\ nodes(x \overset{E\#Q}{\rightsquigarrow} y)$.*

Definition 14 (Skeleton). *Let $PN = (P, T, F, T_v)$ be a labeled Petri net. The* skeleton *of PN is the graph $skel(PN) = (N, E)$ with $N = T_v$ and $E = \{(x, y) \in T_v \times T_v \mid x \overset{F\#T_v}{\rightsquigarrow} y\}$.*

Figure 14 shows the skeleton of the WF-net in Fig. 1 assuming that $T_v = \{a, b, c, d, e, f, l\}$. The resulting graph has four minimal passages.

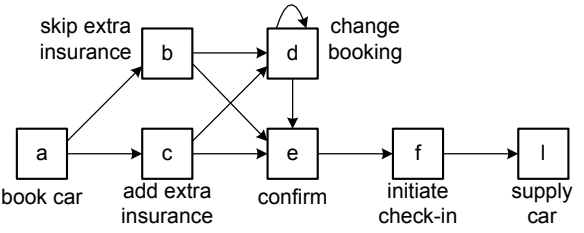


Fig. 14. The skeleton of the labeled Petri net in Fig. 1 (assuming that $T_v = \{a, b, c, d, e, f, l\}$). There are four minimal passages: $(\{a\}, \{b, c\})$, $(\{b, c, d\}, \{d, e\})$, $(\{e\}, \{f\})$, and $(\{f\}, \{l\})$.

Note that only the visible transitions T_v appear in the skeleton. For example, if we assume that $T_v = \{a, f, l\}$ in Fig. 1, then the skeleton is $(\{a, f, l\}, \{(a, f), (f, l)\})$ with only two passages $(\{a\}, \{f\})$ and $(\{f\}, \{l\})$.

If there are multiple minimal passages in the skeleton, we can decompose conformance checking problems into smaller problems *by partitioning the Petri net*

into net fragments and the event log into sublogs. Each passage (X, Y) defines one net fragment $PN^{(X,Y)}$ and one sublog $L\restriction_{X \cup Y}$. We will show that conformance can be checked per passage.

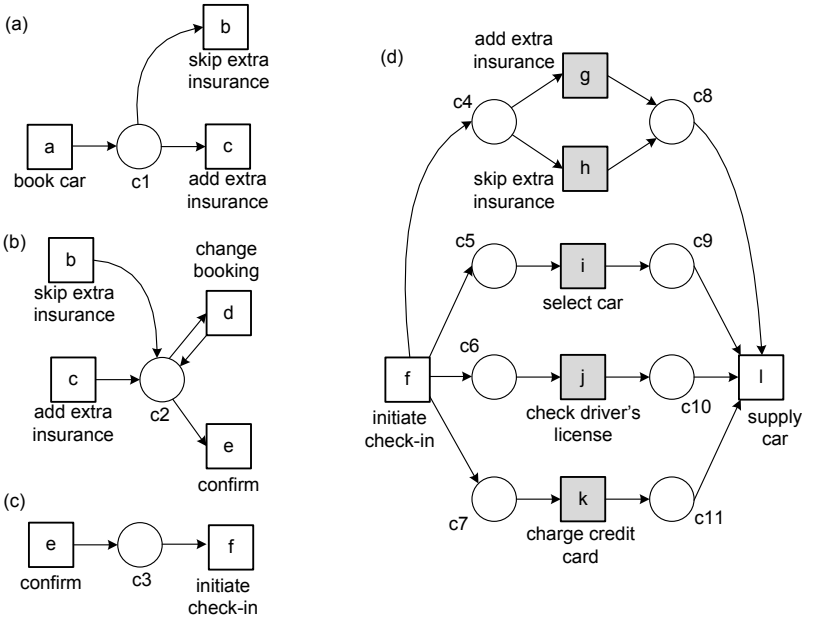


Fig. 15. Four net fragments corresponding to the four passages of the skeleton in Fig. 14: (a) $PN_1 = PN^{(\{a\},\{b,c\})}$, (b) $PN_2 = PN^{(\{b,c,d\},\{d,e\})}$, (c) $PN_3 = PN^{(\{e\},\{f\})}$, and (c) $PN_4 = PN^{(\{f\},\{l\})}$. The invisible transitions, i.e., the transitions in $T \setminus T_v$, are shaded.

Consider event log $L = [\langle a, b, e, f, l\rangle^{20}, \langle a, c, e, f, l\rangle^{15}, \langle a, b, d, e, f, l\rangle^{5}, \langle a, c, d, e, f, l\rangle^{3}, \langle a, b, d, d, e, f, l\rangle^{2}]$, the WF-net PN shown in Fig. 1 with $T_v = \{a, b, c, d, e, f, l\}$, and the skeleton shown in Fig. 14. Based on the four passages, we define four net fragments PN_1, PN_2, PN_3 and PN_4 as shown in Fig. 15. Moreover, we define four sublogs: $L_1 = [\langle a, b\rangle^{27}, \langle a, c\rangle^{18}]$, $L_2 = [\langle b, e\rangle^{20}, \langle c, e\rangle^{15}, \langle b, d, e\rangle^{5}, \langle c, d, e\rangle^{3}, \langle b, d, d, e\rangle^{2}]$, $L_3 = [\langle e, f\rangle^{45}]$, and $L_4 = [\langle f, l\rangle^{45}]$. To check the conformance of the overall event log on the overall model, we check the conformance of L_i on PN_i for $i \in \{1, 2, 3, 4\}$. Since L_i is perfectly fitting PN_i for all i, we can conclude that L is perfectly fitting PN. This illustrates that conformance checking can indeed be decomposed. To formalize this result, we define the notion of a net fragment corresponding to a passage.

Definition 15 (Net Fragment). *Let $PN = (P, T, F, T_v)$ be a labeled Petri net. For any two sets of transitions $X, Y \subseteq T_v$, we define the net fragment $PN^{(X,Y)} = (P', T', F', T'_v)$ with:*

- $Z = nodes(X \overset{F\#T_v}{\rightsquigarrow} Y) \setminus (X \cup Y)$ *are the internal nodes of the fragment,*
- $P' = P \cap Z$,
- $T' = (T \cap Z) \cup X \cup Y$,
- $F' = F \cap ((P' \times T') \cup (T' \times P'))$, *and*
- $T'_v = X \cup Y$.

A process model can be decomposed into net fragments corresponding to minimal passages and an event log can be decomposed by projecting the traces on the activities in these minimal passages. The following theorem shows that conformance checking can be done per passage.

Theorem 1 (Distributed Conformance Checking). *Let $L \in \mathcal{B}(A^*)$ be an event log and let $WF = (PN, in, T_i, out, T_o)$ be a WF-net with $PN = (P, T, F, T_v)$. L is perfectly fitting system net $SN = (PN, [in], [out])$ if and only if*

- *for any $\langle a_1, a_2, \ldots a_k \rangle \in L$: $a_1 \in T_i$ and $a_k \in T_o$, and*
- *for any $(X, Y) \in pas_{min}(skel(PN))$: $L{\restriction_{X \cup Y}}$ is perfectly fitting $SN^{(X,Y)} = (PN^{(X,Y)}, [\,], [\,])$.*

For a formal proof, we refer to [2]. Although the theorem only addresses the notion of perfect fitness, other conformance notions can be decomposed in a similar manner. Metrics can be computed per passage and then aggregated into an overall metric.

Assuming a process model with many passages, the time needed for conformance checking can be reduced significantly. There are two reasons for this. First of all, as Theorem 1 shows, larger problems can be decomposed into a set of independent smaller problems. Therefore, conformance checking can be distributed over multiple computers. Second, due to the exponential nature of most conformance checking techniques, the time needed to solve "many smaller problems" is less than the time needed to solve "one big problem". Existing approaches use state-space analysis (e.g., in [27] the shortest path enabling a transition is computed) or optimization over all possible alignments (e.g., in [11] the A^* algorithm is used to find the best alignment). These techniques do *not* scale linearly in the number of activities. *Therefore, decomposition is useful even if the checks per passage are done on a single computer.*

5.3 Distributed Process Discovery Using Passages

As explained before, conformance checking and process discovery are closely related. Therefore, we can exploit the approach used in Theorem 1 for process discovery provided that some coarse *causal structure* (comparable to the skeleton in Section 5.2) is known. There are various techniques to extract such a causal structure, see for example the dependency relations used by the heuristic miner [29]. The causal structure defines a collection of passages and the detailed discovery can be done per passage. Hence, the discovery process can be distributed. The idea is illustrated in Fig. 16.

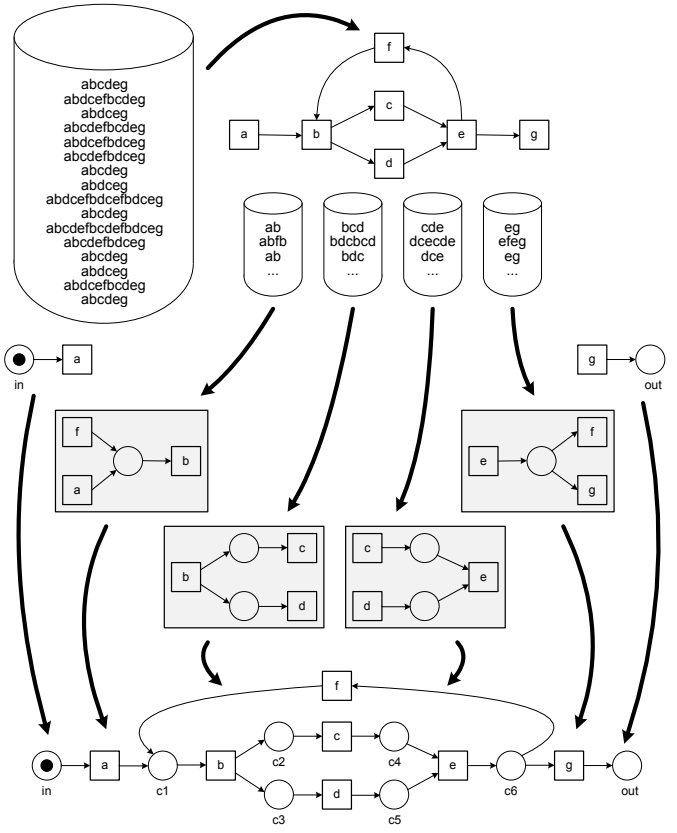


Fig. 16. Distributed discovery based on four minimal passages: $(\{a, f\}, \{b\})$, $(\{b\}, \{c, d\})$, $(\{c, d\}, \{e\})$, and $(\{e\}, \{f, g\})$. A process fragment is discovered for each passage. Subsequently, the fragments are merged into one overall process.

The approach is independent of the discovery algorithm used. The only assumption is that the casual structure can be determined upfront. See [2] for more details.

By decomposing the overall discovery problem into a collection of smaller discovery problems, it is possible to do a more refined analysis and achieve significant speed-ups. The discovery algorithm is applied to an event log consisting of just the activities involved in the passage under investigation. Hence, process discovery tasks can be distributed over a network of computers (assuming there are multiple passages). Moreover, most discovery algorithms are exponential in the number of activities. Therefore, the sequential discovery of all individual passages is still faster than solving one big discovery problem.

6 Conclusion

This paper provides an overview of the different mechanisms to distribute process mining tasks over a set of computing nodes. Event logs can be decomposed vertically and horizontally. In a vertically distributed event log, each case is analyzed by a designated computing node in the network and each node considers the whole process model (all activities). In a horizontally distributed event log, the cases themselves are partitioned and each node considers only a part of the overall process model. These distribution approaches are fairly independent of the mining algorithm and apply to both procedural and declarative languages. Most challenging is the horizontal distribution of event logs while using a procedural language. However, as shown in this paper, it is still possible to horizontally distribute process discovery and conformance checking tasks using the notion of passages.

Acknowledgments. The author would like to thank all that contributed to the ProM toolset. Many of their contributions are referred to in this paper. Special thanks go to Boudewijn van Dongen and Eric Verbeek (for their work on the ProM infrastructure), Carmen Bratosin (for her work on distributed genetic mining), Arya Adriansyah and Anne Rozinat (for their work on conformance checking), and Maja Pesic, Fabrizio Maggi, and Michael Westergaard (for their work on Declare).

References

1. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P.: Decomposing Process Mining Problems Using Passages. BPM Center Report BPM-11-19, BPMcenter.org (2011)
3. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Berlin (2011)
4. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.: Replaying History on Process Models for Conformance Checking and Performance Analysis. In: WIREs Data Mining and Knowledge Discovery (2012)
5. van der Aalst, W.M.P., de Beer, H.T., van Dongen, B.F.: Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In: Meersman, R., Tari, Z. (eds.) CoopIS/DOA/ODBASE 2005. LNCS, vol. 3760, pp. 130–147. Springer, Heidelberg (2005)
6. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of Workflow Nets: Classification, Decidability, and Analysis. Formal Aspects of Computing 23(3), 333–363 (2011)
7. van der Aalst, W.M.P., van Hee, K.M., van der Werf, J.M., Verdonk, M.: Auditing 2.0: Using Process Mining to Support Tomorrow's Auditor. IEEE Computer 43(3), 90–93 (2010)
8. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative Workflows: Balancing Between Flexibility and Support. Computer Science - Research and Development 23(2), 99–113 (2009)

9. van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. Software and Systems Modeling 9(1), 87–111 (2010)
10. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
11. Adriansyah, A., van Dongen, B., van der Aalst, W.M.P.: Conformance Checking using Cost-Based Fitness Analysis. In: Chi, C.H., Johnson, P. (eds.) IEEE International Enterprise Computing Conference (EDOC 2011), pp. 55–64. IEEE Computer Society (2011)
12. Agrawal, R., Gunopulos, D., Leymann, F.: Mining Process Models from Workflow Logs. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 469–483. Springer, Heidelberg (1998)
13. Agrawal, R., Shafer, J.C.: Parallel Mining of Association Rules. IEEE Transactions on Knowledge and Data Engineering 8(6), 962–969 (1996)
14. Jagadeesh Chandra Bose, R.P., van der Aalst, W.M.P., Žliobaitė, I., Pechenizkiy, M.: Handling Concept Drift in Process Mining. In: Mouratidis, H., Rolland, C. (eds.) CAiSE 2011. LNCS, vol. 6741, pp. 391–405. Springer, Heidelberg (2011)
15. Bratosin, C., Sidorova, N., van der Aalst, W.M.P.: Distributed Genetic Process Mining. In: Ishibuchi, H. (ed.) IEEE World Congress on Computational Intelligence (WCCI 2010), Barcelona, Spain, pp. 1951–1958. IEEE (July 2010)
16. Cannataro, M., Congiusta, A., Pugliese, A., Talia, D., Trunfio, P.: Distributed Data Mining on Grids: Services, Tools, and Applications. IEEE Transactions on Systems, Man, and Cybernetics, Part B 34(6), 2451–2465 (2004)
17. Carmona, J.A., Cortadella, J., Kishinevsky, M.: A Region-Based Algorithm for Discovering Petri Nets from Event Logs. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 358–373. Springer, Heidelberg (2008)
18. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology 7(3), 215–249 (1998)
19. Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust Process Discovery with Artificial Negative Events. Journal of Machine Learning Research 10, 1305–1340 (2009)
20. Hilbert, M., Lopez, P.: The World's Technological Capacity to Store, Communicate, and Compute Information. Science 332(60) (2011)
21. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-Guided Discovery of Declarative Process Models. In: Chawla, N., King, I., Sperduti, A. (eds.) IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011), Paris, France, pp. 192–199. IEEE (April 2011)
22. Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.: Big Data: The Next Frontier for Innovation, Competition, and Productivity. McKinsey Global Institute (2011)
23. Alves de Medeiros, A.K., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
24. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative Specification and Verification of Service Choreographies. ACM Transactions on the Web 4(1), 1–62 (2010)
25. Muñoz-Gama, J., Carmona, J.: A Fresh Look at Precision in Process Conformance. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 211–226. Springer, Heidelberg (2010)

26. Muñoz-Gama, J., Carmona, J.: Enhancing Precision in Process Conformance: Stability, Confidence and Severity. In: Chawla, N., King, I., Sperduti, A. (eds.) IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011), Paris, France. IEEE (April 2011)
27. Rozinat, A., van der Aalst, W.M.P.: Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems 33(1), 64–95 (2008)
28. Solé, M., Carmona, J.: Process Mining from a Basis of State Regions. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 226–245. Springer, Heidelberg (2010)
29. Weijters, A.J.M.M., van der Aalst, W.M.P.: Rediscovering Workflow Models from Event-Based Data using Little Thumb. Integrated Computer-Aided Engineering 10(2), 151–162 (2003)
30. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery using Integer Linear Programming. Fundamenta Informaticae 94, 387–412 (2010)
31. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Berlin (2007)

Model-Driven Techniques to Enhance Architectural Languages Interoperability

Davide Di Ruscio, Ivano Malavolta, Henry Muccini,
Patrizio Pelliccione, and Alfonso Pierantonio

University of L'Aquila, Dipartimento di Informatica
{davide.diruscio,ivano.malavolta,henry.muccini,
patrizio.pelliccione,alfonso.pierantonio}@univaq.it

Abstract. The current practice of software architecture modeling and analysis would benefit of using different architectural languages, each specialized on a particular view and each enabling specific analysis. Thus, it is fundamental to pursue architectural language interoperability. An approach for enabling interoperability consists in defining a transformation from each single notation to a pivot language, and vice versa. When the pivot assumes the form of a small and abstract kernel, extension mechanisms are required to compensate the *loss of information*. The aim of this paper is to enhance architectural languages interoperability by means of hierarchies of pivot languages obtained by systematically extending a *root* pivot language. Model-driven techniques are employed to support the creation and the management of such hierarchies and to realize the interoperability by means of model transformations. Even though the approach is applied to the software architecture domain, it is completely general.

1 Introduction

Architecture descriptions shall be developed to address multiple and evolving stakeholders concerns [1]. Being impractical to capture all concerns within a single, narrowly focused Architectural Language (AL) [2], i.e., a form of expression used for architecture description [1], we must accept the co-existence of different domain specific ALs, each one devoted to specific purposes. The use of various ALs requires interoperability among them since bridging the different descriptions to be kept consistent and coherent is of paramount relevance [3]. The need of interoperability at the architecture level is clearly demonstrated by international projects like Q-ImPrESS [4], and ATESST [5] where correspondences among different languages have to be created and maintained.

An approach for enabling interoperability among various notations which is recently getting consensus in different application domains (e.g., [6,7]) consists in organizing them into a star topology with a pivot language in its center: in these cases *interoperability is enabled by defining a transformation from each single notation to the pivot language, and vice versa*. Thus, the pivot language acts as a bridge between all the considered notations and avoids point-to-point direct transformations among them. While how to build a pivot language is still a craftsman activity, two different trends can be noted: (i) building a (rich) pivot language that contains each element required

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 26–42, 2012.

by any AL, like in the Q-Impress project, and (ii) building a (small) kernel pivot language containing a set of core elements common to most of the involved ALs, like in KLAPER [8]. On one hand, the adoption of a rich pivot language tends to reduce the *loss of information* in the pivot-based transformation process from one AL to another. On the other hand, the use of a kernel pivot may give rise to loss of information, since concepts in some of the ALs might be missing in the pivot language (due to the kernel pivot language minimality).

The use of a rich pivot is ideal when ALs have to be related under a closed-world-assumption, i.e., when the set of ALs to be used is a-priori defined. However, a rich pivot difficultly *scales* when new ALs are introduced in the star topology: the rich pivot has to be updated to cover newly introduced concepts. This is an error-prone task that could easily introduce inconsistencies within the pivot. In such a scenario, while the kernel pivot solution is more scalable (since the kernel pivot language is defined once forever and is AL-independent), the addition of new ALs increases the loss of information when new ALs introduce new concepts not included in the kernel pivot. When the closed-world-assumption decays, a new solution is needed to support the interoperability among various ALs while reducing as much as possible the loss of information. This calls for kernel extensions, each extension defined for dealing with specific stakeholder concerns. Moreover, the construction of kernels must be properly controlled to support their coexistence and reuse. The information that can be lost consists of concepts that potentially could be transformed from a source model and properly represented in a target one, but for some reason are neglected by the transformation process.

In this paper we present a Model-Driven Engineering (MDE) approach to enhance the interoperability among ALs by using extensible kernel pivots. The approach (i) encompasses a systematically defined *extension process* that, starting from a small kernel pivot language permits the automated construction of a hierarchy of kernel pivots, and (ii) provides mechanisms to transform from an AL to another by minimizing the loss of information; this is realized by passing through the most informative pivot kernel in the hierarchy for the considered ALs. The overall approach is general and, while applied to the software architecture domain, may be adopted in different domains.

The remaining of the paper is organized as follows. Section 2 highlights limitations and challenges of current pivot-based solutions. Section 3 describes the proposed kernel pivot extension mechanisms. Section 4 applies the approach to a case study in the automotive domain. Section 5 compares our work with related works. Section 6 concludes the paper and highlights future research directions.

2 Interoperability via Pivot Languages

It is becoming common practice to use different ALs to model or to analyze different architectural aspects of the system under development. The Q-Impress project, for example, enables interoperability through a rich pivot language that unifies common aspects of the used ALs. The ATESST project provides means to integrate different model-based tools to develop automotive embedded systems. In the domain of reliability modeling and prediction, Klaper is a kernel language which can be used as the starting point to carry out performance or reliability analysis. **DUALLY** [7] exploits model

transformation techniques and any transformation among ALs is defined by passing through A_0, a kernel pivot metamodel defined as general as possible.

All the projects and research efforts described above adopt a pivot solution for supporting the interoperability among different description languages. Figure 1 shows the main difference between the use of a rich pivot language and a kernel one: filled circles represent modeling concepts, solid lines denote correspondences among AL and pivot language concepts, and finally dashed boxes and dashed lines represent added ALs and correspondences, respectively. A rich pivot language is built with the aim of including the highest number of concepts contemplated by all the interoperating ALs. As shown in Figure 1.a, each concept in any AL finds its correspondence with a rich pivot language element. Differently, a kernel language contains only a core set of concepts (as shown in Figure 1.b), and is kept as small as possible. Such a difference has positive and negative impacts on the way interoperability is realized. In the following we provide a summary of the main strengths and limitations of both solutions.

Interoperability Accuracy: the rich pivot is built with the intent to match any concept coming from the interoperating ALs. Thus, in principle, as soon as a correspondence exists among two ALs, it is caught by the pivot-based transformation. The kernel language solution, being minimal, may instead discard some correspondence, thus limiting the interoperability accuracy. For instance, see `a1` and `a2` in Figure 1.b: while a correspondence among them is found in the rich pivot, it is missing in the kernel-based solution. Information loss is thus introduced. The kernel-based approach is particularly limiting when domain-specific ALs are introduced in the star topology. *Overall*: the rich pivot solution is more accurate;

Pivot Scalability: as soon as a new AL has to be considered, the rich pivot needs to be revised in order to avoid information loss. As shown in Figure 1.a, the insertion of AL_4 implies the addition of the link between AL_4 and the already existing element `b1` in the rich pivot, and the addition of `b2`. This may require a strong revision of the entire rich pivot to solve possible conflicts and to avoid inconsistencies. When AL_4 is added to the kernel language in Figure 1.b, instead, only a new correspondence with `b1` is created. *Overall*: the kernel language approach scales better.

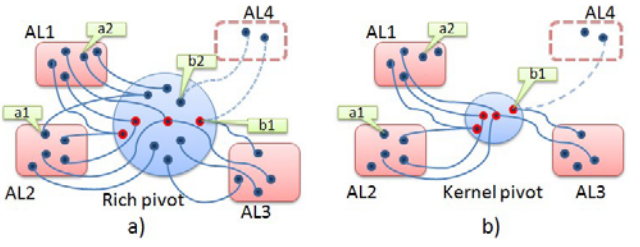


Fig. 1. Interoperability via a: a) rich pivot, b) kernel pivot

In summary, the rich pivot solution is more accurate in terms of interoperability correspondences, but it is less scalable and might require adjustments when a new notation is included. Contrariwise, the kernel solution shows complementary strengths and limitations. *A new solution is needed to support both interoperability accuracy and pivot scalability.*

An approach that is being used consists in making the kernel pivot *extensible*, thus adaptable to new ALs. Language extensibility in the software architecture domain has

been adopted in the xADL [9] XML-based architecture description language (based on XML extension mechanisms), in AADL [10] (through its annexes), in UML (with its profiles), and in our approach for ALs interoperability named **DUALLY** [7]. However, **DUALLY**, which is at the best of our knowledge the most mature framework to support interoperability among various ALs, has shown a certain number of shortcomings. *Firstly*, it is not clear how to manage the extension process when two (or more) extensions are required. Let us suppose that both real-time and behavior extensions are needed. So far, three alternative solutions can be applied: i) extend the kernel with real-time concepts first, then with behavior, ii) extend the kernel with behavior concepts first, then with real-time ones, iii) extend the kernel with both concepts at the same time. The three scenarios may produce different kernel pivots, and so far there is no guideline on how to manage such a multiple extension. *Secondly*, current solutions tend to create ad-hoc extensions, not engineered to be reusable. Even when applying scenarios i) or ii) above, the intermediate kernels are typically lost and not stored for reuse. The extension itself is not considered as a first class element, but simply as an improvement to the original pivot.

The approach we propose in this paper satisfies the requirements of i) a systematic extension process, which provides clear guidelines on how and what to extend, ii) a compositional and reuse-oriented approach, where kernels are re-used and extended, iii) supporting both interoperability accuracy and pivot scalability.

3 The Extension Mechanisms

In this section we propose the mechanisms to extend an existing kernel A with a kernel extension e. In our approach the extension e is a metamodel, that can be re-used for extending different kernels. The proposed mechanisms rely on the adoption of weaving models [11] which relate a kernel A with an extension e. A weaving model wm contains links between elements of a kernel A and elements of an extension e.

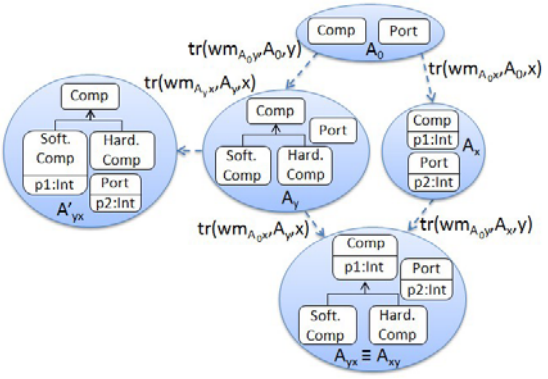


Fig. 2. Example of extensions of A_0

The generation of a kernel A_e, which is an extension of A with e, is performed by executing a transformation tr. tr is defined once forever and applies the extension e to A according to the extension operators used in wm (see Section 3.1). Figure 2 shows a small fragment of A_0 consisting of the metaclasses `Comp` and `Port` that represent a generic component and port, respectively (see [7] for a complete description of A_0).

Let us assume that y is a kernel extension containing the metaclasses `SoftComp` and `HardComp` to model software and hardware components, respectively. This extension can be applied to A_0 by means of

the transformation tr which takes as input the weaving model wm_{A_0y}, the kernel A_0, and the extension y, and generates the new kernel A_y. The kernel A_y is shown in Figure 2 and contains the generic component concept specialized in software and hardware components. Let us assume also that x is another extension consisting of the performance annotations $p1$ and $p2$. This extension can be applied to A_0 by means of the transformation tr which takes as input another weaving model wm_{A_0x}, the kernel A_0, and the extension x. The obtained kernel called A_x is shown in Figure 2 and represents an extension of A_0 in which the $p1$ annotation is added to `Comp` and the $p2$ annotation is added to `Port`.

As previously said, weaving models are used to apply given extensions to existing kernels by specifying the metaclasses which are involved in the operation. Formally, a weaving model can be defined as in Def. 1.

Definition 1 (Weaving Model). *Let $\mathbb{A}$ be the set of all the possible kernels, let $\mathbb{E}$ be the set of all the possible extensions, and let $\mathbb{W}$ be the set of all the possible weaving models. We denote with $wm_{Ae}\in\mathbb{W}$ a weaving model defined between the kernel $A\in\mathbb{A}$ and the extension $e\in\mathbb{E}$. A weaving model $wm_{Ae}=\{wl^1_{Ae}, wl^2_{Ae}, \cdots, wl^n_{Ae}\}$ can be seen as a set of weaving links each establishing a correspondence between elements of A and elements of e. Each link is realized by means of extension operators.*

Referring to Figure 2, the weaving model wm_{A_0x} defined for A_0 can be used also to extend A_y, since A_y contains the metaclasses involved in wm_{A_0x}. In fact, A_y contains the metaclasses `Comp` and `Port` which are considered in the weaving model wm_{A_0x} to attach the annotation $p1$ to `Comp`, and $p2$ to `Port`. In the same way, wm_{A_0y} can be used to extend A_x by applying the extension y to the metaclass `Comp`, and specializes it with the metaclasses `SoftComp` and `HardComp`. These two independent extension journeys converge in a kernel called A_{yx} or A_{xy}. Focusing on the left-hand side of Figure 2, the weaving model wm_{A_yx} is another application of the extension x to the kernel A_y to add the annotation $p2$ to `Port` and the annotation $p1$ to `SoftComp`. In this case we obtain a kernel different from A_{xy}. Specifically, this kernel permits to add $p1$ exclusively to software components.

Extension hierarchies, like the one in Figure 2, contain three types of elements: kernels, extensions, and weaving models that apply extensions to kernels. In order to regulate how kernels and extensions can be involved in specific weaving models, we make use of a type system for kernels and extensions. In other words, a weaving model defined for a kernel can be re-used also for applying extensions to other kernels, under the assumption that these kernels have the metaclasses involved in the weaving model. Def. 2 defines our notion of model type substitutability, which is based on the following notion of model typing: the type of a model is defined "as a set of MOF classes (and, of course, the references that they contain)" [12]. We denote with $\mathbb{T}$ the set of all the possible model types. In our context $\mathbb{T}$ can be partitioned in $\mathbb{T}^{\mathbb{A}}$ and $\mathbb{T}^{\mathbb{E}}$ which denote the types of kernels and extensions, respectively.

Definition 2 (Model Type Substitutability). *Let $T_A\in\mathbb{T}^{\mathbb{A}}$ be the type of a given kernel A, and let $T_e\in\mathbb{T}^{\mathbb{E}}$ be the type of an extension e, then a weaving model wm_{Ae} can be used by the model transformation tr to extend a kernel typed with either T_A or any of its subtypes.*

In our context subtyping depends on a type's hierarchy obtained by means of the extension mechanism that produces a kernel typed T_B by exclusively adding new elements to an existing one, typed T_A (i.e., the deletion of elements from a kernel is not allowed). It is worth mentioning that our extension mechanism ensures that all the elements of an extension e are added to the kernel being extended. This type hierarchy introduces a *strict partial order* $<$ among kernel types: $T_A<T_B$ if T_B is obtained by extending T_A and then T_B can be substituted to T_A. Figure 3 is a generalization of Figure 2 and shows a simple hierarchy of extensions involving a generic kernel A_k and two extensions called x and y. The kernel extensions are regulated by four different weaving models (wm_{A_kx}, wm_{A_ky}, wm_{A_xy}, and wm_{A_yx}), thus producing five different new kernels. More specifically, A_x and A_y are obtained extending A_k with x and y and by means of the weaving models wm_{A_kx} and wm_{A_ky}, respectively. The weaving model wm_{A_kx} takes as input a kernel typed T_{A_k} and the extension x typed T_x. Similarly, the weaving model wm_{A_ky} takes as input a kernel typed T_{A_k} and the extension y typed T_y.

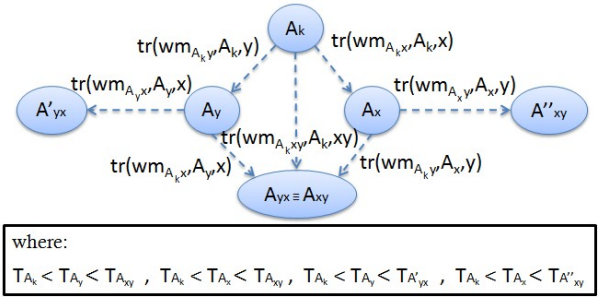


Fig. 3. A hierarchy of kernels

Let us focus now on A_x which is extended by applying the extension y in two different ways. The first way considers the weaving model wm_{A_xy} used by tr to apply the extension typed T_y to elements of a kernel typed T_{A_x}. This kernel contains the elements of A_k and those of x added by using wm_{A_kx}. The weaving model wm_{A_xy} can affect all of them since it considers a kernel typed T_{A_x}. This is not the case of wm_{A_kx}, which can only operate on elements of A_k. This justifies why the sequential compositions $tr(wm_{A_ky}, tr(wm_{A_kx},A_k,x), y)$ and $tr(wm_{A_kx}, tr(wm_{A_ky},A_k,y), x)$ lead to the same target metamodel A_{xy}, i.e., there is a confluence in the extension journeys. The generation of the target metamodel A_{xy} is performed by using a new weaving model wm_{A_kxy} which is the union of wm_{A_kx} and wm_{A_ky}. The execution of $tr(wm_{A_kxy},A_k,xy)$, where xy is a metamodel consisting of the union of the elements of x and y, produces A_{xy}. Formally, the union of two weaving models is defined as in Def. 3.

Definition 3 (Union of Weaving Models). *Let $wm_{Ax}\in\mathbb{W}$ a weaving model defined between the kernel $A\in\mathbb{A}$ and the extension $x\in\mathbb{E}$, and $wm_{Ax}=\{wl^1_{Ax},wl^2_{Ax},\cdots,wl^n_{Ax}\}$. Let $wm_{Ay}\in\mathbb{W}$ a weaving model defined between the kernel $A\in\mathbb{A}$ and the extension $y\in\mathbb{E}$, and $wm_{Ay}=\{wl^1_{Ay},wl^2_{Ay},\cdots,wl^m_{Ay}\}$. The weaving models union $wm_{Ax}\cup wm_{Ay}=\{wl^1_{Ax}, wl^2_{Ax}, \cdots,wl^n_{Ax},wl^1_{Ay},wl^2_{Ay},\cdots,wl^m_{Ay}\}$ is the set of all the weaving links in wm_{Ax} and wm_{Ay}.*

It is important to note that in general the confluence cannot be ensured since it depends on how the extensions have been applied, i.e., on the involved weaving models. In the following we explain why in our approach we have a confluence (see Section 3.1) and

how to identify transformation paths from one AL to another by passing through the kernels hierarchy (see Section 3.2).

3.1 Extension Operators

The extension operators used to create weaving models are *Inherit*, *Reference*, *Expand*, and *Match*. These operators are defined by constraining the composition operators presented in [13] to exclusively enable extensions and avoid conflicts when structural features of the kernel and the extension being applied overlap. They always extend a kernel and then, in case of conflicts during the extension, the kernel element will be the one to be considered. Each operator is always applied on two metaclasses (one belonging to the kernel and one to the extension) that we refer to as source (*s*) and target (*t*) in the remainder of this section. The application of the operators consists of executing the transformation tr that, as explained before, takes as input a weaving model, a kernel, and an extension, and produces an extended kernel according to the applied operators. The extension operators are:

Inherit: This operator specifies that the concept *s* will be a subtype of *t* in the resulting extended kernel. If its application results in a cycle in the inheritance tree, then it is not executed and a warning is raised. The *t* metaclass must belong to the kernel metamodel.

Reference: In the extended kernel, *s* has a reference to *t*. The metaclasses *s* and *t* belong to the kernel or to the extension.

Expand: all the attributes of *s* are copied into *t*. Attributes with the same name are merged. The *t* metaclass must belong to the kernel metamodel.

Match: *s* and *t* represent the same concept; they are merged into a single metaclass which contains the union of all the structural features (i.e., both attributes and references) of *s* and *t*. Their supertype and subtype references are merged as well. The *t* metaclass must belong to the kernel metamodel.

The proposed extension operators have the following properties that underpin the construction of the type hierarchy previously presented.

Property 1 (Monotonicity - kernel preservation). Each operator can only add elements to the kernel being extended. The deletion of kernel elements is forbidden.

Property 2 (Extension integrity). All the elements of the extension metamodel are added to the kernel metamodel according to the operator semantics. In other words, it is not possible to use only a fragment of an extension. This is ensured by the default behavior of the extension mechanism which copies all the extension elements that are not considered by the used operators.

Property 3 (Parallel independence). An operator can be applied only if conflicts[1] among the structural features of the involved metaclasses do not occur. For instance, it is not possible to match a kernel metaclass A containing an attribute $p : Int$ with an extension metaclass B containing an attribute $p : String$ because of the conflicting types of the attribute p.

[1] According to the classification in [14], the conflicts that are considered in the *parallel independence* property are the so-called syntactic conflicts.

By referring to Figure 3, Properties 1, 2, and 3 ensure the confluence of the extension mechanism (see Theorem 1).

Theorem 1 (Confluence). *Given two weaving models* wm_{A_kx} *and* wm_{A_ky} *between the kernel* A_k *and the extensions* x *and* y*, respectively, and* $wm_{A_kx} \cup wm_{A_ky}$ *does not contain weaving links that refer to elements in* x *and* y *which are in conflict, then:*

$$tr(wm_{A_kxy},A_k,xy)=tr(wm_{A_kx},tr(wm_{A_ky},A_k,y),x)=tr(wm_{A_ky},tr(wm_{A_kx},A_k,x),y)$$

where wm_{A_kxy} *is the weaving between the kernel* A_k *and the extension* xy *is given as the union of* wm_{A_kx} *and* wm_{A_ky}*.*

The proof of the theorem is given in Appendix.

3.2 Identification of transformation paths

ALs can be bound to different kernels of the built hierarchy. To better explain both the problematics of the transformation path identification and the provided solution, we use the example presented before.

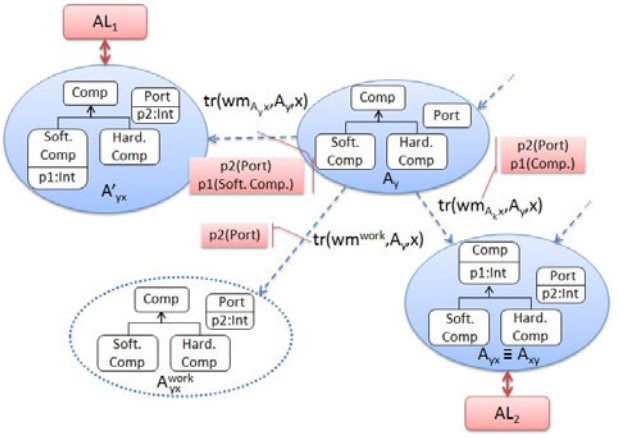


Fig. 4. AL-to-AL transformation management

Figure 4 describes two generic ALs, AL_1 and AL_2, bound to A'_{yx} and A_{yx}, respectively. As described before, A'_{yx} is an extension of A_y that contains the performance annotation $p1$ added to `SoftComp` and the performance annotation $p2$ added to `Port`. Whereas, A_{yx} contains the performance annotation $p1$ added to `Component` and the performance annotation $p2$ added to `Port`. In this simple example the performance annotation $p2$ is present both in A'_{yx} and in A_{yx}; therefore, when transforming from a model specified with AL_1 to a model conforming to AL_2, it is desirable to maintain also the $p2$ annotation. In a transformation realized by passing through A_y we lose such an information. For this reason our approach automatically builds a working kernel, A^{work}_{yx} in Figure 4, which contains also the $p2$ annotation. This working kernel contains the metaclass `Port` with the annotation $p2$, while $p1$ is ignored since in A'_{yx} $p1$ is attached to `SoftComp` and in A_{yx} it is attached to `Comp`. Thus, $p1$ represents information that cannot be automatically translated. Notice that once transforming from AL_1 to AL_2 and back, the values of the $p1$ annotations possibly attached to `SoftComp` instances of AL_1 are preserved by means of the *lost-in-translation* mechanism described in [7].

Formally, let A_l and A_m be the kernels which AL_1 and AL_2 are bound to, respectively. Moreover, let T_{A_l} and T_{A_m} the types of A_l and A_m, respectively. To identify the transformation path between A_l and A_m that minimizes the loss of information, we look for the most "specialized" common ancestor A_{anc} of A_l and A_m such that:

$$((T_{A_{anc}} < T_{A_l}) \wedge (T_{A_{anc}} < T_{A_m})) \wedge (\nexists A' \in \mathbb{A} | (T_{A'} < T_{A_l}) \wedge (T_{A'} < T_{A_m}) \wedge (T_{A_{anc}} < T_{A'}))$$

To understand if we can build a kernel useful to reduce the loss of information, we consider the extensions that have been applied from A_{anc} to A_l and from A_{anc} to A_m. The functions in Def. 4 and Def. 5 are introduced to construct such a kernel.

Definition 4 (extensionApplications). *$extensionApplications$: $\mathbb{A} \times \mathbb{A} \rightarrow 2^{\mathbb{W}}$ is a function that given as input the kernels $A_i \in \mathbb{A}$ and $A_j \in \mathbb{A}$, such that $T_{A_j} < T_{A_i}$ (i.e., A_j is an ancestor of A_i) returns a set containing all the weaving models that have been applied to A_j for building the kernel A_i.*

Definition 5 (usedExtensions). *$usedExtensions$: $\mathbb{A} \times \mathbb{A} \rightarrow 2^{\mathbb{E}}$ is a function that given as input the kernels $A_i \in \mathbb{A}$ and $A_j \in \mathbb{A}$, such that $T_{A_j} < T_{A_i}$ (i.e., A_j is an ancestor of A_i) returns a set containing all the extensions that have been used to extend A_j for building the kernel A_i.*

The transformation path that minimizes the loss of information between A_l and A_m is calculated by means of the $pathIdentification$ algorithm shown in the left-hand side of Figure 5. In particular, $pathIdentification$ gets as input A_l and A_m and calculates the common ancestor A_{anc} (see line 1). Then the next step is to find a kernel that while transforming can reduce the loss of information. To this purpose the algorithm checks if there is an intersection between (i) the extensions that have been applied (i.e., weaving models) to A_{anc} to build A_l, and (ii) those that have been applied to A_{anc} to build A_m. The extension applications are calculated in two steps. Firstly, the sets of weaving models applied to A_{anc} for building the kernels A_l and A_m are calculated (lines 2 and 3, respectively). Secondly, for each set, the union of all the weaving models is calculated. More precisely wm_L and wm_M are the weaving models that have been

Algorithm 1 pathIdentification(A_l, A_m)

1: $A_{anc} \leftarrow calculateCommonAncestor(A_l, A_m)$
2: $L \leftarrow extensionApplications(A_l, A_{anc})$
3: $M \leftarrow extensionApplications(A_m, A_{anc})$
4: $wm_L \leftarrow \bigcup_{wm \in L} wm$
5: $wm_M \leftarrow \bigcup_{wm \in M} wm$
6: **if** $(wm_L \cap wm_M = \emptyset)$ **then**
7: **return** $calculatePath(A_l, A_{anc}, A_m)$
8: **else**
9: $wm_{lm}^{work} \leftarrow wm_L \cap wm_M$
10: $E \leftarrow createWorkingExtension(wm_{lm}^{work}, usedExtensions(A_l, A_{anc}), usedExtensions(A_m, A_{anc}))$
11: $A_{lm}^{work} \leftarrow tr(wm_{lm}^{work}, A_{anc}, E)$
12: **return** $calculatePath(A_l, A_{lm}^{work}, A_m)$
13: **end if**

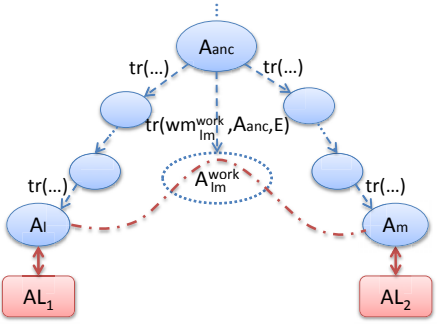


Fig. 5. Working kernel generation

obtained from the union of all the weaving models contained in L and M, respectively (lines 4 and 5). To understand if we can refine the hierarchy by building a new kernel that can reduce the loss of information, the intersection between wm_L and wm_M is calculated (line 6). If the intersection is empty, then all the information that is common to A_l and A_m is already contained into A_{anc}; consequently, the path that minimizes the loss of information between A_l and A_m starts from A_l, navigates the hierarchy up to A_{anc}, and then navigates the hierarchy down to A_m (see line 7).

If the intersection is not empty, then we have to refine the hierarchy as shown in Figure 5 in order to perform transformations (from AL_1 to AL_2 and vice versa) via a kernel more specific than A_{anc}. In other words, the idea is to extend A_{anc} with the information shared between A_l and A_m that is not contained in A_{anc}. The ad-hoc kernel is called A_{lm}^{work} and is automatically generated by using a working weaving model called wm_{lm}^{work}. This wm_{lm}^{work} is obtained from the intersection of wm_L and wm_M (line 9). As shown in the right-hand side of Figure 5, the weaving model wm_{lm}^{work} applies the working extension E to A_{anc} then generating A_{lm}^{work} (line 11). E is obtained by suitably merging the extensions that have been used to extend A_{anc} for building the kernel A_l and those that have been used to extend A_{anc} for building the kernel A_m (line 10). The merging of extensions is realized by means of the function $createWorkingExtension$ that considers only the portion of the extensions involved in at least one of the weaving links in wm_{lm}^{work}. $createWorkingExtension$ does not add new conflicts into A_{lm}^{work} since each weaving link added in A_{lm}^{work} belongs both to A_l and A_m; indeed having a conflict in A_{lm}^{work} would imply to have a conflict in both A_l and A_m. This is not possible since Property 3 of the extension operators ensures that A_l and A_m do not have conflicts (by construction).

It is worth noting that A_{lm}^{work} is a working kernel since it is exclusively used for transformation purposes and we do not allow ALs to be bound to A_{lm}^{work}. Finally, as shown in Figure 5, the path that minimizes the information loss between A_l and A_m starts from A_l, directly passes through A_{lm}^{work} and ends to A_m.

4 Case Study and Discussion

In Section 4.1 we present a case study to show how two real ALs can interoperate by means of the proposed approach. The scale of the considered case study does not allow us to show all technical aspects of the approach. Thus, we show the most automated parts, while more complex technicalities are better described by using small examples as done in Section 3. Then, Section 4.2 discusses issues related to the approach.

4.1 Putting the approach in practice

According to its business needs, an organization decided to draw and analyze the architecture of a system in the vehicular domain by using AADL [10] (with its behavioral annex), complemented with SaveCCM [15] (helpful to support the development of resource-efficient systems and to perform structural preventive analysis). The case study starts from an already existing kernel hierarchy (see the uppermost part of Figure 6) composed of three extensions of the root kernel A_0, namely *Behaviour*, *Embedded systems*, and *Real-time*. Due to space limitations, we do not describe the

concepts contained into the extension metamodels. We assume that two ALs are already bound to the hierarchy: Acme [16] is bound to A_0 and Darwin/FSP [17] to the *Behaviour* kernel. In order to apply the proposed approach, we need to identify the suitable kernel on which each AL can be profitably bound. Focusing on SaveCCM, it contains both real-time and embedded systems concepts. A satisfying kernel does not exist but two existing kernels, namely the *Embedded systems* and the *Real-time*, can be suitably used to obtain a new kernel on which SaveCCM can be bound. In this example the kernel can be produced by reusing both the existing weaving models wm_{A_0E} and wm_{A_0RT}. The obtained kernel, named *RT+E*, is shown in Figure 6. This kernel metamodel is automatically obtained, as explained in Sections 3.1. It is important to note that during this extension a new weaving model, wm_{A_0RT+E}, is automatically generated by composing wm_{A_0E} and wm_{A_0RT}. As explained in Section 3.1 this weaving model is extremely important to support further extensions of the kernel *RT+E*.

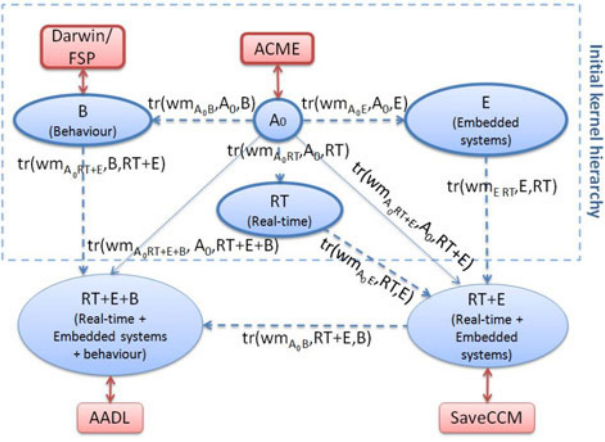


Fig. 6. SaveCCM and AADL into the hierarchy

Similarly to SaveCCM, AADL contains both real-time and embedded systems concepts; however, AADL contains also behavioral concepts since we are considering also its behavior annex. In this specific situation we look for a candidate kernel with real-time, embedded systems, and behavioral concepts. Building on the kernel *RT+E* and by considering also the extension B, we can build a new *RT+E+B* kernel by reusing both the wm_{A_0B} weaving model already used to extend A_0 with B and the

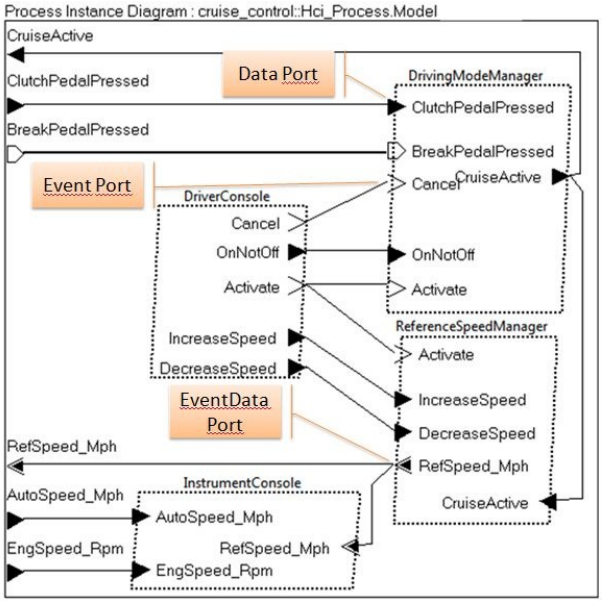


Fig. 7. AADL model of the HCI process

generated weaving model $wm_{A_0 RT+E}$. Once the extension metamodel *RT+E+B* has been generated, AADL can be bound to the hierarchy. *RT+E+B* contains real-time, embedded systems, and behavioral concepts. Finally, suitable model transformations are generated from each weaving model as described in Sections 3.2. Now that the kernel hierarchy is ready to be used, we can proceed by modeling the system of interest. It is a cruise control system, i.e., a system that automatically controls the speed of a vehicle according to the driver settings [18]. In this paper we focus on the Human Control Interface (HCI) subsystem, which is the front-end to the driver. Figure 7 shows the HCI process modeled in AADL. This process is composed of four threads managing the driving mode (`DrivingModeManager`), the reference speed (`ReferenceSpeedManager`), the buttons panel (`DriverConsole`), and a console (`InstrumentConsole`) for special settings of the system.

In order to transform the AADL model to the corresponding SaveCCM model, the transformation chain is calculated as described in Sections 3.2. In this case the calculated path passes through the kernel *RT+E* that is the most specific common ancestor of *RT+E* and *RT+E+B*. By means of this transformation chain we ensure that both real-time and embedded system concepts are accurately translated. Therefore, the information that is lost while transforming is limited to behavioral concepts or to concepts specific to AADL; they cannot be translated to SaveCCM even by using an ad-hoc transformation. However, without a systematically defined extension process SaveCCM and AADL could have been bound to two extensions of A_0 with potential but unexpressed similarities. This may lead to the loss of real-time and embedded system concepts.

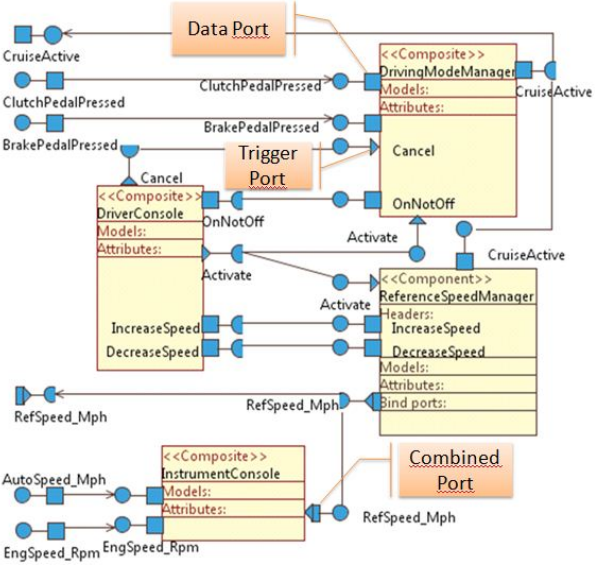


Fig. 8. SaveCCM model of the HCI component

Figure 8 shows the model of the HCI process automatically generated for SaveCCM. SaveCCM does not provide specific modeling constructs for processes and threads and then, as can be seen in the figure, both processes and threads become components; in particular the HCI process becomes a `Composite` component. This is because the generic component metaclass of AADL (which is a superclass of thread, process, memory, etc.) is linked to the component metaclass of the kernel *RT+E+B*, and the SaveCCM component metaclass is linked to the component metaclass of the kernel *RT+E*. We clearly have a loss of information when transforming from AADL to SaveCCM. However, the generated transformations are instrumented to maintain the information which

is lost so to recover it when transforming back from SaveCCM to AADL. `Data`, `Event`, and `EventData` ports are linked to the corresponding concepts in *RT+E+B*, which are linked in turn with `Data`, `Trigger`, and `Combined` ports of SaveCCM, respectively. Therefore, the semantics of the modeled ports is maintained when transforming from AADL to SaveCCM. This is obtained thanks to the kernel hierarchy. Without such a hierarchy, i.e., by passing directly through A_0, we loose the specific information related to ports since A_0 has only the concept of generic port.

4.2 Discussion

In this section we discuss the following aspects: (i) generalization of the approach, (ii) its scalability, and (iii) overhead added by the kernel hierarchy to the transformation.

Generalization: the overall approach is applied to the software architecture domain and specifically to ALs. However, the kernel hierarchy and transformation management can be easily applied to different domains by simply substituting A_0 with a different root kernel metamodel. The definition of the root kernel metamodel is strategic and requires particular attention. Please refer to the discussion section in [19] for more details about the process we followed for defining A_0. Finally, we believe that the proposed approach could be used as a new "profiling" mechanism able to support the extensibility mechanisms envisioned by Jacobson and Cook in the UML of the future [20].

Approach Scalability: according to Section 3, a kernel can be extended in several different ways depending on the specified weaving model. As described in Section 3.2, some "working" metamodels need to be added to the hierarchy in order to properly manage the transformations. Thus, from the scalability point of view it is important to understand the order of magnitude of the hierarchy. As reported in [7] the number of available architecture description languages is around 50 or 60. An estimation of the possible extensions is more difficult to be performed but based on the number of available ALs we are confident that this will not compromise the approach applicability.

Overhead: the kernel hierarchy adds some overhead to the transformations. In order to quantify this overhead it is important to understand the operations that need to be performed during the transformations and to identify the operations that are performed once forever. In Section 3.2 we explained the need of having a working metamodel and the procedure to build it. This metamodel and related weaving models are created once forever. Therefore, this cannot be considered as overhead of the transformation from one AL to another. The overhead that is added to each transformation from one AL to another is related to the fact that the transformation is actually a chain of transformations instead of a direct transformation from one AL to another. Assuming a constant time t for each transformation, the overhead can be quantified as $(t \times x) - t$, where x is the number of transformations composing the considered chain. In the case study presented in this work, we used an Intel Pentium D-3.2Ghz, with 4GB DDR-II of RAM, running

Windows 7 Professional. The generation of the transformation chain and its execution took less than four seconds with a source AADL model consisting of 603 modeling elements. The experience we had with the case study was encouraging from the point of view of the efficiency of the overall approach.

5 Related work

State-of-the-art approaches on ALs interoperability have been discussed in Section 2 outlining what is missing and then motivating the proposed approach. In this section we compare our work with existing work in the area of model-driven engineering.

Over the last years a number of work has been proposed to cope with the problem of tool integration and interoperability in MDE. Such works can be classified into *Transformation-based approaches* and *Metamodel integration approaches* [21]. The former approaches, like [22,6], propose the adoption of model transformations which aim to serve as a bridge between the various tools that have to interoperate. In particular, model transformations are used to transform data required by heterogeneous tools. Differently to our work, such approaches rely on manually written transformations defined with respect to the notations adopted by the considered tools. Metamodel integration approaches, like [23], rely on the definition of a common metamodel to establish tool interoperability. Even though such approaches are similar to our work, they do not provide mechanisms supporting the extension of the common metamodel.

The problem of interoperability has been tackled also in the context of model-to-model transformation languages. In [24] the authors propose an approach based on a *Common Intermediate Language* to support interoperability between different model transformation languages. Differently from our approach the authors analyze a set of well-known transformation languages and identify common characteristics which are captured in a common metamodel which is not extensible.

In [25] the authors propose an approach based on consistency rules, and bidirectional model transformations to automate the synchronizations of AUTOSAR (Automotive Open System ARchitecture)and SysML (System Modeling Language) modelsEven though the approach is general and can be applied on any couple of modeling languages, it differs from our work since the used model transformations which underpin the synchronization mechanism are manually written and are not organized in an extension hierarchy as proposed in this paper.

Going back to the nineties, a family of works have been proposed to exploit a single formal kernel language to integrate specifications written in different languages. One of the most prominent work in this family is the one by Jackson and Zave [26] in which Z is used as a common semantic domain for the composition of partial specifications defined in different languages. The resulting composed specification is then used to check the consistency of the initial partial specifications. Our goal is quite different since we consider the kernels hierarchy as an intermediate means for transforming models across different languages, rather then a way to check their global consistency.

6 Conclusion and Future Work

Approaches to support architectural interoperability typically choose to organize the different notations in a star topology with an intermediate central pivot. In a context in which the set of involved notations cannot be *a-priori* established, the pivot assumes the form of a small kernel. Since the transformations are always performed through the small kernel that can be very abstract, important information can be lost during the transformation. This calls for kernel extensions. This paper proposes a model-driven approach to (i) build the extensions and organize them in a hierarchy, (ii) realize the interoperability (through the hierarchy) by means of model transformations, and (iii) manage the overall hierarchy. The extension is performed through operators that have properties that ensure the extension confluence.

We realized a prototype automatizing the overall approach: it is a plugin for Eclipse that allowed us to perform experiments on some systems. As future work we plan to release the tool as an open source project and to experiment it on industrial case studies.

References

1. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. John Wiley & Sons (2009)
2. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE TSE 26(1) (2000)
3. Giese, H., Neumann, S., Niggemann, O., Schätz, B.: 2 Model-Based Integration. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) MBEERTS 2010. LNCS, vol. 6100, pp. 17–54. Springer, Heidelberg (2010)
4. Q-ImPrESS consortium, http://www.q-impress.eu (last access, September 2011)
5. ATESST2 consortium, http://www.atesst.org/ (last access, September 2011)
6. Sun, Y., Demirezen, Z., Jouault, F., Tairas, R., Gray, J.: A Model Engineering Approach to Tool Interoperability. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 178–187. Springer, Heidelberg (2009)
7. Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.A.: Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies. IEEE TSE 36(1) (2010)
8. Grassi, V., Mirandola, R., Sabetta, A.: Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. J. Syst. Softw. 80(4), 528–558 (2007)
9. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A comprehensive approach for the development of modular software architecture description languages. TOSEM 14(2) (2005)
10. Feiler, H.P., Lewis, B., Vestal, S.: The SAE Architecture Analysis and Design Language (AADL) Standard. In: RTAS Workshop on Model-driven Embedded Systems, pp. 1–10 (2003)
11. Bézivin, J.: On the Unification Power of Models. Software and Systems Modeling 4(2), 171–188 (2005)
12. Steel, J., Jézéquel, J.M.: On model typing. Software and System Modeling 6(4), 401–413 (2007)
13. Di Ruscio, D., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: Developing next generation ADLs through MDE techniques. ACM/IEEE ICSE 2010, 85–94 (2010)

14. Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering 28(5), 449–462 (2002)
15. Kerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M.: The SAVE approach to component-based development of vehicular systems. Jour. Syst. Softw. 80(5), 655–667 (2007)
16. Garlan, D., Monroe, R., Wile, D.: Acme: An Architecture Description Interchange Language. In: CASCON 1997, pp. 169–183 (1997)
17. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes 21(6) (1996)
18. Varona-Gomez, R., Villar, E.: Aads+: Aadl simulation including the behavioral annex. In: Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, pp. 379–384. IEEE Computer Society, Washington, DC (2010)
19. Eramo, R., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: A model-driven approach to automate the propagation of changes among Architecture Description Languages. In: Software and Systems Modeling, SoSyM (2010)
20. Jacobson, I., Cook, S.: The Road Ahead for UML (2010), http://www.drdobbs.com/architecture-and-design/224701702
21. Seifert, M., Wende, C., Assmann, U.: Anticipating unanticipated tool interoperability using role models. In: Proc. of MDI 2010, pp. 52–60. ACM (2010)
22. Ehrig, K., Taentzer, G., Varró, D.: Tool Integration by Model Transformations based on the Eclipse Modeling Framework. EASST Newsletter 12 (2006)
23. Baumgart, A.: A common meta-model for the interoperation of tools with heterogeneous data models. In: Proc. of MDTPI 2010 (2010)
24. Jouault, F., Kurtev, I.: On the interoperability of model-to-model transformation languages. Sci. Comput. Program. 68(3), 114–137 (2007)
25. Giese, H., Hildebrandt, S., Neumann, S.: Towards integrating sysml and autosar modeling via bidirectional model synchronization. In: MBEES, pp. 155–164 (2009)
26. Zave, P., Jackson, M.: Conjunction as composition. ACM Trans. Softw. Eng. Methodol. 2, 379–411 (1993)

Appendix: Proof of Theorem 1

Let us assume (ad absurdum) that:

- $tr(wm_{A_k xy}, A_k, xy) = A'$,
- $tr(wm_{A_k x}, tr(wm_{A_k y}, A_k, y), x) = A''$, and
- $A' \neq A''$

(the symmetric, i.e., $tr(wm_{A_k xy}, A_k, xy) = A'$, $tr(wm_{A_k y}, tr(wm_{A_k x}, A_k, x), y) = A''$, and $A' \neq A''$ will directly follow). This can happen in four cases:
1. a metaclass C exists in A' and does not in A''. This means that C exists in A_k, in x, or in y. In case C exists in A_k, this implies that the application of $wm_{A_k x}$ or $wm_{A_k y}$ deletes it. This is absurd for Property 1. In case C exists in x or in y, this implies that $wm_{A_k x}$ or $wm_{A_k y}$ do not add it during the extension. This is absurd for Property 2.

2. a metaclass C exists in A'' and does not in A'. In case C exists in A_k, this implies that wm' deletes it. This is absurd since the operators that we use in wm' have to respect Property 1. In case C exists in xy, this implies that wm' does not add it during the

extension. This is absurd since wm' is basically the union of wm_{A_kx} and wm_{A_ky} and then it respects Property 2.

3. a metaclass C exists both in A' and A'' and these two versions differ on some structural features, i.e., attributes and references. This can be caused exclusively due to deletion or conflicting additions performed by either wm_{A_kx} and wm_{A_ky} or wm'. This is absurd since Property 1 forbids the deletion and Property 3 prevents conflicts.

4. a metaclass C exists both in A' and A'' and these two versions differ on some parent. This can be caused by different applications of the *inherit* operator. This leads to an absurd since: i) a weaving model cannot delete a class parent for Property 1, ii) the sequential application of wm_{A_kx} and wm_{A_ky} cannot add class parents in a different way from wm' (wm' is the union of wm_{A_kx} and wm_{A_ky} and its existence ensures that Property 3 is satisfied).

Moving from Specifications to Contracts in Component-Based Design*

Sebastian S. Bauer[1], Alexandre David[2], Rolf Hennicker[1],
Kim Guldstrand Larsen[2], Axel Legay[2,3],
Ulrik Nyman[2], and Andrzej Wąsowski[4]

[1] Ludwig-Maximilians-Universität München, Germany
[2] Computer Science Department, Aalborg University, Denmark
[3] INRIA/IRISA, Rennes Cedex, France
[4] IT University of Copenhagen, Denmark

Abstract. We study the relation between specifications of component behaviors and contracts providing means to specify assumptions on environments as well as component guarantees. We show how a contract framework can be built in a generic way on top of any specification theory which supports composition and specification refinement. Our contract framework lifts refinement to the level of contracts and proposes a notion of contract composition on the basis of dominating contracts. Contract composition satisfies a universal property and can be constructively defined if the underlying specification theory is complete, i.e. it offers operators for quotienting and conjoining specifications. We illustrate our generic construction of contracts by moving a specification theory for modal transition systems to contracts and we show that a (previously proposed) trace-based contract theory is an instance of our framework.

1 Introduction

Over the years we have seen a remarkable growth of complexity and size of software systems. This growth has been possible due to rapid development in hardware and software technology. Development of software today uses strong abstraction and encapsulation principles, that allows componentizing systems into comprehensible units.

This rapid growth of size and complexity of systems has inspired intensive research into component-oriented design and analysis methods for software. In the domain of safety critical concurrent software a number of interface theories have been proposed to this end, starting with the seminal work of Alfaro and Henzinger [2] devoted to tracking communication errors in discrete systems, followed by numerous extensions addressing other errors, or other forms of abstraction [1,16,17]. These include abstract specification of discrete finite-state systems exploiting may/must modalities [18,20,21,23,26,33,34,36], specification

* Work partially supported by MT-LAB (a VKR Centre of Excellence), by an "Action de Recherche Collaborative" ARC (TP)I, and by the EU project ASCENS, 257414.

J, de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 43–58, 2012.

of systems manipulating complex data [4,8,35], specification of real-time embedded systems and real-time communication protocols [3,11,14,24], specification of randomized and probabilistic systems [12], and modeling of resource usage [6,13]. This proliferation of results is both positive and negative. Positive since it is a sign of fast progress in the field. Negative, because many works appear similar, yet it is difficult to compare them.

We attempt to develop a synthesis of the existing work in a uniform common framework. Altogether these theories have led to a shared understanding of what are the main ingredients of a mature specification theory for behavioral components; namely notions of satisfaction and refinement, together with composition operators such as conjunction, parallel compositions, and quotients. Nevertheless, despite this agreement, and despite the algebraic similarity of many specification theories, no uniform meta-theory exists that would formalize the abstract structure to enable better comparability of work, and reuse of results, channeling proliferation into higher quality and impact.

Independently, a number of contract theories, based on assume-guarantee (AG) reasoning have been developed, with a similar aim of approaching the compositional design. Contract theories differ from specification theories in that they strictly follow the principle of separation of concerns. They separate the specification of assumptions from specification of guarantees, a choice largely inspired by early ideas on manual proof methods of Misra, Chandy [30] and Jones [22], along with the wide acceptance to pre-/post-condition style of specification in programming [29]. Contract theories exist for discrete systems [10,25,31] and probabilistic systems [15,37].

Even though the specification theory, and the contract theory research have similar objectives, it is not clear so far what the two approaches offer with respect to each other, and whether their development is making the others complementary, or superfluous. So our second goal is to understand not only the essential structure of specification theories, but also their relation to contract theories. All in all we set off to organize (somewhat) the field of compositional specification for behavioral components.

We define contracts as pairs, (A, G), where A is a specification of assumptions, and G is a specification of guarantees. This leads us to our hypothesis that most specification theories should have enough structure to be used as a basis of an associated contract theory with explicit assumptions and guarantees. Dually, we observe that contract theories tend to degenerate to specification theories in the following simple manner: a specification G is describing the same models as a contract (tt, G) — so a contract without any assumption. Thus any reasonably complete contract theory can be used as a specification theory.

We make this intuition formal by developing a meta-theory of specifications and contracts. First, in Sect. 2, we propose a simple and general axiomatization of specification theories, able to capture the algebraic structure of most of the current specification theories (some frameworks require small adaptation, because not all of them have been originally developed with a complete set of operators in mind). Second, we demonstrate in Sect. 3 how a contract framework

can be derived from a specification theory, using our abstract constructions. As a result we are able to instantiate "for free" a contract theory with good properties of contracts from any specification theory fulfilling our axioms.

Any such derived contract theory is automatically equipped with:

- An implementation and an environment semantics reflecting the set of interfaces and environments that satisfy the guarantees and assumptions of the contract, respectively.
- A refinement relation that allows to compare contracts in terms of sets of implementations and legal environments.
- A structural composition, which encapsulates contracts for two communicating components into one contract for the composition of the two.

These results follow automatically as soon as the specification theory is equipped with parallel composition, conjunction, and a quotient of parallel composition. A number of specification theories have been proposed recently that satisfy our assumptions. In the course of this paper, we illustrate our general constructions by moving two specification theories to two contract theories: a simple trace-based specification theory, in which specifications are represented as sets of runs or traces (inspired by Benveniste et al. [10]), and as a more detailed example, we use modal specifications [32] in Sect. 4 to derive so-called modal contracts. All proofs can be found in [5].

We would like to stress that there are many other specification theories that fit into our framework, for instance, timed specifications [11], which allow us to derive "for free" a contract theory for timed systems, which has not yet been proposed in the literature.

2 Specification Theories

In our study the abstract concept of a specification theory defines rudimentary properties that should be satisfied by any formal framework for component behavior specifications. Given a class $\mathscr{S}$ of specifications, a specification theory includes a composition operator $\otimes$ to combine specifications to larger ones.[1] Additionally, a specification theory must offer a refinement relation $\leq$ to relate "concrete" and "abstract" specifications, i.e. $S \leq T$ means that S refines T. To obtain a specification theory, refinement must be compositional in the sense that it must be preserved by the composition operator.

Formally, a *specification theory* is a triple $(\mathscr{S}, \otimes, \leq)$ consisting of a class $\mathscr{S}$ of specifications, a parallel composition operator $\otimes : \mathscr{S} \times \mathscr{S} \to \mathscr{S}$ and a reflexive and transitive refinement relation $\leq \subseteq \mathscr{S} \times \mathscr{S}$, such that for all $S, S', T, T' \in \mathscr{S}$,

$$\text{whenever } S' \leq S \text{ and } T' \leq T, \text{ then } S' \otimes T' \leq S \otimes T. \tag{A1}$$

[1] The composition operator is, in general, partial since it is not always syntactically meaningful to compose specifications, due to syntactic constraints. In this work, however, to avoid a lot of technicalities, we will restrict ourselves to total composition operators – though the theory is easily extendable to partial composition operators.

The refinement relation induces an equivalence relation $=$ on specifications, by $S = T$ if and only if $S \leq T$ and $T \leq S$. The composition operator is commutative and associative with respect to this equivalence relation.

Obviously, in a top-down design, the requirements for a specification theory support independent development of components. To a certain extent a specification theory supports also bottom-up design, where existing components can be reused as parts of a larger system architecture, as long as local refinements are correct and local specifications fit into the context.

Specification theories sometimes come along with an operator $/$, called *quotient*, which is dual to parallel composition and which allows to synthesize specifications: When given a requirement specification T of the overall system and a smaller specification S, then the quotient T/S is the most general specification such that $S \otimes (T/S) \leq T$. Formally, quotient is a partial operator $/ : \mathscr{S} \times \mathscr{S} \hookrightarrow \mathscr{S}$ that satisfies

$$T/S \text{ defined if and only if } \exists X \in \mathscr{S} : S \otimes X \leq T. \tag{A2}$$

$$\text{If } T/S \text{ defined, then } S \otimes (T/S) \leq T. \tag{A3}$$

$$\text{If } T/S \text{ defined, then } \forall X \in \mathscr{S} : S \otimes X \leq T \implies X \leq T/S. \tag{A4}$$

When two separate teams independently develop specifications that are intended to be realized by the same component, then it is useful to have a *conjunction* operator $\wedge$ that computes the most general specification that realizes both specifications (if this is possible). Formally, conjunction is a partial operator $\wedge : \mathscr{S} \times \mathscr{S} \hookrightarrow \mathscr{S}$ such that

$$S \wedge T \text{ defined if and only if } \exists X \in \mathscr{S} : X \leq S \text{ and } X \leq T. \tag{A5}$$

$$\text{If } S \wedge T \text{ defined, then } S \wedge T \leq S \text{ and } S \wedge T \leq T. \tag{A6}$$

$$\text{If } S \wedge T \text{ defined, then } \forall X \in \mathscr{S} : X \leq S \text{ and } X \leq T \implies X \leq S \wedge T. \tag{A7}$$

When a specification theory supports quotient as well as conjunction, then we call it a *complete* specification theory.

Example 1. *As our running example we revisit the contract framework of Benveniste et al. [10], for two reasons: first, it uses a simple trace-based language to represent behavior of components, and specification operators boil down to simple set operations which we believe helps to understand the abstract requirements of specification theories; second, we will show that in fact our general constructions applied to this trace-based specification theory exactly results in the contract framework (in a simplified version) described in [10].*

In this simple theory, a global set $\mathbb{P}$ of ports is assumed over which components can communicate by reading and writing port values. The class of specifications consists of all (possibly empty) subsets of $\mathcal{R}(\mathbb{P})$ which is the set of all runs over $\mathbb{P}$ where each run assigns a history of values to the ports in $\mathbb{P}$. For example, a run could be a function $\rho : \mathbb{R}_{\geq 0} \to (\mathbb{P} \to \mathbb{V})$ from the time domain $\mathbb{R}_{\geq 0}$ to a valuation $\mathbb{P} \to \mathbb{V}$ of the ports, for some value set $\mathbb{V}$.

In this setting, refinement is simply defined by set inclusion, composition and conjunction is intersection (they are the same since we are dealing with a single

global signature). Note that conjunction is total, as the empty set is also a specification. For any two specifications T and S, the dual operation to composition, quotient, is defined by $T/S =_{def} T \cup \neg S$, where $\neg A =_{def} \mathcal{R}(\mathbb{P}) \setminus A$. Notice that indeed quotient is the maximal specification X such that S composed with X refines T, i.e. $S \cap X \subseteq T$.

In the following we will see that if we apply the general constructions of our contract framework to the trace-based case we will obtain the contract framework of Benveniste et al. [10].

3 Building a Contract Framework

For the development of our abstract contract framework, we assume to be given a specification theory $(\mathscr{S}, \otimes, \leq)$ as defined in the previous section.

3.1 Contracts and Their Semantics

On top of the specification theory we define a notion of a contract which explicitly distinguishes between assumption and guarantee specifications.

Definition 1. *A* contract *is a pair (A, G) where $A, G \in \mathscr{S}$ are two specifications.*

In a contract (A, G), the specification A expresses the assumption on the environment of the component, whereas the specification G describes the guarantee of any component implementation to the environment given that the environment respects the assumption A. For the definition of implementation correctness, we use a notion of *relativized refinement* which is derived from the refinement relation of the underlying specification theory.

Definition 2. *Relativized refinement is the ternary relation in $\mathscr{S} \times \mathscr{S} \times \mathscr{S}$ defined as follows: for all $S, E, T \in \mathscr{S}$,*

$$S \leq_E T \quad \text{if and only if} \quad \forall E' \in \mathscr{S} : E' \leq E \implies S \otimes E' \leq T \otimes E'.$$

$S \leq_E T$ intuitively means that S refines T if both are put in any context E' that refines E; in particular, $S \otimes E \leq T \otimes E$. The following lemma summarizes properties of relativized refinement that are easy consequences of the definition.

Lemma 1. *Relativized refinement is a preorder, and for all $S, E, E', T \in \mathscr{S}$, whenever $S \leq_E T$ and $E' \leq E$ then $S \leq_{E'} T$.*

The *implementation semantics* of a contract (A, G) is given by the set of all specifications that satisfy the contract guarantee G *under the assumption* A:

$$[\![C]\!]_{\text{impl}} = \{I \in \mathscr{S} \mid I \leq_A G\}.$$

This is a significant generalization of pure specification theories where it is usually assumed that implementations must literally satisfy the specification. The

environment semantics of the contract (A, G) consists of all (environment) specifications for (or users of) the component satisfying the assumption A of the contract:

$$[\![C]\!]_{\text{env}} = \{E \in \mathscr{S} \mid E \leq A\}.$$

In summary, the semantics of a contract is given by both implementation semantics and environment semantics. Two contracts are *semantically equivalent*, if they have the same (implementation and environment) semantics.

Example 2. *In our trace-based example the relativized refinement $S \leq_E T$ can be easily shown to be equivalent to $S \cap E \subseteq T$; note that all specifications describe sets of runs over the same global set of ports $\mathbb{P}$.*

Our first result is a direct consequence of the definition of a contract and contract semantics: Whenever one has a correct environment and a correct implementation of a contract, then their composition is a refinement of the composition of assumption and guarantee of the contract.

Theorem 1. *Let $C = (A, G)$ be a contract. For all $E, I \in \mathscr{S}$, if $E \in [\![C]\!]_{\text{env}}$ and $I \in [\![C]\!]_{\text{impl}}$ then $E \otimes I \leq A \otimes G$.*

The implementation semantics of a contract in general depends on both the assumption A and the guarantee G. However, if the implementation semantics of (A, G) is independent of the assumption A, we say that the contract (A, G) is in normal form.

Definition 3. *A contract $C = (A, G)$ is* in normal form *if for all specifications $I \in \mathscr{S}$, $I \leq_A G$ if and only if $I \leq G$.*

It may be the case that a contract (A, G) can by transformed into a semantically equivalent contract (A, G^{nf}) in normal form by weakening of G to G^{nf}. In the examples considered here the underlying specification theories are powerful enough to allow such a weakening for any contract (A, G).

Example 3. *For a contract (A, G) in our trace-based example, a semantically equivalent contract in normal form (see [10]) is given by $(A, G \cup \neg A)$. It is indeed in normal form according to our definition since for any specification I, $I \cap A \subseteq G$ if and only if $I \subseteq G \cup \neg A$.*

3.2 Refinement of Contracts

Next, we turn to the question how contracts can be refined. We follow here a standard approach inspired by notions of behavioral subtyping [28] that a contract C' refines another contract C if C' admits less implementations than C, but more legal environments than C.

Definition 4. *Let C and C' be two contracts. The contract C' refines the contract C (is stronger than C), written $C' \preceq C$, if $[\![C']\!]_{\text{impl}} \subseteq [\![C]\!]_{\text{impl}}$ and $[\![C']\!]_{\text{env}} \supseteq [\![C]\!]_{\text{env}}$.*

The refinement relation between contracts is reflexive and transitive. Obviously, two contracts C, C' are semantically equivalent if and only if $C' \preceq C$ and $C \preceq C'$. The following theorem characterizes contract refinement by contra-/covariant (relativized) refinement of corresponding assumptions and guarantees.

Theorem 2. *Let (A, G) and (A', G') be two contracts. Then $(A', G') \preceq (A, G)$ if and only if $A \leq A'$ and $G' \leq_A G$.*

An immediate consequence is that whenever two contracts (A, G), (A', G') are in normal form, then $(A', G') \preceq (A, G)$ if and only if $A \leq A'$ and $G' \leq G$.

Example 4. *Refinement of contracts (A, G) by (A', G') is called dominance in [10] (not to be mixed up with our notion of dominance later on), and is defined by $A \subseteq A'$ and $G' \subseteq G$ which matches our definition of contract refinement if contracts are in normal form. For the other cases we have achieved a more thorough (weaker) definition of refinement which we would suggest to use for the trace-based approach as well.*

3.3 Composition of Contracts

When implementations I_1 and I_2 of individual components are composed, their composition is only semantically meaningful if the contracts, say C_1, C_2, of the single components fit together. This mean that there exists a 'larger' contract C which subsumes C_1 and C_2 such that (1) the composition $I_1 \otimes I_2$ is a correct implementation of C, and (2) each correct environment of C controls the single implementations in such a way that they mutually satisfy the assumptions of the single contracts. Inspired by [31] we call such a contract C a dominating contract for C_1 and C_2.

Definition 5. *Let C, C_1, and C_2 be contracts. C* dominates *C_1 and C_2 if the following two conditions are satisfied:*

1. *Any composition of correct implementations of C_1 and C_2 results in a correct implementation of the contract C:*
   - $\forall I_1 \in [\![C_1]\!]_{\text{impl}} : \forall I_2 \in [\![C_2]\!]_{\text{impl}} : I_1 \otimes I_2 \in [\![C]\!]_{\text{impl}}$
2. *For any correct environment of the contract C_1, the composition with a correct implementation of the C_1 (C_2) results in a correct environment of C_2 (C_1). Formally, for all $E \in [\![C]\!]_{\text{env}}$,*
   - $\forall I_1 \in [\![C_1]\!]_{\text{impl}} : E \otimes I_1 \in [\![C_2]\!]_{\text{env}}$,
   - $\forall I_2 \in [\![C_2]\!]_{\text{impl}} : E \otimes I_2 \in [\![C_1]\!]_{\text{env}}$.

We say that two contracts C_1, C_2 are dominatible *if there exists a contract C dominating C_1, C_2.*

A composition of two contracts C_1 and C_2 is a strongest dominating contract for C_1 and C_2.

Definition 6. *A contract C is called* contract composition *of the contracts C_1 and C_2 if*

1. *C dominates C_1 and C_2,*
2. *for all contracts C', if C' dominates C_1 and C_2 then $C \preceq C'$.*

Contract compositions, if they exist, are unique up to semantic equivalence of contracts. We will now turn to the questions (1) whether two contracts are dominatible and (2) whether the composition of two contracts exists and, if so, whether it can be *constructively* defined. For this purpose we generally assume in the following that any contract has a normal form, i.e. for any $C = (A, G)$ there exists a semantically equivalent contract $C^{nf} = (A^{nf}, G^{nf})$ which is in normal form. Due to the definition of environment semantics, without loss of generality, we can always assume in the following that $A^{nf} = A$.

We consider first question (1), for which the following lemma is useful. It follows directly from the definition of a dominating contract.

Lemma 2. *Two contracts C_1 and C_2 are dominatible if and only if their normal forms C_1^{nf} and C_2^{nf} are dominatible.*

The next theorem provides a characterization of dominatability. The idea is that there must be an environment under which implementations of the single contracts can be adapted to meet each others assumptions.

Theorem 3. *Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be two contracts with normal forms $C_1^{nf} = (A_1, G_1^{nf})$ and $C_2^{nf} = (A_2, G_2^{nf})$ respectively. C_1 and C_2 are dominatible if and only if $\exists E \in \mathscr{S} : G_1^{nf} \otimes E \leq A_2$ and $G_2^{nf} \otimes E \leq A_1$.*

We now turn to question (2) from above. For this we assume from now on a complete specification theory (recall that such a theory has quotient and conjunction) over which contracts are constructed.

Definition 7. *Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be two contracts with normal forms $C_1^{nf} = (A_1, G_1^{nf})$ and $C_2^{nf} = (A_2, G_2^{nf})$ respectively. $C_1 \boxtimes C_2$ is defined if and only if C_1 and C_2 are dominatible and then*

$$C_1 \boxtimes C_2 =_{def} ((A_1/G_2^{nf}) \wedge (A_2/G_1^{nf}), G_1^{nf} \otimes G_2^{nf}).$$

Note that, by Lemma 2, $C_1 \boxtimes C_2$ is semantically equivalent to $C_1^{nf} \boxtimes C_2^{nf}$. The next lemma shows that $C_1 \boxtimes C_2$ is indeed well-defined.

Lemma 3. *Let C_1 and C_2 be two contracts with normal forms as in Def. 7. $(A_1/G_2^{nf}) \wedge (A_2/G_1^{nf})$ is defined if and only if $\exists E \in \mathscr{S} : G_1^{nf} \otimes E \leq A_2$ and $G_2^{nf} \otimes E \leq A_1$, if and only if C_1 and C_2 are dominatible.*

The next theorem answers question (2) from above.

Theorem 4. *If the contracts C_1 and C_2 are dominatible, then $C_1 \boxtimes C_2$ is (up to semantic equivalence) the composition of C_1 and C_2.*

The next statements deal with the relationship between contract composition and contract refinement. First, dominance is preserved under refinement of individual contracts.

Theorem 5. *Let C_1, C_1', C_2, C_2', C be contracts such that $C_1' \preceq C_1$ and $C_2' \preceq C_2$. If C dominates C_1 and C_2, then C dominates also C_1' and C_2'.*

Second, contract refinement is preserved under contract composition, thus our contract framework satisfies itself the requirements of a specification theory of Sect. 2 if we admit *partial* composition (which has been disregarded in Sect. 2 just for reasons of simplicity).

Theorem 6. *Let C_1, C_2, D_1, D_2 be contracts such that C_1 and C_2 are dominatible. If $D_1 \preceq C_1$ and $D_2 \preceq C_2$ then $D_1^{nf} \boxtimes D_2^{nf} \preceq C_1^{nf} \boxtimes C_2^{nf}$.*

Example 5. *In [10], contract composition is defined by*

$$(A_1, G_1) \boxtimes (A_2, G_2) = ((A_1 \cap A_2) \cup \neg(G_1 \cap G_2), G_1 \cap G_2).$$

Note that the assumption can be reformulated to $(A_1 \cup \neg G_1 \cup \neg G_2) \cap (A_2 \cup \neg G_1 \cup \neg G_2)$, and since the contracts (A_1, G_1) and (A_2, G_2) are in normal form we have $A_1 \cup \neg G_1 = A_1$ and $A_2 \cup \neg G_2 = A_2$. Hence we get $(A_1 \cup \neg G_2) \cap (A_2 \cup \neg G_1)$ as assumption which, all in all, exactly matches our definition of $\boxtimes$ for contracts.

4 Modal Contracts

To illustrate our general constructions for moving from a specification theory to contracts, we consider a well-established specification theory based on modal transition systems that has gained considerable interest in recent years, as it nicely supports loose specifications together with stepwise refinement. Modal transition systems [27] are labeled transition systems with two types of transition relations: *may* transitions model optional (allowed) behavior that need not be implemented in a refinement, and *must* transitions model required behavior. In [32] a complete specification theory for modal specifications (which correspond to deterministic modal transition systems) has been defined, which allows us to get *modal contracts* for free. Modal contracts have been defined already in [19,31] and we will comment on the differences in the next section.

We briefly sketch the specification theory for modal specifications, for a thorough introduction see [32]. A modal specification (MS) is formally defined as a tuple $S = (St, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ where St is the set of states, $s_0 \in St$ is the initial state, Σ is the set of actions, and $\dashrightarrow, \longrightarrow \subseteq St \times \Sigma \times St$ are the may and must transition relation, respectively, such that $\longrightarrow \subseteq \dashrightarrow$. Any MS is required to be deterministic: for all states $s, s', s'' \in St$ and all actions $\alpha \in \Sigma$, if $(s, \alpha, s'), (s, \alpha, s'') \in \dashrightarrow$ then $s' = s''$. In the following, we usually write $s \overset{\alpha}{\dashrightarrow} s'$ for $(s, \alpha, s') \in \dashrightarrow$, and similarly for must transitions.

We consider a simple component-based system consisting of two components: component *Server* with contract (A_{Server}, G_{Server}) over the action set $\Sigma_{Server} = \{msg, secret\_msg, auth, send\}$ (i.e. both A_{Server}, G_{Server} have the set of actions Σ_{Server}), and a component *User* with contract (A_{User}, G_{User}) over set of actions $\Sigma_{User} = \{auth, send\}$. The two contracts can be seen in Fig. 1(a)–(d). May transitions are drawn with dashed arrows, and must transitions with solid arrows. May transitions underlying must transitions are not drawn for simplicity.

The contract (A_{Server}, G_{Server}) intuitively expresses the following protocol: First, the environment can issue a message (msg) that is then sent by the server

to the user (*send*). Second, the environment can also issue a secret message (*secret_msg*), that is only sent to the user if the server receives an authentication code from the user (*auth*). More precisely, the assumption A_{Server} formulates the following requirements on the environment:

- The authentication code may always be received.
- New messages (secret or not) are only allowed to be sent in the initial state.
- Once a message is received, the environment must be ready to accept the sending of the server.
- Once a secret message is received, the authentication code must be received.

The contract (A_{User}, G_{User}) for the user component is simpler: The guarantee is that the messages can always be received from the server, however, the sending of the authentication code may not be possible. The assumption A_{User} always allows the actions *auth* and *send*, without any specific order.

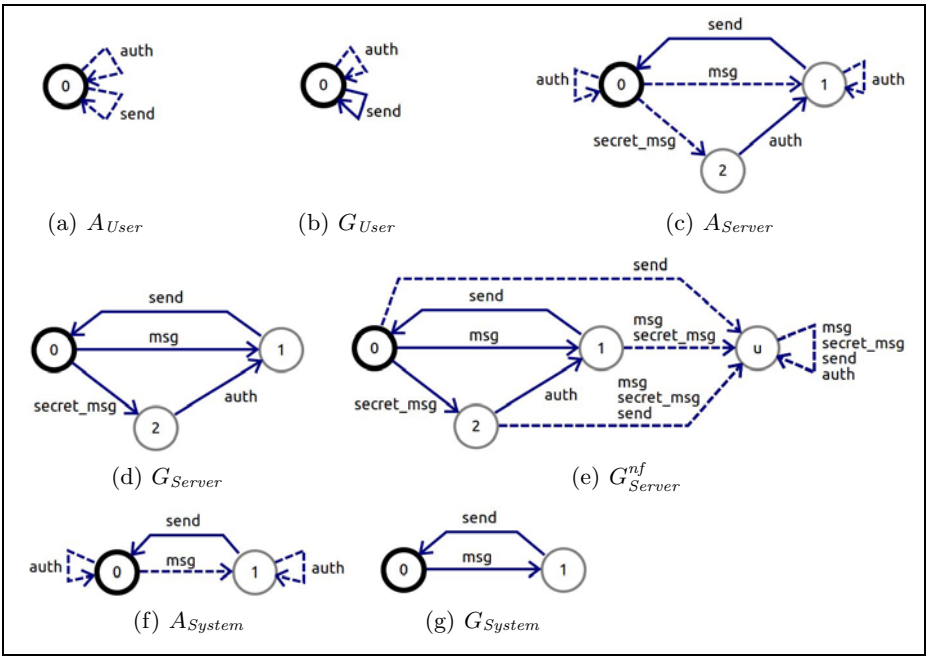


Fig. 1. Modal contracts for a simple message system

Before we discuss how these two modal contracts are composed, we first have to discuss the underlying specification theory, so refinement together with all the specification operators for MS. Refinement of MS is defined as follows: an MS S refines another MS T, written $S \leq_m T$, if they have the same set of actions Σ and if there exists a relation $R \subseteq St_S \times St_T$ such that $(s_0, t_0) \in R$ and for

all $(s,t) \in R$ and all $\alpha \in \Sigma$, whenever $s \overset{\alpha}{\dashrightarrow} s'$ then there exists $t \overset{\alpha}{\dashrightarrow} t'$ and $(s',t') \in R$, and whenever $t \overset{\alpha}{\longrightarrow} t'$ then there exists $s \overset{\alpha}{\longrightarrow} s'$ and $(s',t') \in R$. For instance, in Fig. 1, G_{User} is a refinement of A_{User}, i.e. $G_{User} \leq_m A_{User}$.

$\otimes$	$s_2 \overset{\alpha}{\longrightarrow} s_2'$	$s_2 \overset{\alpha}{\dashrightarrow} s_2'$
$s_1 \overset{\alpha}{\longrightarrow} s_1'$	$(s_1,s_2) \overset{\alpha}{\longrightarrow} (s_1',s_2')$	$(s_1,s_2) \overset{\alpha}{\dashrightarrow} (s_1',s_2')$
$s_1 \overset{\alpha}{\dashrightarrow} s_1'$	$(s_1,s_2) \overset{\alpha}{\dashrightarrow} (s_1',s_2')$	$(s_1,s_2) \overset{\alpha}{\dashrightarrow} (s_1',s_2')$

$/$	$s_2 \overset{\alpha}{\longrightarrow} s_2'$	$s_2 \overset{\alpha}{\dashrightarrow} s_2',\ s_2 \not\overset{\alpha}{\longrightarrow}$	$s_2 \not\overset{\alpha}{\dashrightarrow}$
$s_1 \overset{\alpha}{\longrightarrow} s_1'$	$(s_1,s_2) \overset{\alpha}{\longrightarrow} (s_1',s_2')$	$(s_1,s_2) \in \lightning$	$(s_1,s_2) \in \lightning$
$s_1 \overset{\alpha}{\dashrightarrow} s_1'$	$(s_1,s_2) \overset{\alpha}{\dashrightarrow} (s_1',s_2')$	$(s_1,s_2) \overset{\alpha}{\dashrightarrow} (s_1',s_2')$	$(s_1,s_2) \overset{\alpha}{\dashrightarrow} \mathsf{u}$
$s_1 \not\overset{\alpha}{\dashrightarrow}$			$(s_1,s_2) \overset{\alpha}{\dashrightarrow} \mathsf{u}$

$\wedge$	$s_2 \overset{\alpha}{\longrightarrow} s_2'$	$s_2 \overset{\alpha}{\dashrightarrow} s_2'$	$s_2 \not\overset{\alpha}{\dashrightarrow}$
$s_1 \overset{\alpha}{\longrightarrow} s_1'$	$(s_1,s_2) \overset{\alpha}{\longrightarrow} (s_1',s_2')$	$(s_1,s_2) \overset{\alpha}{\longrightarrow} (s_1',s_2')$	$(s_1,s_2) \in \lightning$
$s_1 \overset{\alpha}{\dashrightarrow} s_1'$	$(s_1,s_2) \overset{\alpha}{\longrightarrow} (s_1',s_2')$	$(s_1,s_2) \overset{\alpha}{\dashrightarrow} (s_1',s_2')$	
$s_1 \not\overset{\alpha}{\dashrightarrow}$	$(s_1,s_2) \in \lightning$		

Fig. 2. Transition relations for the specification operators $\otimes$, $/$, $\wedge$ on MS

The specification operators composition, quotient and conjunction are described hereafter and we assume that the involved MS always have the same set of actions. Composition of MS ($\otimes$) is defined by synchronizing on shared actions. The rules of $\otimes$ for MS can be seen in Fig. 2; note that only the synchronization of two must transition yields a must transition, in all other cases it yields a may transition.[2]

The two missing operators quotient and conjunction need some more involved definition, because both are partial operators. During quotient as well as conjunction, inconsistencies may arise, however, that does not mean that the whole result is inconsistent; we instead apply a pruning operator ρ to remove all those inconsistent states. More precisely, given an MS S with a set of inconsistent states $\lightning \subseteq St$, the pruned version $\rho(S)$ gives the largest MS which refines S but no state of $\rho(S)$ is related (in the sense of refinement) to an inconsistent state in $\lightning$. The formal definition of pruning can be found in [32].

With pruning at hand, we can define quotient S_1/S_2 (as the dual operator to composition) and conjunction $S_1 \wedge S_2$, as shown in Fig. 2. The set $\lightning$ models in both cases the set of inconsistent states, and in the definition of quotient, the state u is a new *universal* state in which, for every action, there is a looping may transition to the same state u.

[2] The notation $s \not\overset{\alpha}{\dashrightarrow}$ means that there does not exist s' such that $s \overset{\alpha}{\dashrightarrow} s'$, and similar for must transitions.

For writing down contracts based on MS, it is useful to be able to handle dissimilar set of actions when applying specification operators, see [32]. Given an MS S over the set of actions Σ, and a larger set of actions $\Sigma' \supseteq \Sigma$,

- the strong extension of S, written $S_{\uparrow\Sigma'}$, adds for each new action $a \in \Sigma' \setminus \Sigma$ a may and a must loop with that action to all states in S.
- Similar, the weak extension of S, written $S_{\Uparrow\Sigma'}$, adds for each new action (only) a may loop (for all new actions) to all states.

The specification operators are, for the general case, extended to MS with dissimilar sets of actions as follows. If S and T are two MS with sets of actions Σ_S and Σ_T, respectively, and $\Sigma = \Sigma_S \cup \Sigma_T$, then $S \otimes T$ is defined by $S_{\uparrow\Sigma} \otimes T_{\uparrow\Sigma}$, $S \wedge T$ is defined by $S_{\Uparrow\Sigma} \wedge T_{\Uparrow\Sigma}$, and T/S is defined by $T_{\Uparrow\Sigma}/S_{\uparrow\Sigma}$.

Relativized refinement (see Def. 2), induced by modal refinement, can be shown to be equivalent with the following direct definition: If S, T, E are MS over the same set of actions Σ, then $S \leq_E T$ *if and only if* there exists a relation $R \subseteq St_S \times St_E \times St_T$ such that $(s_0, e_0, t_0) \in R$, and for all $(s, e, t) \in R$, all $\alpha \in \Sigma$,

1. if $s \overset{\alpha}{\dashrightarrow} s'$ and $e \overset{\alpha}{\dashrightarrow} e'$ then there exists $t \overset{\alpha}{\dashrightarrow} t'$ such that $(s', e', t') \in R$,
2. if $t \overset{\alpha}{\longrightarrow} t'$ and $e \overset{\alpha}{\dashrightarrow} e'$ then there exists $s \overset{\alpha}{\longrightarrow} s'$ such that $(s', e', t') \in R$.

Every modal contract can be transformed to an equivalent contract in normal form, by weakening the guarantee by the assumption. It turns out that there is a direct definition of a so-called *weakening operator*, that exactly does what we are looking for: $I \leq_A G$ if and only if $I \leq A \triangleright G$, where $A \triangleright G$ is the weakening of G by A. Formally, if A and G are two MS over the same set of actions Σ, then $A \triangleright G$ is defined to be the MS $((St_A \times St_G) \cup \{\mathsf{u}\}, (a_0, g_0), \Sigma, \dashrightarrow, \longrightarrow)$ where u is a fresh state (the universal state), and where the transition relations are defined as shown in the table in Fig. 3.

$\triangleright$	$g \overset{\alpha}{\longrightarrow} g'$	$g \overset{\alpha}{\dashrightarrow} g'$	$g \not\overset{\alpha}{\dashrightarrow}$
$a \overset{\alpha}{\dashrightarrow} a'$	$(a, g) \overset{\alpha}{\longrightarrow} (a', g')$	$(a, g) \overset{\alpha}{\dashrightarrow} (a', g')$	
$a \not\overset{\alpha}{\dashrightarrow}$	$(a, g) \overset{\alpha}{\dashrightarrow} \mathsf{u}$	$(a, g) \overset{\alpha}{\dashrightarrow} \mathsf{u}$	$(a, g) \overset{\alpha}{\dashrightarrow} \mathsf{u}$

Fig. 3. Rules for weakening ($\triangleright$)

Coming back to the example, the contract (A_{Server}, G_{Server}) is obviously not in normal form, but with the weakening operator at hand, we can transform the contract to the semantically equivalent contract $(A_{Server}, G^{nf}_{Server})$ where $G^{nf}_{Server} =_{def} A_{Server} \triangleright G_{Server}$, see Fig. 1(e). As one can see, the normalized contract has lots of additional transitions, and it is often better to draw non-normal form contracts which are usually considerably smaller. Note that (A_{User}, G_{User}) is already in normal form.

We can finally compose our two contracts. As the watchful reader might have already noticed, A_{Server} is expecting the user to answer in any case with the authentication code once a secret message is received. But G_{User} does not provide the authentication code because it may be the case that he/she is not aware of the code. Thus we have an inconsistency arising here, and as a result of applying quotient and conjunction while building the new (weakest) assumption $A_{System} = (A_{Server}/G_{User}) \wedge (A_{User}/G^{nf}_{Server})$, one can see in Fig. 1 that – as expected – the environment is not allowed to issue a secret message anymore. The resulting guarantee G_{System} of the composed contract has been slightly simplified by leaving out some may transitions to a universal state (as in G_{Server}) but the overall contract (A_{System}, G_{System}) is obviously semantically equivalent to $(A_{System}, G^{nf}_{Server} \otimes G_{User})$.[3] Our theory in Sect. 3 now tells us that (A_{System}, G_{System}) is indeed the strongest contract that dominates the contract of the server and the user.

5 Conclusion, Related Work, and Future Work

This paper studies the relationship between specifications of component behaviors and contracts. The general framework for contracts is inspired by the work of Benveniste et al. [10]. They have chosen a trace-based approach to represent interfaces which (as we have shown) is a specification theory and an instance of our proposed abstract contract framework. The idea to equip a specification with implementation and environment semantics has been used by the authors already in [7] where UML protocol state machines were considered as specifications of component interfaces.

Modal contracts have already been introduced and investigated in several previous works, including [19,31]. Raclet and Goessler [19] propose an implementation semantics that is slightly different to ours. In their paper, an implementation I satisfies a contract (A, G) if $A \wedge I \leq_m G$ whenever $A \wedge I$ is defined, which is in fact equivalent to our definition of contract satisfaction, but only for implementations (that are modal specifications where the must and may transition relations coincide). Our satisfaction relation is more powerful as it works for arbitrary modal specifications. Refinement and composition is only syntactically defined, without any semantic considerations as we do it in this paper, hence they lack the universal property for contract compositions. In [31], Quinton and Graf define an abstract framework of contracts which however tends to be technically overloaded due to the integration of complex composition operators. Besides this difference, the satisfaction relation of contracts is the same as in our work. Our notion of (semantic) dominance is inspired by their (syntactical) definition, but still their work lacks of a careful discussion about dominance and the universal property of contract composition. In summary, in comparison

[3] This "inverse" operation to normalizing contracts is in fact useful when drawing larger specifications, and can be automatically applied to a (composed) contract to reduce its number of states and transitions while remaining semantically equivalent.

to both works [19,31], we consider our work as "more semantical" as implementation and environment semantics of contracts are carefully taken into account for the definition of contracts and contract operators.

There are various directions for future work. As an example, we have simplified our setup in this work by ignoring compatibility and consistency issues between interfaces, although we are convinced that they can be integrated without problems. Another major objective is to implement our modal contract theory in the MIO Workbench [9].

References

1. Aarts, F., Vaandrager, F.: Learning I/O Automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 71–85. Springer, Heidelberg (2010)
2. de Alfaro, L., Henzinger, T.A.: Interface automata. In: FSE, pp. 109–120. ACM Press (2001)
3. de Alfaro, L., Henzinger, T.A., Stoelinga, M.I.A.: Timed Interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)
4. de Alfaro, L., da Silva, L.D., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable Interfaces. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 81–105. Springer, Heidelberg (2005)
5. Bauer, S.S., David, A., Hennicker, R., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Moving from specifications to contracts in component-based design. Tech. Rep. 1201, LMU Munich, Germany (January 2012)
6. Bauer, S.S., Fahrenberg, U., Juhl, L., Larsen, K.G., Legay, A., Thrane, C.R.: Quantitative Refinement for Weighted Modal Transition Systems. In: Murlak, F., Sankowski, P. (eds.) MFCS 2011. LNCS, vol. 6907, pp. 60–71. Springer, Heidelberg (2011)
7. Bauer, S.S., Hennicker, R.: Views on Behaviour Protocols and Their Semantic Foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 367–382. Springer, Heidelberg (2009)
8. Bauer, S.S., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: A Modal Specification Theory for Components with Data. In: FACS 2011. LNCS. Springer, Heidelberg (2011)
9. Bauer, S.S., Mayer, P., Schroeder, A., Hennicker, R.: On Weak Modal Compatibility, Refinement, and the MIO Workbench. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 175–189. Springer, Heidelberg (2010)
10. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple Viewpoint Contract-Based Specification and Design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 200–225. Springer, Heidelberg (2008)
11. Bertrand, N., Legay, A., Pinchinat, S., Raclet, J.-B.: A Compositional Approach on Modal Specifications for Timed Systems. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 679–697. Springer, Heidelberg (2009)
12. Caillaud, B., Delahaye, B., Larsen, K.G., Legay, A., Pedersen, M.L., Wasowski, A.: Constraint markov chains. Theor. Comput. Sci. 412(34), 4373–4404 (2011)
13. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource Interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)

14. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: HSCC, pp. 91–100. ACM (2010)
15. Delahaye, B., Caillaud, B., Legay, A.: Probabilistic contracts: a compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects. Formal Methods in System Design 38(1), 1–32 (2011)
16. Doyen, L., Henzinger, T.A., Jobstman, B., Petrov, T.: Interface theories with component reuse. In: EMSOFT, pp. 79–88. ACM Press (2008)
17. Emmi, M., Giannakopoulou, D., Păsăreanu, C.S.: Assume-Guarantee Verification for Interface Automata. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 116–131. Springer, Heidelberg (2008)
18. Godefroid, P., Jagadeesan, R.: On the Expressiveness of 3-Valued Models. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 206–222. Springer, Heidelberg (2002)
19. Goessler, G., Raclet, J.B.: Modal contracts for component-based design. In: SEFM, pp. 295–303. IEEE Computer Society (2009)
20. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: *Don't Know* in the μ-*Calculus*. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 233–249. Springer, Heidelberg (2005)
21. Huth, M., Jagadeesan, R., Schmidt, D.A.: Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 155–169. Springer, Heidelberg (2001)
22. Jones, C.B.: Development methods for computer programs including a notion of interference. Ph.D. thesis, Oxford University Computing Laboratory (1981)
23. Larsen, K.G.: Modal Specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1990)
24. Larsen, K.G., Legay, A., Traonouez, L.-M., Wąsowski, A.: Robust Specification of Real Time Components. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 129–144. Springer, Heidelberg (2011)
25. Larsen, K.G., Nyman, U., Wąsowski, A.: Interface Input/Output Automata. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 82–97. Springer, Heidelberg (2006)
26. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O Automata for Interface and Product Line Theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)
27. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS. IEEE Computer Society (1988)
28. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16(6), 1811–1841 (1994)
29. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)
30. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Software Eng. 7(4), 417–426 (1981)
31. Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: SEFM, pp. 377–381. IEEE Computer Society (2008)
32. Raclet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: A modal interface theory for component-based design. Fundam. Inform. 108(1-2), 119–149 (2011)
33. Raclet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Passerone, R.: Why are modalities good for interface theories? In: ACSD, pp. 119–127. IEEE Computer Society (2009)

34. Sassolas, M., Chechik, M., Uchitel, S.: Exploring inconsistencies between modal transition systems. Software and System Modeling 10(1), 117–142 (2011)
35. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM Trans. Program. Lang. Syst. 33(4), 14 (2011)
36. Wei, O., Gurfinkel, A., Chechik, M.: Mixed Transition Systems Revisited. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 349–365. Springer, Heidelberg (2009)
37. Xu, D.N., Gössler, G., Girault, A.: Probabilistic Contracts for Component-Based Design. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 325–340. Springer, Heidelberg (2010)

The SynchAADL2Maude Tool

Kyungmin Bae[1], Peter Csaba Ölveczky[2],
José Meseguer[1], and Abdullah Al-Nayeem[1]

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

Abstract. SynchAADL2Maude is an Eclipse plug-in that uses Real-Time Maude to simulate and model check Synchronous AADL models. Synchronous AADL is a variant of the industrial modeling standard AADL that supports the modeling of synchronous embedded systems. In particular, Synchronous AADL can be used to define in AADL the synchronous models in the PALS methodology, in which the very hard tasks of modeling and verifying an asynchronous distributed real-time system that should be virtually synchronous can be reduced to the much simpler tasks of modeling and verifying the underlying synchronous design.

1 Introduction

The *Architecture Analysis & Design Language* (AADL) [6] is an industrial modeling standard used in avionics, aerospace, automotive, medical devices, and robotics communities—including Honeywell, Rockwell-Collins, Lockheed Martin, General Dynamics, Airbus, the European Space Agency, Dassault, EADS, Ford, and Toyota—to describe an embedded real-time system as an assembly of software components mapped onto an execution platform.

A number of tools support the formal analysis of different aspects of models in various fragments of AADL. However, since the components in AADL models interact asynchronously, their model checking becomes unfeasible even for fairly small models due to the state space explosion caused by the interleavings.

We therefore define in [1] a variant of AADL, called *Synchronous AADL*, for modeling *synchronous* real-time systems in AADL. This effort was motivated by the observation that many automotive and avionics systems should be *virtually synchronous*—that is, conceptually, there is a logical period during which all components perform a transition and send messages to each other—that must be realized in a distributed environment with network delays, skewed local clocks, etc. Together with colleagues at UIUC and Rockwell-Collins, we have proposed the PALS transformation [3,4], whose key idea is that one can model and verify the much simpler synchronous design, and PALS then provides a correct-by-construction distributed asynchronous model. There are also other transformations relating synchronous and asynchronous systems for distributed real-time architectures, such as the time-triggered architecture (TTA) [2]. Synchronous AADL makes it possible to define such synchronous models in AADL.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 59–62, 2012.

The *SynchAADL2Maude* OSATE[1] plug-in is a recent simulation and linear temporal logic (LTL) model checking tool for Synchronous AADL. The tool automatically synthesizes a Real-Time Maude [5] model from a Synchronous AADL model, provides support to conveniently define LTL properties of the Synchronous AADL model, and performs the Real-Time Maude model checking *within* OSATE. This enables a model-engineering process for important classes of distributed real-time systems that combines the convenience of AADL modeling, the complexity reduction of PALS and TTA, and formal verification in Real-Time Maude. We illustrate the use of SynchAADL2Maude in Section 3 with a virtually synchronous avionics system, whose distributed asynchronous version (even in very simple settings) has millions of reachable states and cannot be feasibly model checked, but where the Synchronous AADL model of the corresponding synchronous PALS design can be verified by our tool in less than a second.

The tool, together with related papers and technical reports, is available at `http://www.cs.illinois.edu/~kbae4/SynchAADL/`.

2 Background: Real-Time Maude and Synchronous AADL

Real-Time Maude [5] is a rewriting-logic-based formal specification language and analysis tool for real-time systems. Real-Time Maude provides simulation capabilities, as well as (unbounded and time-bounded) explicit-state reachability analysis and LTL and timed CTL model checking.

The *Synchronous AADL* modeling language [1] supports the modeling of synchronous designs in AADL, including both synchronous PALS designs and other synchronous designs that can be mapped onto different distributed real-time architectures. Synchronous AADL is an annotated sublanguage of AADL, identifying a set of AADL models that can be considered as synchronous, and adding a *property set* `SynchAADL` to declare Synchronous AADL-specific properties. Since Synchronous AADL is intended to model synchronous *designs*, it disregards the hardware and scheduling features of AADL and focuses on the behavioral and structural subset of AADL, namely, hierarchical system, process, and thread components, ports and connections, and thread behaviors defined in the *behavior annex* standard. The formal Real-Time Maude semantics of Synchronous AADL is defined in [1].

3 Using the SynchAADL2Maude Tool

We exemplify the use of the SynchAADL2Maude tool with an avionics system developed by Steve Miller and Darren Cofer at Rockwell-Collins [4]. In *integrated modular avionics*, there are multiple physically separated *cabinets* on the aircraft so that physical damage does not take out the computer system. The *active standby* system considers the case of two cabinets and focuses on the logic for deciding which side is *active*. The architecture of the system is shown in Figure 1.

[1] The OSATE modeling environment provides a set of Eclipse plug-ins for AADL.

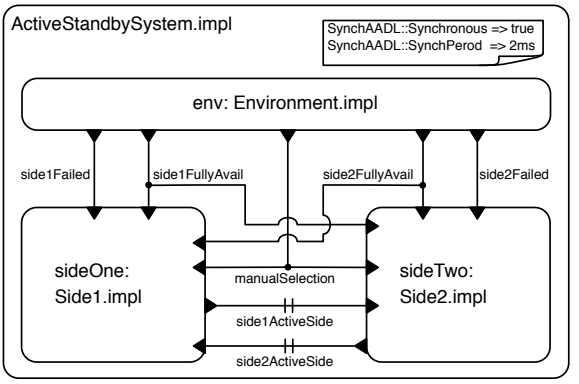


Fig. 1. The architecture of the active standby system

In SynchAADL2Maude, the properties to be verified are managed by an XML file. One important property that the system should satisfy is that *if a side is failed, the other side should become active.* Side i has failed if it has received the value `true` in its `side`i`Failed` port. Using the predefined proposition `value of` *port* `in component` *thread* `is` v, the formula `side`i`Failed` can be defined as follows:

```
<definition> <name>side1Failed</name>
 <value>value of side1Failed in component MAIN -> sideOne -> sideProcess -> sideThread is true
 </value>
</definition>
```

The formulas `side`i`Active` are defined in the same way. The LTL property to be verified is then declared by the `command` tag as follows (where '`~`', '`->`', '`[]`', and '`O`' denote, resp., negation, implication, and the "always" and "next" operators):

```
<command>  <name>R4</name>
 <value type="ltl"> [] (((side1Failed /\ ~side2Failed) -> O (~side2Failed -> side2Active)) /\
                        ((side2Failed /\ ~side1Failed) -> O (~side1Failed -> side1Active)))
 </value>
</command>
```

Figure 2 shows the SynchAADL2Maude window for the active standby system. The `Constraints Check`, `Code Generation`, and `Perform Verification` buttons are used to, respectively, check whether a model is a valid Synchronous AADL model, generate the corresponding Real-Time Maude model, and model check the LTL properties given by the XML property file and shown in the "AADL Property Requirement" table. The results of the model checking are shown in the "Maude Console." Counterexamples from the LTL model checking are presented in a reasonably intuitive and concise way.

We have verified each requirement of the Synchronous AADL model of the active standby system, which has 203 reachable states, in 0.6 seconds on an Intel Xeon 2.93 GHz with 24GB RAM. As shown in [3], where we define directly in

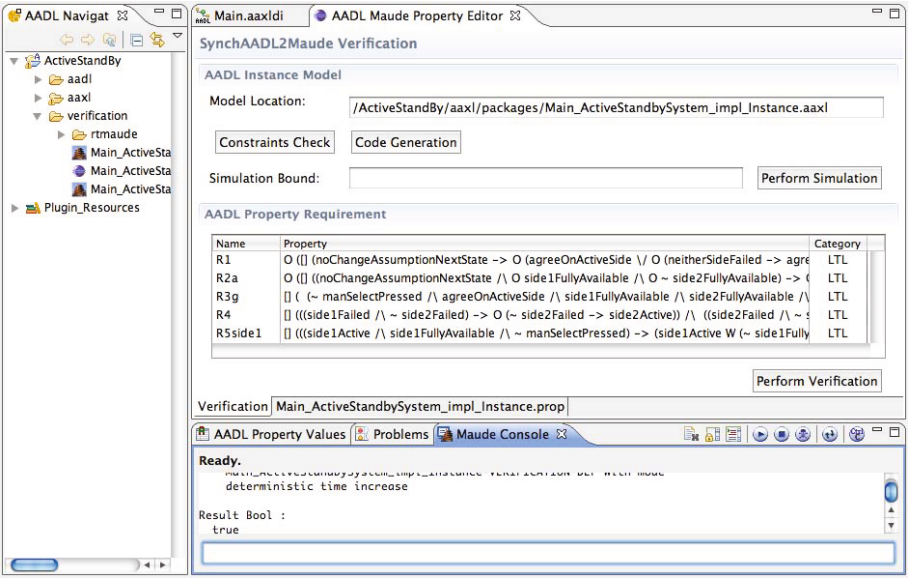


Fig. 2. SynchAADL2Maude window in OSATE

Real-Time Maude models of both the synchronous and the asynchronous design of the active standby system, it is unfeasible to model check the corresponding *asynchronous* design: the *simplest possible* asynchronous model—with no message delays, no execution times, and perfect local clocks—has 3,047,832 reachable states and its model checking takes 1,249 seconds. If the message delay can be either 0 or 1 then no model checking terminates in reasonable time.

Acknowledgments. This work has been supported by Boeing Corporation under grant C8088, by The Research Council of Norway, and by the "Programa de Apoyo a la Investigación y Desarrollo" (PAID-02-11) of the Universitat Politécnica de València.

References

1. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and Its Formal Analysis in Real-Time Maude. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 651–667. Springer, Heidelberg (2011)
2. Kopetz, H., Bauer, G.: The time-triggered architecture. Proc. IEEE 91(1) (2003)
3. Meseguer, J., Ölveczky, P.C.: Formalization and Correctness of the PALS Architectural Pattern for Distributed Real-Time Systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 303–320. Springer, Heidelberg (2010)
4. Miller, S.P., Cofer, D.D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proc. DASC 2009. IEEE (2009)
5. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
6. SAE AADL Team: AADL homepage (2009), http://www.aadl.info/

Consistency of Service Composition

José Luiz Fiadeiro[1] and Antónia Lopes[2]

[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 address the problem of ensuring that, when an application executing a service binds to a service that matches required functional properties, both the application and the service can work together, i.e., their composition is consistent. Our approach is based on a component algebra for service-oriented computing in which the configurations of applications and of services are modelled as asynchronous relational nets typed with logical interfaces. The techniques that we propose allow for the consistency of composition to be guaranteed based on properties of service orchestrations (implementations) and interfaces that can be checked at design time, which is essential for supporting the levels of dynamicity required by run-time service binding.

1 Introduction

In recent years, several proposals have been made to characterise the fundamental structures that support service-oriented computing (SOC) independently of the specific languages or platforms that may be adopted to develop or deploy Web services. In this paper, we contribute to this effort by investigating the problem of ensuring that, when an application executing a service binds to a service that it requested, the result is consistent, i.e., both the executing service and the service to which it binds can operate together in the sense that there is a trace that represents an execution of both. In particular, we show how consistency can be checked based on properties of service orchestrations (implementations) and interfaces that can be established at design time. Checking for consistency at discovery time would not be credible because, in SOC, there is no time for the traditional design-time integration and validation activities as the SOA middleware brokers need to discover and bind services at run time.

In order to formulate a notion of consistency and the conditions under which it can be ensured in a way that is as general as possible, i.e., independently of any particular orchestration model (automata, Petri-nets, and so on), we adopt a fairly generic model of behaviour based on traces of observable actions as executed by implementations of services in what are often called 'global computers' — computational infrastructures that are available globally and support the distributed execution of business applications. More precisely, we build on the asynchronous, message-oriented model of interaction that we developed in [10] over which interfaces are defined as temporal logic specifications. That is, instead of a process-oriented notion of interface (which prevails in

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 63–77, 2012.

most approaches to service orchestration and choreography), we adopt a declarative one that follows in the tradition of logic-oriented approaches to concurrent and distributed system design (as also adopted in [8] for component-based design). One advantage of this approach is that we are able to distinguish between what can be checked at design time to ensure consistency of binding (based on implementations) and what needs to be checked at discovery (run) time to ensure compatibility (based on interfaces).

Having this in mind, in Section 2, we introduce some basic definitions around trace-based models of behaviour and revisit and reformulate, in a more general setting, the notion of asynchronous relational net (ARN) proposed in [10]. In Section 3, we define consistency and prove a sufficient condition for the composition of two consistent ARNs to be consistent, which is based on the notion of safety property. Finally, in Section 4, we discuss which logics support interfaces for ARNs that implement safety properties and propose one such logic that is sufficiently expressive for SOC.

Related work. Most formal approaches that have been proposed for either service choreography or orchestration are process-oriented, for example through automata, labelled transition systems or Petri-Nets. In this context, several notions of compatibility have been studied aimed at ensuring that services are composable. Compatibility in this context may have several different meanings. For example, [16] addresses the problem of ensuring that, at service-discovery time, requirements placed by a requester service are matched by the discovered services — the requirements of the requester are formulated in terms of a graph-based model of a protocol that needs to be simulated by the BPEL orchestration of any provided service that can be discovered. That is, compatibility is checked over implementations. However, one has to assume that the requester has formulated its requirements in such a way that, once bound to a discovered service that meets the requirements, its implementation will effectively work together with that of the provided service in a consistent way — a problem not addressed in that paper.

A different approach is proposed in [6] where compatibility is tested over the interfaces of services (not their implementations), which is simpler and more likely to be effective because a good interface should hide (complex) information that is not relevant for compatibility. A limitation of this approach is that it is based on a (synchronous) method-invocation model of interaction: as argued in [13], web-service composition languages such BPEL (the Business Process Execution Language [20]) rely on an (asynchronous) message-passing model, which is more adequate for interactions that need to run in a loosely-coupled operating environment. An example of an asynchronous framework is the class of automata-based models proposed in [5,7,11], which is used for addressing a number of questions that arise in *choreography*, namely the realisability of conversation protocols among a fixed number of peers in terms of the local behaviour generated by implementations of the peers. Our interest is instead in how dependencies on external services that need to be discovered can be reflected in the interface of a peer and in determining properties of such interfaces that can guarantee that the *orchestration* of the peer can bind to that of a discovered service in a way that ensures consistency of the joint behaviour.

In this respect, the notions of interface that are proposed in [6] do not clearly separate between interfaces for clients of the service and interfaces for providers of required external services, i.e., the approach is not formulated in the context of run-time service

discovery and binding. Furthermore, [6] does not propose a model of composition of implementations (what is called a component algebra in [8]) so one has to assume that implementations of services with compatible interfaces, when composed, are 'consistent'. The interface and component algebra that we proposed in [10] makes a clear distinction between interfaces for services provided and services requested. Our model, which extends the framework proposed by de Alfaro and Henzinger for component-based systems [8], is based on an asynchronous version of relational nets adapted to SCA (the Service Component Architecture [17]) and defines a component algebra that is is compositional in relation to the binding of required with provided service interfaces. The purpose of this paper is precisely to formulate a notion of consistency at the level of the component algebra through which one can ensure, at design time, that matching required with provided services at the interface level leads to a consistent implementation of the composite service when binding the implementations of the requester and the provider services.

2 Asynchronous Relational Nets

2.1 Trace-Based Models of Behaviour

The processes that execute in SOC are typically reactive and interactive. Their behaviour can be observed in terms of the actions that they perform. For simplicity, we use a linear model, i.e., we observe streams of actions (which we call segments). In order not to constrain the environment in which processes execute and communicate, we take traces that capture complete behaviours 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. Observing an empty set of actions in a trace reflects an execution step during which a process is idle, i.e., a step performed by the environment without the involvement of the process.

More precisely, given a set A (of actions), a *trace* λ over A is an element of $(2^A)^\omega$, i.e., an infinite sequence of sets of actions. We denote by $\lambda(i)$ the i-th element of λ, by λ_i the prefix of λ that ends at $\lambda(i)$, and by λ^i the suffix of λ that starts at $\lambda(i)$. A *segment* over A is an element of $(2^A)^*$, i.e., a finite sequence of sets of actions. We use $\pi \prec \lambda$ to mean that the segment π is a prefix of λ. Given $A' \subseteq A$, we denote by $(\pi \cdot A')$ the segment obtained by extending π with A'.

Definition 1 (Property and Closure). *Let A be an alphabet.*

- *A* property Λ *over A is a subset of* $(2^A)^\omega$.
- *Given* $\Lambda \subseteq (2^A)^\omega$, *we define* $\Lambda^f = \{\pi \in (2^A)^* : \exists \lambda \in \Lambda(\pi \prec \lambda)\}$ — *the set of prefixes of traces in* Λ, *also called the* downward closure *of* Λ.
- *Given* $\Lambda \subseteq (2^A)^\omega$, *we define* $\bar{\Lambda} = \{\lambda \in (2^A)^\omega : \forall \pi \prec \lambda(\pi \in \Lambda^f)\}$ — *the set of traces whose prefixes are in* Λ, *also called the* closure *of* Λ.
- *A property* Λ *is said to be* closed *iff* $\Lambda \supseteq \bar{\Lambda}$.

The closure operator is defined according to the Cantor topology on $(2^A)^\omega$ used in [1] for characterising safety and liveness properties (see also [4]). In that topology, the closed sets are the safety properties (and the dense ones are the liveness properties).

Functions between sets of actions, which we call alphabet maps, are useful for defining relationships between individual processes and the networks in which they operate. Alphabet maps induce translations that preserve and reflect closed properties:

Proposition and Definition 2 (Translation). *Let $\sigma{:}A{\to}B$ be a function (alphabet map).*

- *For every $\lambda'{\in}(2^B)^\omega$, we define $\lambda'|_\sigma{\in}(2^A)^\omega$ pointwise as $\lambda'|_\sigma(i){=}\sigma^{-1}(\lambda'(i))$.*
- *For every set $\Lambda{\subseteq}(2^A)^\omega$, we define $\sigma(\Lambda) = \_|_\sigma{}^{-1}(\Lambda) = \{\lambda'{\in}(2^B)^\omega : \lambda'|_\sigma{\in}\Lambda\}$.*
- *For every closed property Λ over A, $\sigma(\Lambda)$ is a closed property over B.*
- *For every closed property Λ' over B, $\Lambda'|_\sigma$ is a closed property over A.*

Notice that every alphabet map σ defines a contravariant translation $\_|_\sigma$ between traces by taking the inverse image of the set of actions performed at each step.

2.2 Asynchronous Relational Nets

In this section, we revisit the component algebra proposed in [10] based on the notion of asynchronous relational net (ARN). The main difference is that, where in [10] we formalised ARNs in terms of logical specifications, we are now interested in behaviours (model-theoretic properties) so that we can define and analyse consistency in logic-independent terms. We revisit specifications in the context of interfaces in Sec. 4.

In an asynchronous communication model, interactions are based on the exchange of messages that are transmitted through channels. We organise messages in sets that we call ports: a *port* is a finite set (of messages). 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). Therefore, every port M has a partition $M^- \cup M^+$. The actions of sending (publishing) or receiving (being delivered) a message m are denoted by $m!$ and $m¡$, 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 $m¿$ for the action of discarding the message: as discussed later, processes cannot refuse the delivery of messages but they should be able to discard them, for example if they arrive outside the protocol expected by the process.

More specifically, if M is a port:

- Given $m{\in}M^-$, the set of actions associated with m is $A_m = \{m!\}$.
- Given $m{\in}M^+$, $A_m = \{m¡, m?, m¿\}$
- The set of actions associated with M is $A_M = \bigcup_{m\in M} A_m$.

A *process* consists of a finite set γ of mutually disjoint ports — i.e., each message that a process can exchange belongs to exactly one of its ports — and a non-empty property Λ over $A_\gamma = \bigcup_{M\in\gamma} A_M$ defining the behaviour of the process.

Interactions in ARNs are established through channels. A *channel* consists of a set M of messages and a non-empty property Λ over the alphabet $A_M{=}\{m!, m¡ : m{\in}M\}$. Channels connect processes through their ports. Given ports M_1 and M_2 and a channel $\langle M, \Lambda\rangle$, a *connection* between M_1 and M_2 via $\langle M, \Lambda\rangle$ consists of a pair of injective maps $\mu_i{:}M{\to}M_i$ such that $\mu_i^{-1}(M_i^+) = \mu_j^{-1}(M_j^-)$, $\{i, j\}{=}\{1, 2\}$ — i.e., a connection

establishes a correspondence between the two ports such that any two messages that are connected have opposite polarities. Each injection μ_i is called the *attachment* of M to M_i. We denote the connection by the triple $\langle M_1 \overset{\mu_1}{\leftarrow} M \overset{\mu_2}{\rightarrow} M_2, \Lambda\rangle$.

Definition 3 (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* $\langle \gamma_p, \Lambda_p\rangle$ *to every node* p *and a connection* $\langle \gamma_c, \Lambda_c\rangle$ *to every edge* c *such that:*
  - *If* $c{=}\{p, q\}$ *then* γ_c *is a pair of attachments* $\langle M_p \overset{\mu_p}{\leftarrow} M_c \overset{\mu_q}{\rightarrow} M_q\rangle$ *for some* $M_p{\in}\gamma_p$ *and* $M_q{\in}\gamma_q$.
  - *If* $\gamma_{\{p,q\}}{=}\langle M_p \overset{\mu_p}{\leftarrow} M_{\{p,q\}} \overset{\mu_q}{\rightarrow} M_q\rangle$ *and* $\gamma_{\{p,q'\}}{=}\langle M'_p \overset{\mu'_p}{\leftarrow} M_{\{p,q'\}} \overset{\mu'_{q'}}{\rightarrow} M'_{q'}\rangle$ *with* $q \neq q'$*, then* $M_p \neq M'_p$.

We also define the following sets:

- $A_p = p.A_{\gamma_p}$ *is the language associated with the node* p.
- $A_\alpha = \bigcup_{p\in P} A_p$ *is the language associated with* α.
- $A_c = \langle p.\_\circ\mu_p, q.\_\circ\mu_q\rangle(A_{M_c})$ *is the language associated with* $\gamma_c{:}\langle M_p \overset{\mu_p}{\leftarrow} M_c \overset{\mu_q}{\rightarrow} M_q\rangle$.
- $\Lambda_\alpha = \{\lambda{\in}(2^{A_\alpha})^\omega{:}\ \forall p{\in}P(\lambda|_p{\in}\Lambda_p) \wedge \forall c{\in}C(\lambda|_c{\in}\Lambda_c)\}$.

We often refer to the ARN through the quadruple $\langle P, C, \gamma, \Lambda\rangle$ 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 Λ returns the corresponding properties. 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.

We take the set Λ_α to define the set of possible traces observed on α — those traces over the alphabet of the ARN that are projected to traces of all its processes and channels. The alphabet of A_α is itself the union of the alphabets of the processes involved translated by prefixing all actions with the node from which they originate.

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), i.e., processes and channels act as *types*. This is why it is essential that, in the ARN, it is possible to trace actions to the instances of processes where they originate (all the actions of channels are mapped to actions of processes through the attachments so it is enough to label actions with nodes).

In general, not every port of every process (instance) of an ARN is necessarily connected to a port of another process. Such ports provide the points through which the ARN can interact with other ARNs. An *interaction-point* of an ARN $\alpha = \langle P, C, \gamma, \Lambda\rangle$ 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 of ARNs [10]:

Proposition and Definition 4 (Composition of ARNs). *Let* $\alpha_1 = \langle P_1, C_1, \gamma_1, \Lambda_1 \rangle$ *and* $\alpha_2 = \langle P_2, C_2, \gamma_2, \Lambda_2 \rangle$ *be ARNs such that* P_1 *and* P_2 *are disjoint, and a family* $w^i = \langle M_1^i \xleftarrow{\mu_1^i} M \xrightarrow{\mu_2^i} M_2^i, \Psi^i \rangle$ $(i = 1 \ldots n)$ *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* $p_1^i \neq p_1^j$ *if* $i \neq j$ *and* $p_2^i \neq p_2^j$ *if* $i \neq j$. *The composition*

$$\alpha_1 \; ||\Big|_{\langle p_1^i, M_1^i \rangle, w^i, \langle p_2^i, M_2^i \rangle}^{i=1\ldots n} \; \alpha_2$$

is the ARN whose graph is $\langle P_1 \cup P_2, C_1 \cup C_2 \cup \bigcup_{i=1\ldots n} \{p_1^i, p_2^i\} \rangle$ *and whose 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.

In order to illustrate the notions introduced in the paper, we consider a simplified bank portal that mediates the interactions between clients and the bank in the context of different business operations such as the request of a credit. Fig. 1 depicts an ARN with two interconnected processes that implement this business operation. Process *Clerk* is responsible for the interaction with the environment and for making decisions on credit requests, for which it relies on an external process $RiskEvaluator$ that is able to evaluate the risk of the transaction. The graph of this ARN consists of two nodes $c{:}Clerk$ and $e{:}RiskEvaluator$ and an edge $\{c, e\}{:}w_{ce}$ where:

- *Clerk* is a process with two ports: L_c and R_c. In port L_c, the process receives messages $creditReq$ and $accept$ and sends $approved$, $denied$ and $transferDate$. Port R_c has outgoing message $getRisk$ and incoming message $riskValue$. The *Clerk*'s behaviour is as follows: immediately after the delivery of the first $creditReq$ message on port L_c, it publishes $getRisk$ on R_c; then it waits five time units for the delivery of $riskValue$, upon which it either publishes $denied$ or $approved$ (we abstract from the criteria that it uses for deciding on the credit); if $riskValue$ does not arrive by the deadline, *Clerk* publishes $denied$ on L_c; after sending $approved$ (if ever), *Clerk* waits twenty time units for the delivery of $accept$, upon which it sends $transferDate$; all other deliveries of $creditReq$ and $accept$ are discarded. The property that corresponds to this behaviour is denoted by Λ_c in Fig. 1.
- *RiskEvaluator* is a process with a single port (L_e) with incoming message $request$ and outgoing message $result$. Its behaviour is quite simple: every time $request$ is delivered, it takes no more than three time units to publish $result$. The property that corresponds to this behaviour is denoted by Λ_e in Fig. 1.
- The port R_c of *Clerk* is connected with the port L_e of *RiskEvaluator* through $w_{ce}{:}\langle R_c \xleftarrow{\mu_e} \{m, n\} \xrightarrow{\mu_c} L_e, \Lambda_w \rangle$, with $\mu_c{=}\{m \mapsto getRisk, n \mapsto riskValue\}$, $\mu_e{=}\{m \mapsto request, n \mapsto result\}$. The corresponding channel is reliable: it ensures to delivering $getRisk$, which *RiskEvaluator* receives as $request$, and it ensures to delivering $result$, which *Clerk* receives as $riskValue$, both without any delay. The property that corresponds to this behaviour is denoted by Λ_w in Fig. 1.

3 Consistency

An important property of ARNs, and the one that justifies this paper, is consistency:

Definition 5 (Consistent ARN). *An ARN* α *is said to be* consistent *if* Λ_α *is not empty.*

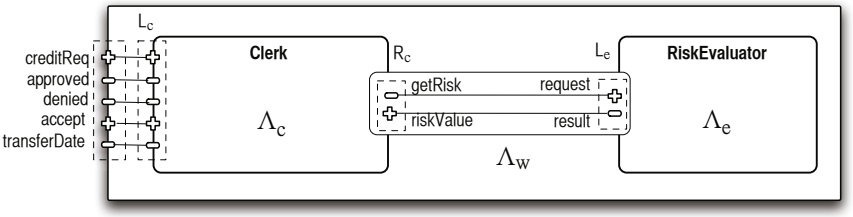


Fig. 1. An example of an ARN with two processes connected through a channel

Consistency means that the processes, interconnected through the channels, can cooperate and generate at least a joint trace. Naturally, one cannot expect every ARN to be consistent as the interference established through the connections may make it impossible for the processes involved to make progress together. Therefore, some important questions, which this paper attempts to answer, are: *How can one check that an ARN α is consistent without calculating the set Λ_α? How can one guarantee that the composition of two consistent ARNs is consistent based on properties of the ARNs and the interconnections that can be checked at design time?*

In order to answer these questions, we are going to discuss a related property: the ability to make (finite) progress no matter the segment that the ARN has executed, which we call progress-enabledness. We show that, for certain classes of ARNs, progress-enabledness implies consistency. We also provide sufficient conditions for the composition of two progress-enabled ARNs to be progress-enabled that can be checked at design time.

3.1 Progress-Enabled ARNs

Consistency is about infinite behaviours, i.e., it concerns the ability of all processes and channels to generate a full joint trace. However, it does not guarantee that, having engaged in a joint partial trace (finite segment), the processes can proceed: it may happen that a joint partial trace is not a prefix of a joint (full) trace, which would be undesirable as it is not possible for individual processes to anticipate what other processes will do — as discussed in Sec. 4, interconnections in the context of SOC are established at run time based on interfaces that capture *what* processes do, not *how* they do it. This is why, in [10], we introduced another useful property of ARNs: that, after any joint partial trace, a joint step can be performed.

Definition 6 (Progress-Enabled ARN). *For every ARN α, let*

$$\Pi_\alpha = \{\pi \in 2^{A_\alpha *} : \forall p \in P(\pi|_p \in \Lambda_p^f) \wedge \forall c \in C(\pi|_c \in \Lambda_c^f)\}$$

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

The set Π_α consists of all the partial traces that the processes and channels can jointly engage in. Notice that, as long as the processes and channels involved in α are consistent, Π_α is not empty: it contains at least the empty trace!

Therefore, by itself, being progress-enabled does not guarantee that an ARN is consistent: moving from finite to infinite behaviours requires the analysis of what happens

'at the limit'. A progress-enabled but inconsistent ARN guarantees that all the processes and channels will happily make joint progress but at least one will be prevented from achieving a successful full trace at the limit. Therefore, it seems justifiable that we look for a class of ARNs for which being progress-enabled implies consistency, which we do in the next subsection. However, in relation to the points that we raised at the beginning of this section, we still need to show that, by investigating a stronger property (being progress-enabled and consistent), we have not made the questions harder to answer.

In [10], we also identified properties of ARNs and channels that guarantee that the composition of two progress-enabled ARNs is progress-enabled: 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 7 (Delivery-enabled). *Let* $\alpha{=}\langle P, C, \gamma, \Lambda\rangle$ *be an ARN,* $\langle p, M\rangle{\in}I_\alpha$ *one of its interaction-points, and* $D_{\langle p,M\rangle}{=}\{p.m¡ : m{\in}M^+\}$. *We say that* α *is delivery-enabled in relation to* $\langle p, M\rangle$ *if, for every* $(\pi{\cdot}A){\in}\Pi_\alpha$ *and* $B{\subseteq}D_{\langle p,M\rangle}$, $(\pi{\cdot}B \cup (A\backslash D_{\langle p,M\rangle})){\in}\Pi_\alpha$.

That is, being delivery-enabled at an interaction point requires that any joint prefix of the ARN can be extended by any set of messages delivered at that interaction-point. Note that this does not interfere with the decision of the process to publish messages: $B{\cup}(A\backslash D_{\langle p,M\rangle}))$ retains all the publications present in A. Also notice that accepting the delivery of a message does not mean that a process will act on it; this is why we distinguish between executing a delivered message $(m?)$ and discarding it $(m¿)$. For example, the processes $Clerk$ and $RiskEvaluator$ informally described in Sec. 2.2 define, individually, atomic ARNs that are delivery-enabled: they put no restrictions on the delivery of messages.

Definition 8 (Publication-enabled). *Let* $h{=}\langle M,\Lambda\rangle$ *be a channel and* $E_h = \{m! : m \in M\}$. *We say that* h *is publication-enabled iff, for every* $(\pi{\cdot}A){\in}\Lambda^f$ *and* $B{\subseteq}E_h$, *we have* $\pi{\cdot}(B{\cup}(A\backslash E_h)){\in}\Lambda^f$.

The requirement here is that any prefix can be extended by the publication of any set of messages, i.e., the channel should not prevent processes from publishing messages. Notice that this does not interfere with the decision of the channel to deliver messages: $(B{\cup}(A\backslash E_h))$ retains all the deliveries present in A. An example is the channel used in Fig. 1, which we informally described in Sec. 2.2.

These two properties allow us to prove that the composition of two progress-enabled ARNs is progress-enabled [10]:

Theorem 9. *Let* $\alpha = (\alpha_1 \|^{i=1\ldots n}_{\langle p_1^i, M_1^i\rangle, w^i, \langle p_2^i, M_2^i\rangle} \alpha_2)$ *be a composition of progress-enabled ARNs where, for each* $i = 1\ldots n$, $w^i = \langle M_1^i \xleftarrow{\mu_1^i} M \xrightarrow{\mu_2^i} M_2^i, \Lambda^i\rangle$. *If, for each* $i{=}1\ldots n$, 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* $h^i{=}\langle M^i,\Lambda^i\rangle$ *is publication-enabled, then* α *is progress-enabled.*

3.2 Safe ARNs

The class of ARNs for which we can guarantee consistency are those that involve only closed (safety) properties (cf. Def. 1). As discussed above, progress-enabledness guarantees that all the processes and channels can progress by making joint steps but does

not guarantee that successful full traces will be obtained at the limit. Choosing to work with safety properties essentially means that 'success' does not need to be measured at the limit, i.e., checking the ability to make 'good' progress is enough.

From a methodological point of view, restricting ARNs to safety properties is justified by the fact that, within SOC, we are interested in processes whose liveness properties are bounded (bounded liveness being itself a safety property). This is because, in typical business applications, one is interested only in services that respond within a fixed (probably negotiated) delay. In SOC, one does not offer as a service the kind of systems that, like operating systems, are not meant to terminate

Definition 10 (Safe processes, channels and ARNs). *A process* $\langle \gamma, \Lambda \rangle$ *(resp. channel* $\langle M, \Lambda \rangle$*) is said to be* safe *if* Λ *is closed. A* safe ARN *is one that is labelled with safe processes and channels.*

Proposition 11. *For every safe ARN* α*,* Λ_α *is a closed (safety) property.*

Proof. Λ_α *is the intersection of the images of the properties of the processes and channels associated with the nodes and edges of the graph. According to Prop. 2, those images are safety properties. The result follows from the fact that an intersection of closed sets in any topology is itself a closed set.*

Theorem 12 (Consistency). *Any safe progress-enabled ARN is consistent.*

Proof. Given that the processes and channels in a safe ARN are consistent, Π_α *(cf. Def. 6) is not empty (it contains at least the empty segment* ϵ*).* Π_α *can be organised as a tree, which is finitely branching because* A_α *is finite. If the ARN is progress-enabled, the tree is infinite. By Kőnigs lemma, it contains an infinite branch* λ*.*

We now prove that $\lambda \in \Lambda_\alpha$*, i.e.,* $\lambda|_p \in \Lambda_p$ *for all* $p \in P$ *and* $\lambda|_c \in \Lambda_c$ *for all* $c \in C$*. Let* $p \in P$ *and* $\pi \prec \lambda|_p$*. We know that* π *is of the form* $\pi'|_p$ *where* $\pi' \in \Pi_\alpha$*. Therefore,* $\pi \in \Lambda_p^f$*. It follows that* $\lambda|_p \in \overline{\Lambda_p}$*. Because* Λ_p *is closed, we can conclude that* $\lambda|_p \in \Lambda_p$*. The same reasoning applies to all channels.*

Note that, in the case of non-safe ARNs, being progress-enabled is a necessary but not sufficient condition to ensure consistency. For example, consider the following two processes: P recurrently sends a given message m and Q is able to receive a message n but only a finite, though arbitrary, number of times. If these processes are interconnected through a reliable channel that ensures to delivering n every time m is published, it is easy to conclude that the resulting ARN is not consistent in spite of being progress-enabled: after having engaged in any joint partial trace, both processes and the channel can proceed (Q will let the channel deliver n once more if necessary); however, they are not able to generate a full joint trace because P will want to send m an infinite number of times and Q will not allow the channel to deliver n infinitely often.

Because the composition of safe ARNs through safe channels is safe, Theo. 9 can be generalised to guarantee consistency of composition:

Corollary 13 (Consistency of composition). *The composition of safe progress-enabled ARNs is both safe and progress-enabled (and, hence, consistent) provided that interconnections are made through safe publication-enabled channels and over interaction-points in relation to which the ARNs are delivery-enabled.*

It remains to determine how ARNs can be proved to be safe, progress-enabled, and delivery-enabled in relation to interaction points, and channels to be safe and publication-enabled. In this respect, another important result (see [10] for details) is that the composition of two ARNs is delivery-enabled in relation to all the interaction-points of the original ARNs that remain disconnected and in relation to which they are delivery-enabled. Therefore, because every process defines an (atomic) progress-enabled ARN (by virtue of being consistent), the proof that an ARN is progress-enabled can be reduced to checking that individual processes are delivery-enabled in relation to their ports and that the channels are publication-enabled. On the other hand, ensuring that processes and channels are safe relates to the way they are specified and implemented.

All these questions are addressed in the next section, where we also discuss how service interfaces should be specified in the context of orchestrations that are safe and progress-enabled. In particular, we show that all the properties that can guarantee consistent composition can be checked at (process/channel) design time, not at (ARN) composition time (which, in SOC, is done at run time).

4 Interface Specifications for Safe ARNs

4.1 Interfaces and Orchestrations

Making the discovery and binding of services to be based on interfaces, not implementations, has the advantage of both simplifying those processes (as interfaces should offer a more abstract view of the behaviour of the services) and decoupling the publication of services in registries from their instantiation when needed. In [10] we proposed an interface theory for ARNs based on linear temporal logic (LTL), which distinguishes between provides- and requires-points:

- A *provides-point* r consists of a port M_r together with a consistent set of sentences Φ_r over A_{M_r} that express what the service offers to any of its clients.
- A *requires-point* r consists of a port M_r and a consistent set of sentences Φ_r over A_{M_r} that express what the service requires from an external service, together with a consistent set of sentences Ψ_r over $\{m!, m¡ \colon m{\in}M_r\}$ that express requirements on the channel through which it expects to interact with the external service.
- Matching a requires-point of a service interface with a provides-point of another service interface amounts to checking that the specification of the latter entails that of the former.

In Fig. 2, we present an example of an interface for a credit service using a graphical notation similar to that of SCA. On the left, we have a provides-point $Customer$ and, on the right, a requires-point $IRiskEvaluator$. The set of sentences Φ_c, in the logic discussed in the next subsection, specifies the service offered at $Customer$:

- $(creditReq¡\ \mathcal{R}\ (creditReq¡ \supset \Diamond_{\leq 10}(approved! \vee denied!)))$ — either $approved$ or $denied$ are published within ten time units of the first delivery of $creditReq$.
- $\Box(approved! \supset (accept¡\ \mathcal{R}_{\leq 20}\ (accept¡ \supset \Diamond_{\leq 2} transferDate!)))$ — if $accept$ is received within twenty time units of the publication of $approved$, $transferDate$ will be published within 2 time units.

The specification Φ_r of $IRiskEvaluator$ requires the external service to react to the delivery of every $request$ by publishing $result$ in no more than four time units: $\Box(request_{\text{¡}} \supset \Diamond_{\leq 4} result!)$.

The connection with the external service is required to ensure that messages are transmitted immediately to the recipient.

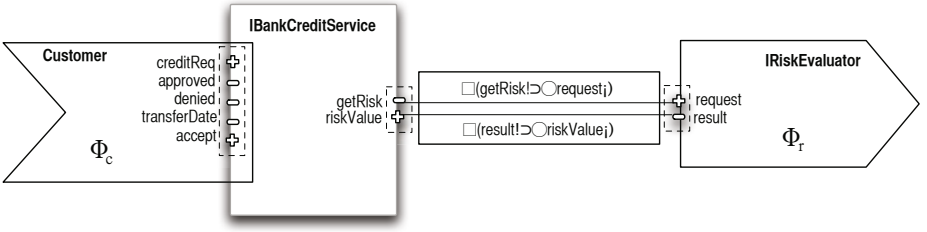


Fig. 2. An example of a service interface

An ARN orchestrates a service-interface by assigning interaction-points to interface-points in such a way that the behaviour of the ARN validates the specifications of the provides-points on the assumption that it is interconnected to ARNs that validate the specifications of the requires-points through channels that validate the corresponding specifications. Notice that ensuring consistency is essential because an interconnection that leads to an inconsistent composition would vacuously satisfy any specification (there would be no behaviours to check against the specification).

Therefore, in order to check that an ARN α orchestrates a service-interface I:

1. For every requires-point r of I, we consider an ARN α_r defined by a single process $\langle M_r, \Lambda_r \rangle$ where Λ_r is a safety property that validates Φ_r and makes α_r delivery-enabled in relation to r, which is representative of the safe and progress-enabled ARNs that can be interconnected at r, i.e., that provides a 'canonical' orchestration of a service that offers a provides-point that matches r.
2. For every requires-point r of I, we consider a channel c_r=$\langle M_r, \Lambda_r \rangle$ where Λ_r is a safety property that validates Ψ_r and makes the channel publication-enabled, which represents the most general channel that can be used for interconnecting an orchestration with an external service.
3. We consider the composition α^* of α with all the α_r via $\langle M_{p_r} \overset{\theta_r}{\leftarrow} M_r \overset{id}{\rightarrow} M_r, \_ \rangle$ where p_r is the interaction-point of α that corresponds to the requires-point r through the mapping θ_r:$M_r^{op} \rightarrow M_p$ (for every port M, we denote by M^{op} the port defined by $M^{op+}=M^-$ and $M^{op-}=M^+$).
4. For α to orchestrate the interface I we require that $\Lambda_{\alpha^*}|_{A_{M_r}} \vDash \Phi_r$ for every provides-point r of I. Notice that $\Lambda_{\alpha^*}|_{A_{M_r}}$ is the projection of the traces of the composed ARN on the alphabet of the provides-point r which, by Prop. 2, is a safety property.

The question now is how to choose such canonical processes $\langle M_r, \Lambda_r \rangle$ (and channels). Typically, in logic, the collection Λ_{Φ_r} of all traces that validate Φ_r (Ψ_r in the case of channels) would meet the requirement because any other ARN would give rise

to fewer traces over A_{M_r}. However, if we want to restrict ourselves to processes and channels that are safe, one has to choose interfaces in the class of specifications that denote safety properties, i.e., for which Λ_{Φ_r} is closed. For example, not every specification in LTL is in that class. The same applies to provides-points because, by Prop. 2, $\Lambda_{\alpha^*}|_{A_{M_r}}$ is a safety property. In this case, because the properties offered in a provides-points derive from the ARN that orchestrates the interface, we would need to be able to support the development of safe processes and channels from logical specifications. Therefore, we need to discuss which logics support that class of specifications.

4.2 A Logic of Safety Properties

Several extensions of LTL (e.g., Metric Temporal Logic – MTL [14]) have been proposed in which different forms of bounded liveness can be expressed through eventuality properties of the form $\Diamond_I \phi$ where I is a time interval during which ϕ is required to become true. Another logic of interest is PROMPT-LTL [15] in which, instead of a specific bound for the waiting time, one can simply express that a sentence ϕ will become true within an unspecified bound — $\Diamond_{\mathbf{p}}\phi$. Yet another logic is PLTL [3] in which one can use variables in addition to constants to express bounds on the waiting time and reason about the existence of a bound (or of a minimal bound) for a response time.

The logic we propose to work with, which we call SAFE-LTL, is a 'safety' fragment of LTL — positive formulas with 'release' and 'next' — which corresponds to the fragment of PLTL where intervals are finite and bounded by constants. This logic can also be seen as a restricted version of Safety MTL [18] (a fully decidable fragment of MTL) where, instead of an explicit model of real-time, we adopt an implicit one in which time is measured by the natural numbers (as in PLTL). From a methodological point of view, the adoption of an implicit, discrete time model can be justified by the fact that, in SOC, one deals with 'business' time where delays are measured in discrete time units that are global (i.e., the time model is synchronous even if the interaction model is asynchronous). This is somewhat different from time-critical systems, for which a continuous time model (i.e., with no fixed minimal time unit) is more adequate.

Definition 14 (SAFE-LTL). *Let A be an alphabet.*

- *The language of* SAFE-*LTL over A is defined by (where $a \in A$):*

$$\phi \ ::= \ a \mid \neg a \mid \phi \vee \phi \mid \phi \wedge \phi \mid \bigcirc \phi \mid \phi \,\mathcal{R}\, \phi$$

- *Sentences are interpreted over $\lambda \in (2^A)^\omega$ as follows :*
  $\lambda \vDash a$ *iff* $a \in \lambda(0)$; $\lambda \vDash \neg a$ *iff* $a \notin \lambda(0)$
  $\lambda \vDash \phi_1 \wedge \phi_2$ *iff* $\lambda \vDash \phi_1$ *and* $\lambda \vDash \phi_2$; $\lambda \vDash \phi_1 \vee \phi_2$ *iff* $\lambda \vDash \phi_1$ *or* $\lambda \vDash \phi_2$
  $\lambda \vDash \bigcirc \phi$ *iff* $\lambda^1 \vDash \phi$
  $\lambda \vDash \phi_1 \,\mathcal{R}\, \phi_2$ *iff, for all j, either* $\lambda^j \vDash \phi_2$ *or there exists* $k<j$ *s.t.* $\lambda^k \vDash \phi_1$

Notice that sentences are in positive form: negation is only available for atomic propositions (actions). This allows us to define $(a \supset \phi)$ as an abbreviation for $(\neg a \vee \phi)$ as used in the interface specifications above. We also use $\Box\phi$ as an abbreviation of $(false\, \mathcal{R}\, \phi)$.

The bounded operators used in the interface specifications given in Sec. 4.1 amount to the following abbreviations where $t \in \mathbb{N}$:

- $(\phi_1 \mathcal{R}_{\leq t} \phi_2) \equiv \phi_2 \wedge (\phi_1 \vee \bigcirc\phi_2) \wedge \cdots \wedge (\phi_1 \vee \bigcirc\phi_1 \vee \cdots \vee \bigcirc^{t-1}\phi_1 \vee \bigcirc^t\phi_2)$
- $(\phi_1 \mathcal{U}_{\leq t} \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge \bigcirc\phi_2) \vee \cdots \vee (\phi_1 \wedge \bigcirc\phi_1 \wedge \cdots \wedge \bigcirc^{t-1}\phi_1 \wedge \bigcirc^t\phi_2)$
- $\Box_{\leq t}\phi \equiv false\, \mathcal{R}_{\leq t}\, \phi \equiv \phi \wedge \bigcirc\phi \wedge \cdots \wedge \bigcirc^t\phi$
- $\Diamond_{\leq t}\phi \equiv true\, \mathcal{U}_{\leq t}\, \phi \equiv \phi \vee \bigcirc\phi \vee \cdots \vee \bigcirc^t\phi$

Theorem 15 (Safety). *All the sentences of* SAFE-*LTL express safety properties, i.e., for every sentence ϕ, the set of traces that satisfy it is closed.*

Proof. See [19] for a similar logic that uses 'unless' instead of 'release'.

Corollary 16 (Safe specifications). *It follows from the previous theorem that all specifications over* SAFE-*LTL are safe, i.e., for all sets of sentences Φ, the set Λ_Φ of all traces λ such that $(\lambda \vDash \Phi)$ is a safety property.*

Proof. The results follow from the fact that the intersection of any number of closed properties is closed.

4.3 Ensuring Delivery/Publication-Enabledness

In addition to making sure that specifications generate safety properties, it is important to guarantee that specifications associated with requires-points generate processes that are delivery-enabled in relation to their port and channels that are publication-enabled. Ensuring delivery/publication-enabledness is not the same as proving that an implementation satisfies a specification because those properties are not expressible as sentences whose satisfaction can be checked over individual traces: they need to be checked over the set of all traces that satisfy the specification.

Traces are observations of the behaviours of systems that implement processes. Typical examples of (models of) such systems that are used in association with a logic are finite automata of some kind such that, for every specification $\langle A, \Phi\rangle$, there is a system S_Φ over the alphabet A such that Λ_{S_Φ}=Λ_Φ. The idea is then to check delivery/publication-enabledness directly over S_Φ.

In the case of LTL, systems are *non-deterministic Bücchi automata* (NBAs) [21]. An NBA over an alphabet A is a tuple of the form $\langle Q, \delta, Q_0, Q_\infty\rangle$ where Q is a finite set of states, $Q_0 \subseteq Q$ is the subset of initial states, $Q_\infty \subseteq Q$ is the set of accepting states, and $\delta : Q \times A \to 2^Q$ is the transition relation. The property defined by $\langle Q, \delta, Q_0, Q_\infty\rangle$ is the set of infinite sequences of elements of A that, starting on an initial state, generate a run that visits at least one of the accepting states infinitely often.

In relation to safety properties, there is also a *closure* operator on NBAs [2]: the closure of $\langle Q, \delta, Q_0, Q_\infty\rangle$ is $\langle Q, \delta, Q_0, Q\rangle$, i.e., the NBA obtained by making all states accepting. A reduced NBA (i.e., one in which every state leads to an accepting state) defines a safety property if and only if its closure defines the same property. Furthermore, every NBA is equivalent to a reduced one.

Therefore, given that we are interested in working with safe specifications, we can choose closed reduced NBAs as models of implementations of processes and channels. In this case, it is easy to see that all that needs to be checked for processes (resp. channels) to be delivery (resp. publication) enabled is that, from every state of the automata that implement them, the set of transitions from that state satisfies the corresponding

property, i.e., for every set of deliveries (resp. publications), there is a transition that delivers (resp. publishes) exactly those messages. As a result, the complexity of the checking process is in the order of the product of the size of the automaton and of the sub-language of deliveries/publications.

5 Concluding Remarks

In this paper, we discussed the problem of ensuring that the composition of orchestrations of matching service interfaces is consistent, i.e., that the orchestrations of both services can effectively work together when interconnected through the communication channels that bind them. Our findings led us to propose a refinement of the service interface and component algebra presented in [10] in which services are orchestrated by asynchronous relational nets that exhibit only safety properties (i.e., any 'bad' behaviour should be able to detected after a finite number of steps) and are progress-enabled (i.e., always able to make progress, even if by remaining idle). The advantages of working with safe progress-enabled ARNs are that they are consistent (Theo. 12) and closed under composition provided that interconnections are made through channels that are safe and publication-enabled and over interaction-points in relation to which the ARNs are delivery-enabled (Cor. 13).

We also investigated the nature of the logics that should be used for specifying service interfaces and describing the processes and channels through which services are orchestrated. In particular, we exhibited a fragment of LTL in which only safety properties can be specified and argued that this fragment is expressive enough for the typical properties through which service interfaces are specified. In this setting, binding services, through the provides-points of their interfaces, to requires-points of the interfaces of discovered services, leads to a consistent composition of the service orchestrations.

Finally, we showed that, by using a logic such as SAFE-LTL, closed reduced NBAs can be used as models of implementations of safe processes and channels, and that checking processes/channels for delivery/publication enabledness can be done over those automata with a complexity that is in the order of the product of the size of the automata and of the sub-languages of deliveries/publications. Equally importantly, these checks can be made at design time, i.e., when implementations are chosen for orchestrating service interfaces. Therefore, there is no need for any additional checking to be made at discovery/run time to guarantee consistency; the only checking that needs to be made at run time is that the specifications of provides-points entail the specifications of the corresponding requires-points.

One point that we intend to investigate further concerns the interplay between consistency, safety, and the behavioural model. We intend to explore the use of sub-domains of traces that are applicable to SOC and generalise the underlying time model (and associated logic) using the notion of 'safety relative to a given condition' developed in [12]. Choosing a sub-domain can have an impact in the structure of the automata and the complexity of checking that processes and channels satisfy delivery/publication enabledness (and that ARNs orchestrate service interfaces), which are aspects that we did not have space left in the paper to analyse and explain in full.

Acknowledgments. We would like to thank Nir Piterman for many helpful comments and suggestions. This work was partially supported by FCT under contract (PTDC/EIA-EIA/103103/2008) and by the Tracing Networks research programme funded by the Leverhulme Trust.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. 21(4), 181–185 (1985)
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2(3), 117–126 (1987)
3. Alur, R., Etessami, K., Torre, S.L., Peled, D.: Parametric temporal logic for "model measuring". ACM Trans. Comput. Log. 2(3), 388–407 (2001)
4. Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press (2008)
5. Betin-Can, A., Bultan, T., Fu, X.: Design for verification for asynchronously communicating web services. In: Ellis, Hagino (eds.) [9], pp. 750–759
6. Beyer, D., Chakrabarti, A., Henzinger, T.A.: Web service interfaces. In: Ellis, Hagino (eds.) [9], pp. 148–159
7. 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)
8. 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)
9. Ellis, A., Hagino, T. (eds.): Proceedings of the 14th International Conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14. ACM (2005)
10. Fiadeiro, J.L., Lopes, A.: An Interface Theory for Service-Oriented Design. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 18–33. Springer, Heidelberg (2011)
11. 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)
12. Henzinger, T.A.: Sooner is safer than later. Inf. Process. Lett. 43(3), 135–141 (1992)
13. Kazhamiakin, R., Pistore, M., Santuari, L.: Analysis of communication models in web service compositions. In: Carr, L., Roure, D.D., Iyengar, A., Goble, C.A., Dahlin, M. (eds.) WWW, pp. 267–276. ACM (2006)
14. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2(4), 255–299 (1990)
15. Kupferman, O., Piterman, N., Vardi, M.Y.: From liveness to promptness. Formal Methods in System Design 34(2), 83–103 (2009)
16. Martens, A.: Process oriented discovery of business partners. In: Chen, C.-S., Filipe, J., Seruca, I., Cordeiro, J. (eds.) ICEIS (3), pp. 57–64 (2005)
17. OSOA. Service component architecture: Building systems using a service oriented architecture (2005), White paper, http://www.osoa.org
18. Ouaknine, J., Worrell, J.: Safety Metric Temporal Logic Is Fully Decidable. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 411–425. Springer, Heidelberg (2006)
19. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Asp. Comput. 6(5), 495–512 (1994)
20. TC, O.W.: Web services business process execution language. Version 2.0. Technical report, OASIS (2007)
21. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. 115(1), 1–37 (1994)

Stable Availability under Denial of Service Attacks through Formal Patterns*

Jonas Eckhardt[1,2,3], Tobias Mühlbauer[1,2,3], Musab AlTurki[4], José Meseguer[4], and Martin Wirsing[1,5]

[1] Ludwig Maximilian University of Munich
[2] Technical University of Munich
[3] University of Augsburg
[4] University of Illinois at Urbana-Champaign
[5] IMDEA Software

Abstract. Availability is an important security property for Internet services and a key ingredient of most service level agreements. It can be compromised by distributed Denial of Service (DoS) attacks. In this work we propose a formal pattern-based approach to study defense mechanisms against DoS attacks. We enhance pattern descriptions with formal models that allow the designer to give guarantees on the behavior of the proposed solution. The underlying executable specification formalism we use is the rewriting logic language Maude and its real-time and probabilistic extensions. We introduce the notion of stable availability, which means that with very high probability service quality remains very close to a threshold, regardless of how bad the DoS attack can get. Then we present two formal patterns which can serve as defenses against DoS attacks: the Adaptive Selective Verification (ASV) pattern, which enhances a communication protocol with a defense mechanism, and the Server Replicator (SR) pattern, which provisions additional resources on demand. However, ASV achieves availability without stability, and SR cannot achieve stable availability at a reasonable cost. As a main result we show, by statistical model checking with the PVeStA tool, that the composition of both patterns yields a new improved pattern which guarantees stable availability at a reasonable cost.

Keywords: formal patterns, meta-object pattern, rewriting logic, availability, denial of service, statistical model checking, cloud computing.

1 Introduction

On December 8, 2010 at 07:53 AM EDT, MasterCard issued a statement that "MasterCard is experiencing heavy traffic on its external corporate website [...]. There is no impact whatsoever on our cardholders ability to use their cards for secure transactions" [19]. In fact, by that time, a distributed Denial of Service

* This work has been partially sponsored by the Software Engineering Elite Graduate Program, the EU-funded projects FP7-257414 ASCENS and FP7-256980 NESSoS, and AFOSR Grant FA8750-11-2-0084. The fourth author was also partially supported by the "Programa de Apoyo a la Investigación y Desarrollo" (PAID-02-11) of the Universitat Politècnica de València.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 78–93, 2012.

attack (DoS) brought the website down and made their web presence unavailable for most customers for several hours. Availability is an important security property for Internet services and a key ingredient of most service level agreements.

DoS defense mechanisms help maintaining availability; nevertheless even when equipped with defense mechanisms, systems will typically show performance degradation. Thus, one of the goals of security measures is to achieve stable availability, which means that with very high probability service quality remains very close to a constant quantity, which does not change over time, regardless of how bad the DoS attack can get. Cloud Computing, by offering the possibility of dynamic resource allocation, can be used to leverage stable availability when combined with DoS defense mechanisms. Service-oriented systems such as the MasterCard service are distributed systems operating in a dynamically changing environment. They need to cope with changing numbers of user demands and with hostile attacks. To be used/operated safely, services have to satisfy functional as well as non-functional requirements and it is not a priori clear what is the best realization of a service in each particular situation. Model-driven approaches to service development offer the possibility of tackling these issues at a high level of abstraction during early stages of system analysis and design. In particular, design patterns have been successfully used for improving programming solutions in several domains, including object-orientation [13], service-oriented computing [17,12] and security [25]. Patterns are general, reusable solutions to commonly occurring problems in software design; they clearly define the programming context, the problem and the advantages and disadvantages of design solutions (see e.g., [13,25]).

In this work, we introduce formal patterns which, in addition to "normal" patterns, come with formal guarantees and enable automated pattern composition, often resulting in semi-automatic construction of new models with improved properties. We use this pattern-based approach to study defense mechanism against DoS attacks in a model-based setting. We present two formal patterns which can serve as defenses against DoS attacks: the Adaptive Selective Verification (ASV) [15] pattern defending against DoS attacks, and the Server Replicator (SR) pattern in a cloud setting. As underlying executable specification formalism we use the rewriting logic language Maude and its real-time and probabilistic extensions. The ASV protocol is a well-known defense against DoS attacks in the typical situation that clients and attackers use a shared channel where neither the attacker nor the client have full control over the communication channel [15]. The ASV protocol adapts to increasingly severe DoS attacks and provides improved availability. However, it cannot provide stable availability. By replicating servers one can dynamically provision more resources to adapt to high demand situations and achieve stable availability; but the cost of provisioned servers drastically increases in a DoS attack situation. These two patterns are modeled in Maude and then formally composed to obtain the new improved ASV$^+$SR pattern. As a main result we show, by analyzing the quantitative properties of ASV$^+$SR with the statistical model checker PVeStA, that ASV$^+$SR guarantees stable availability at a reasonable cost.

Outline. The paper is structured as follows: Sect. 2 introduces the notion of stable availability and gives a short account of the prerequisites on rewriting logic, Maude, and the statistical model checking of quantitative properties with the PVeStA tool in Maude. In Sect. 3 we present the concept of formal patterns and give three examples: (i) the general meta-object pattern (Sect. 3.1), (ii) the ASV pattern (Sect. 3.2), and (iii) the SR pattern (Sect. 3.3). In Sect. 4 we present the ASV$^+$SR composition pattern and validate the properties of the composed system using the PVeStA tool. We conclude by discussing related work, summarizing our results and sketching further work.

2 Prerequisites

2.1 Rewriting Logic and Maude

Rewriting logic [21] is a simple computational logic to specify concurrent and object-oriented systems as *rewrite theories*, that is, as triples (Σ, E, R), where (Σ, E) is an *order-sorted equational theory* with syntax and type structure specified by the signature Σ, and with (possibly conditional) Σ-equations E; and where R is a set of (possibly conditional) *rewrite rules* of the form $t \to t'$ *if cond*, with t, t' Σ-terms, and *cond* the rule's condition.

The Maude system [9] executes rewrite theories, with a self-explanatory typewriter syntax almost isomorphic to the mathematical syntax. The key concept in Maude is that of a module. An object-oriented module defines a class named K and attributes $a_1 \ldots, a_n$. An object o in a given state can be represented as a term of the form $\langle o : K \mid a_1 : v_1, \ldots, a_n : v_n \rangle$ where $v_1 \ldots, v_n$ are the corresponding values stored in those attributes. A *message* addressed to object o with contents d can be represented as a term $(o \leftarrow d)$; and all messages in a system are then terms of sort *Message*. The distributed systems we consider in this paper are systems, made up of objects that communicate with each other by asynchronous message passing. The *distributed state* of such a system is a *multiset* or "soup" of objects and messages, called a *configuration*. Mathematically, this is specified by declaring a sort *Configuration* with subsort inclusions $Object, Message < Configuration$, and an associative and commutative multiset union operator with empty syntax: $\_\_ : Configuration\ Configuration \longrightarrow Configuration$ and with identity element *null*.

For example, a simple client class may have name *Client*; a simple server class may have name *Server* and an attribute *bf* for storing the received messages in a buffer. In a simple request-response message exchange pattern (cf. [27]) a client c sends request packets ($req(c)$) to the server. In response, the server sends response packets (ack) back to the client. The following term defines a configuration containing one server object s with a request from *c1* in the buffer, two client objects and one message addressed to *c1*.

$$\langle s : Server \mid bf : req(c1) \rangle \ \langle c1 : Client \mid \rangle \ \langle c2 : Client \mid \rangle \ (c1 \leftarrow ack)$$

The following rewrite rule defines the reaction of any server object s upon receipt of a request $(s \leftarrow req(c))$ from any client c.

$$\textbf{rl}\ (s \leftarrow req(c))\ \langle s : Server \mid bf : b \rangle\ \rightarrow \langle s : Server \mid bf : b\ req(c) \rangle\ (c \leftarrow ack)\ .$$

The server adds $req(c)$ to the buffer and sends an acknowledgement $(c \leftarrow ack)$ back to the client c. Although not illustrated by the rule above, upon receiving message an object can send several messages to other objects, and can create new objects.

Rewriting logic can naturally model concurrent systems, which can be both *real-time* and *probabilistic*. Real-Time systems are supported by rewrite theories (Σ, E, R) whose underlying equational theory (Σ, E) includes among its types an algebraic data type *Time* representing time instants (which may be either discrete or continuous), and whose global states are pairs of the form (t, r), with t a term representing a "discrete" state, and r a time value of sort *Time* representing the global clock. The rewrite rules in R can then be either *instantaneous* rules, which do not change the global clock, or *tick* rules, which advance the global time (see [24]). Probabilistic concurrent systems, which may also be real-time systems, are modeled by *probabilistic rewrite rules* of the form

$$l : t(\boldsymbol{x}) \rightarrow t'(\boldsymbol{x}, \boldsymbol{y})\ \ \textit{if}\ \ cond(\boldsymbol{x})\ \ \textit{with probability}\ \ \boldsymbol{y} := \pi_l(\boldsymbol{x})$$

where the righthand side term t' has new variables $\boldsymbol{y}$ disjoint from the variables $\boldsymbol{x}$ appearing in t which make the application of the rule non-deterministic. The probabilistic nature of the rule is expressed by the probability distribution $\pi_l(\boldsymbol{x})$ with which values for the extra variables $\boldsymbol{y}$ are chosen; where $\pi_l(\boldsymbol{x})$ is in general not fixed, but parametric on the righthand side variables $\boldsymbol{x}$. In this paper, we use the PMaude [4] notation for probabilistic rewrite rules.

A *parameterized module* $M[X :: P]$ has a formal parameter X satisfying a parameter theory P; M can be instantiated by another module Q via a theory interpretation $V : P \longrightarrow Q$, called a *view*, with the usual pushout semantics (see [9]). We denote the resulting module by $M[V]$ or shorter by $M[Q]$ if V is clear from the context.

2.2 Statistical Model Checking of Quantitative Properties

Temporal logic properties of a probabilistic system can be model checked either by exact model checking algorithms or, in an approximate but more scalable way, by *statistical model checking* (see, e.g., [26,29,4]). The idea of statistical model checking is to verify the satisfaction of a temporal logic property by statistical methods up to a user-specified level of statistical confidence. For this, a large enough number of Monte-Carlo simulations of the system are performed, and the formula is evaluated on each of the simulations.

Current statistical model checking algorithms assume that the system is purely probabilistic, i.e., that there is no nondeterminism in the choice of transitions. Using the methodology presented in [4] and further extended in this work to the case of reflective "Russian dolls" architectures, a wide class of object-oriented probabilistic real-time distributed systems can be expressed as purely probabilistic systems. In particular, all the distributed systems considered in this paper fall within this broad class.

To analyze the behavior of systems with respect to quantitative properties related to performance and QoS, a *quantitative* temporal logic, where the result of evaluating a formula is not a Boolean true/false value, but a real number, can be used. For this purposes we use the QuaTEx quantitative temporal logic [4], and the PVeStA [6] parallelization of its associated VeStA tool and model checking algorithm [4]. In Sect. 4.2 we will present several QuaTEx expressions formalizing crucial quantitative properties related to DoS protection and will model check them in PVeStA. We refer the reader to [4] for a detailed description of QuaTEx expressions and their model checking algorithm. In this paper, we will compute the expected value of a path expression based on definitions of the form $F(t) =$ **if** $time() > t$ **then** EXP **else** $\bigcirc(F(t))$, where $\bigcirc$ is the next operator, $time()$ is a state function returning the global time, and EXP is a real-valued state function.

2.3 Stable Availability

Availability is a key security property by which a system remains available to its users under some conditions. This property can be compromised by a DoS attack, which may render a system unavailable in practice. What all DoS defense mechanisms have in common is the goal of protecting a system's availability properties in the face of a DoS attack. But availability properties are *quantitative* properties: some DoS defense mechanisms may provide better QoS properties and therefore better availability properties than others. In fact, even when protected against DoS, performance degradation will typically be experienced in some aspects of system behavior such as, for example, the average Time To Service (TTS) experienced by clients, the success ratio with which clients manage to communicate with their server, or the average bandwidth (or some other cost measure) that a client needs to spend to successfully communicate with its server. Obviously, an ideal DoS protection scheme is one that renders the system to a large extent *impervious* to the DoS attack, no matter how bad the attack can get.[1] That is, up to some acceptable and constant performance degradation, the system behaves in a "business as usual" manner: as if no attack had taken place, even when in fact the attack worsens over time. We call this property *stable availability*. As we shall show in Sect. 4, stable availability can be achieved in some cases by using an appropriate meta-object architecture for DoS protection.

More precisely, the stable availability of a system assumes a shared channel [14], where DoS attackers can at most monopolize a maximum percentage of the overall bandwidth. Under these circumstances, stable availability is formulated as a requirement parameterized by explicitly specified and *quantifiable* availability properties such as, for example, TTS, success ratio, average bandwidth, and so on. The system is then said to be *stably available* with respect to the specified

[1] In the shared channel model of [14], attackers can have a potentially very large but not absolute share of the overall bandwidth, so that honest users will still have some bandwidth available. This is a realistic assumption in most situations, and a key difference between DoS attackers and Dolev-Yao attackers, who, having full control of the channel, can always destroy *all* honest user messages.

quantities if and only if, with very high probability, each such quantity q remains very close (up to fixed bounds ε) to a *threshold* θ ($\mid q - \varepsilon \mid < \theta$), which does not change over time, regardless of how bad the DoS attack can get within the bounds allowed by the shared channel assumption.

3 Formal Patterns

Pattern-based approaches have been successfully introduced to help developers choose appropriate design and programming solutions [13]. However, these informal patterns typically offer limited help for assessing the required functional and non-functional properties. This is particularly important in the case of distributed systems, which are notoriously hard to build, test, and verify. To ameliorate this problem we are proposing to enhance pattern descriptions with executable specifications that can support the mathematical analysis of qualitative and quantitative properties; thus allowing the designer to give guarantees on the behavior of the proposed solution.

A formal pattern Pat is structured in the usual way (cf. e.g. [25,12]) in context, problem, solution, advantages and shortcomings (and other features such as forces, related patterns which we mostly omit here for simplicity); but instead of using UML or Java we describe the solution formally as a parameterized module $M[S]$ in Maude (with parameter theory S) and draw many of the advantages and shortcomings of a pattern from formal analyses. Moreover, the context typically describes also the assumptions of the parameter theory S.

Pattern composition $Pat + Pat'$ of two patterns Pat and Pat' formalized as parameterized Maude modules $P[S]$ and $P'[S']$ can be achieved by an appropriate "parameterized view" (see [9]) connecting both patterns. For example, we may instantiate S' to $P[S]$, yielding the composed pattern $P'[P[S]]$. The problem statement and context of $Pat + Pat'$ can then be systematically derived from those of Pat and Pat'.

In the following we present several formal patterns which can be very useful to make distributed systems adaptable to changing and potentially hostile environments, and show how to design and analyze such systems in a modular and predictable way.

3.1 The Meta-object Pattern

Concurrency is not the only challenge for distributed systems: *adaptation* is just as challenging, since many distributed systems need to function in highly unpredictable and potentially hostile environments such as the Internet, and need to satisfy safety, real-time and Quality of Service (QoS) requirements which are essential for their proper behavior. To meet these adaptation challenges and the associated requirements, a modular approach based on *meta-objects* can be extremely useful. A meta-object pattern MO is defined as follows:

Context. A concurrent and distributed object-based system.

Problem. How can the communication behavior of one or several objects be dynamically *mediated/adapted/controled* for some specific purposes?

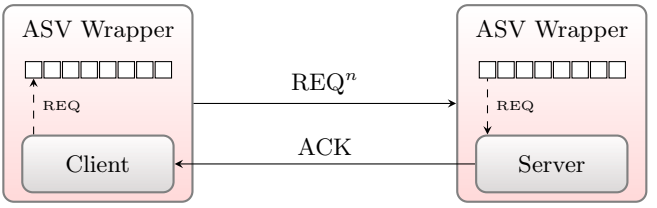


Fig. 1. Application of the ASV meta-object on a client-server request-response service

Solution. A meta-object is an object which dynamically *mediates/adapts/controls* the communication behavior of one or several objects under it. In rewriting logic, a meta-object can be specified as an object of the form $\langle o : K \mid conf : c, a_1 : v_1, \ldots, a_n : v_n \rangle$, where c is a term of sort *Configuration*, and all other $v_1 \ldots, v_n$ are not configuration terms. The configuration c contains the object or objects that the meta-object o controls. Thus the parameterized module $MO[X]$ introduces the meta-object constructor; the parameter X specifies the sorts $s_1, \ldots, s_n$ and attributes $a_1, \ldots, a_n$ of the controlled system.

Advantages and Shortcomings. MO defines a general control and wrapper architecture; but may add communication indirection and the requirement for language specific object visibility.

There are many different MO patterns: If c contains a single object, the meta-object o is sometimes called an *onion-skin* meta-object [2], because o itself could be wrapped inside another meta-object, and so on, like the skin layers in an onion. More generally, c may not only contain several objects $o_1 \ldots, o_m$ inside: it may also be the case that some of these o_i are themselves meta-objects that contain other objects, which may again be meta-objects, and so on. That is, the more general reflective meta-object architectures are so-called "Russian dolls" architectures [22], because each meta-object can be viewed as a Russian doll which contains other dolls inside, which again may contain other dolls, and so on.

In the following we will present meta-object patterns that illustrate both the onion-skin case, and the general Russian dolls case.

3.2 The ASV DoS Protection Meta-object Pattern

The ASV protocol [15] is a cost-based, DoS-resistant protocol where bandwidth is used as currency by a server to discriminate between good and malicious users; that is, honest clients spend more bandwidth by replicating their messages.

Context. Client-server request-reply system under DoS attack, shared channel attacker model [14].

Problem. How can the system be protected against DoS attacks?

Solution. Informally described, the server and the clients are wrapped by meta-objects with the following key features: The client wrappers attempt to adapt to the current level of attack by exponentially replicating the client requests up to a fixed bound. The server wrapper adapts to the level of the attack by dropping randomly packets, with a higher probability as the attack becomes more severe. Only the remaining requests are processed by the server.

Fig. 1 illustrates the ASV meta-object pattern.

A first modularized formalization of the ASV protocol was given by AlTurki in [5]. In this work we extend this specification by making its modularization more explicit using parametrized modules. The modularized ASV meta-object specification ($ASV[S]$) is parametric in the client-server system S. In particular, we assume that S indicates the maximal load *maxLoad* per server. Clients have a time-out window (which is set to the expected worst case round-trip delay between the client and the server) and a replication threshold, i.e. the maximum number of times a client tries to send requests to the server before it gives up.

We present only the behavior of the server wrapper in a little more detail. The wrapper counts the incoming requests and places them in a buffer *buf*. If the buffer length of the servers exceeds *maxLoad*, a coin is tossed to decide whether an incoming message should be dropped or not, i.e., it is randomly decided according to a Bernoulli distribution *Ber* with success probability $\mathit{floor}(\mathit{maxLoad})/(\mathit{cnt} + 1.0)$. If the message is not dropped, a position of *buf* is randomly chosen with uniform distribution *Uni* and the new message is stored at this position (replacing another message).

$$
\begin{aligned}
&\textbf{crl}\ (s \leftarrow c)\ \langle s : \mathit{asvServer} \mid \mathit{count} : \mathit{cnt}, \mathit{buf} : L\rangle\ \rightarrow \\
&\ \textbf{if}\ (y_2)\ \textbf{then}\ \ \langle s : \mathit{asvServer} \mid \mathit{count} : \mathit{cnt} + 1.0, \mathit{buf} : L[y_1] := c\rangle \\
&\ \textbf{else}\ \langle s : \mathit{asvServer} \mid \mathit{count} : \mathit{cnt} + 1.0, \mathit{buf} : L\rangle\ \textbf{fi} \\
&\textbf{if}\ \mathit{float}(L.\mathit{size}) \geq \mathit{floor}(\mathit{maxLoad}) \\
&\textbf{with probability}\ y_1 := \mathit{Uni}(L.\mathit{size}) \\
&\ \textbf{and}\ y_2 := \mathit{Ber}(\mathit{floor}\,(\mathit{maxLoad})\,/(\mathit{cnt} + 1.0)).
\end{aligned}
$$

In addition, the server wrapper periodically empties its buffer and sends the contents to the wrapped server. Answers of the server are forwarded to the client.

Advantages & Shortcomings. The ASV protocol has remarkably good properties, such as closely approximating *omniscience* [15]: although only local knowledge is used by each protocol participant, ASV's emergent behavior closely approximates the behavior of an idealized DoS defense protocol in which all relevant parameters describing the state of the attack are instantaneously known to all participants. However, it cannot provide stable availability [11,23].

3.3 The Server Replicator Meta-object

In high-demand situations, Cloud-based services can benefit from the scalability of the Cloud, i.e., from the dynamic allocation of resources. The **S**erver **R**eplicator meta-object (SR) is a simple pattern that adapts to high-demand situations by leveraging the scalability of the Cloud [11,23].

Context. Client-server request-reply system; possibility of provisioning additional resources.

Problem. How can the system adapt to an increasing amount of requests, e.g., caused by a DoS attack?

Solution. The SR wraps instances of servers that provide a service, dynamically provisions new such instances to adapt to an increasing load, and distributes incoming requests among them.

The meta-object SR ($SR[S]$) is parametric in the client-server system S, whose servers (of class ($Server$)) it creates instances of. In order to be replicable, the servers in S need to fulfill a theory which specifies how a server instance is created ($replicate$) and initialized ($init$); and how many requests it can handle within a specific timeframe ($maxLoadPerServer$). Additional parameters in S specify a replication strategy which determines the *overloading factor* which must be exceeded before a new server is provisioned.

SR performs the following tasks:

Provisioning New Instances of the Server. SR periodically evaluates its replication strategy and, if necessary, spawns a new server instance. The behavior of spawning a new server is described by the rewrite rule

$$\begin{aligned}&\textbf{crl}\ (sr \leftarrow spawnServer)\ \langle sr : ServerReplicator \mid server\text{-}list : SL, config : NG\ C\rangle\\ &\quad\rightarrow \langle sr : ServerReplicator \mid server\text{-}list : (sa; SL),\\ &\qquad config : (NG.next)\ C\ replicate(sa)\ init(sa)\rangle\\ &\textbf{if}\ sa := NG.new\ .\end{aligned}$$

Removing Instances of the Server. SR winds down the number of replicated servers when the load decreases. We do not model this behavior. One solution would be to synchronize the communication between SR and a server instance by using a buffer. SR sets a server instance it wants to remove as inactive and no longer forwards requests to it. When an inactive server has processed all requests in its buffer, it removes itself from the configuration.

Distribution of Incoming Messages. SR randomly distributes incoming requests among its servers in a uniform way using the rule

$$\begin{aligned}&\textbf{rl}\ (sr \leftarrow CO)\ \langle sr : ServerReplicator \mid server\text{-}list : SL, config : C\rangle\ \rightarrow\\ &\langle sr : ServerReplicator \mid server\text{-}list : SL, config : (y_1 \leftarrow CO)\ C\rangle\\ &\textbf{with probability}\ y_1 := Random(SL)\ .\end{aligned}$$

where *Random* randomly chooses a server from a list of servers.

Forwarding Messages to the Outside. Additionally, SR specifies rules to forward messages that address client objects located outside its boundary.

Advantages & Shortcomings. SR can provide stable availability. However, the cost of provisioning servers drastically increases in high-demand situations.

4 Stable Availability under Denial of Service Attacks through Formal Patterns

How can meta-object patterns be used to make a Cloud-based client-server request-response service resilient to DoS attacks with minimum performance degradation, that is, achieving in fact stable availability at reasonable cost?

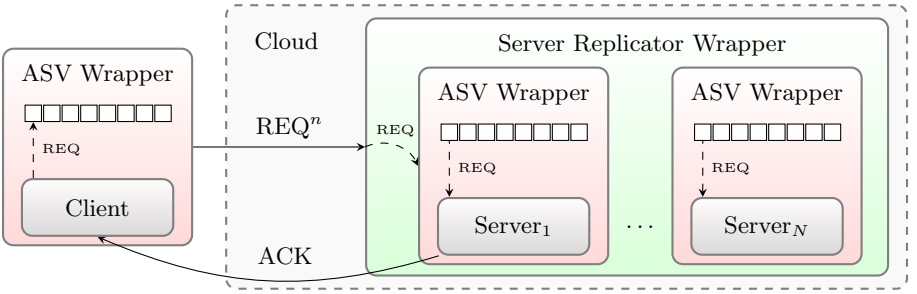


Fig. 2. Application of the ASV$^+$SR meta-object composition on a Cloud-based client-server request-response service

We propose to investigate this question by composing a client-server system S with appropriate meta-object patterns.

4.1 ASV$^+$SR Meta-object Composition Pattern

Combining the ASV and SR meta-object patterns into ASV$^+$SR enables us to overcome their respective shortcomings while keeping their advantages.

Context. Client-server request-reply system under DoS attack, shared channel attacker model [14]; possibility of provisioning additional resources.

Problem. How can the system be protected against the DoS attack and provide stable availability at reasonable cost?

Solution. The application of the meta-object composition on S, $SR[ASV[S], \rho]$, (where ρ maps the formal parameter ($Server$) to ($asvServer$) and ($maxloadServer$) to ($maxLoad$)) protects the service against DoS attacks in two dimensions of adaption: (i) the ASV mechanism; and (ii) the SR replication mechanism. Fig. 2 gives an overview of the composition.

We define the factor k that proportionally adjusts the degree of ASV protection in the meta-object composition, i.e., k reflects how much the ASV mechanism is used compared to the SR replication mechanism. An overloading factor of $k = 1$ means that the ASV mechanism remains nearly unused, while an overloading factor of $k = \infty$ means that the replication mechanism is unused. Thus, we propose an overloading factor of $1 < k < \infty$.

The replication strategy for computing the number of server replicas γ is defined as

$$\gamma(m, t) = \max\left(1, \frac{m}{maxLoadPerServer(t) \cdot k}\right)$$

where m denotes the number of messages that have been received by the SR up to time t; and $maxLoadPerServer(t)$ is defined as

$$maxLoadPerServer(t) = \left\lfloor \frac{t}{T} \right\rfloor \cdot maxLoad_S$$

where T is the ASV server timeout period and $maxLoad_S$ denotes the buffer size of the ASV server.

Advantages & Shortcomings. We will show that the ASV$^+$SR composition provides stable availability under DoS attacks at the cost of provisioning a predictable amount of instantiated servers given by the overload factor.

4.2 Statistical Model Checking Analysis

We use the Maude-based specification of the ASV$^+$SR meta-object pattern with a client-server system to perform parallelized statistical quantitative model checking on 20 to 40 cluster nodes using PVeStA. The expected values of the following QuaTEx path expressions were computed with a 99% confidence interval of size at most 0.01:

Client Success Ratio. The client success ratio defines the ratio of clients that receive an acknowledgement from the server.

$$\begin{aligned} \mathit{successRatio}(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{countSuccessful}()/\mathit{countClients}() \\ \textbf{else } \bigcirc\,(\mathit{successRatio}(t)) \end{aligned}$$

where *countClients*() and *countSuccessful*() respectively count the total number of clients, and the number of clients with *"connected"* status.

Average TTS. The average TTS is the average time it takes for a successful client to receive an acknowledgement from the server.

$$\begin{aligned} \mathit{avgTTS}(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{sumTTS}()/\mathit{countSuccessful}() \\ \textbf{else } \bigcirc\,(\mathit{avgTTS}(t)) \end{aligned}$$

where *sumTTS*() is the sum of the TTS values of all successful clients.

Number of Servers. The number of servers represents the number of ASV servers that are spawned by the SR meta-object.

$$\begin{aligned} \mathit{servers}(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{countServers}() \\ \textbf{else } \bigcirc\,(\mathit{servers}(t)) \end{aligned}$$

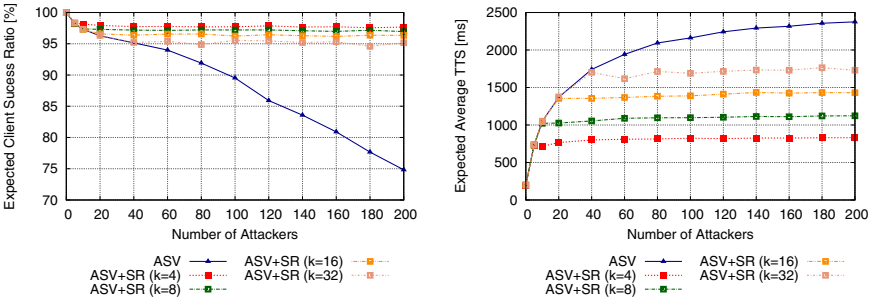
where *countServers*() is the number of replicated servers.

For statistical model checking purposes we set the parameters of the ASV and SR meta-objects as follows:

ASV. The mean server processing rate is set to 600 packets per second, the timeout window of the clients to 0.4 seconds, the retrial span of the clients to 7, and the client arrival rate to 0.08.

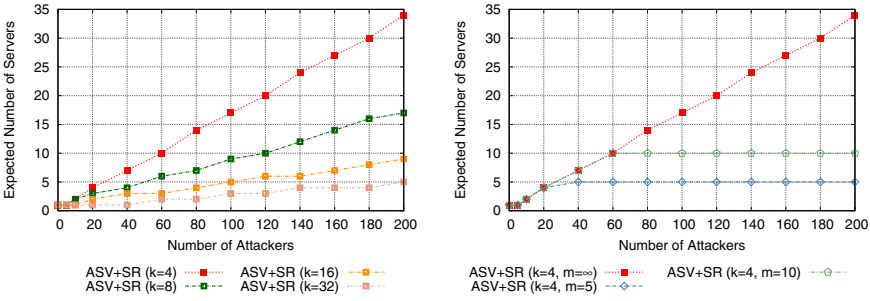
SR. The check period is set to 0.01 seconds and we vary the overloading factor k (4, 8, 16, 32). Forward and replication delays are not considered in our experiments.

The properties are checked for a varying number of attackers (1 to 200). Each attacker issues 400 fake requests per second. It is of note that 1.5 attackers already overwhelm a single server. The values of the ASV and attack parameters correspond to the values chosen in [7,15]. Additionally, an initial generation delay of 0.05 seconds is introduced and the duration of a simulation is set to 30 seconds.



(a) Client success ratio (b) Average TTS

Fig. 3. Performance of the ASV$^+$SR protocol with a varying load factor k and no resource bounds

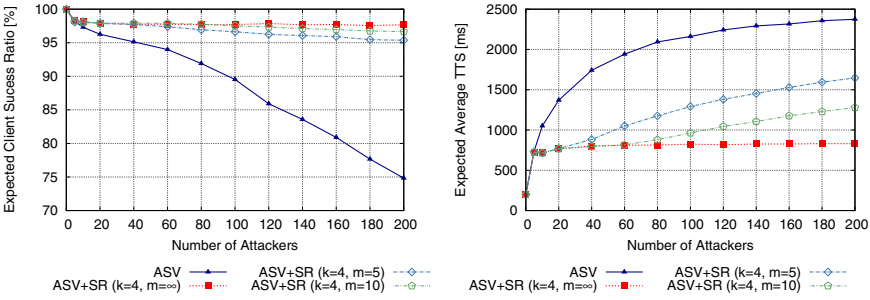


(a) Number of servers (unlimited) (b) Number of servers (limited)

Fig. 4. Expected number of servers using the ASV$^+$SR protocol

In the following, we will consider two general cases in which the SR can provision: (i) an unlimited number of servers, and (ii) servers up to a limit m of 5 or 10 servers, because, due to economical and physical restrictions, resources are limited. The results in (i) will indicate how many servers are needed to provide stable service guarantees, while the results in (ii) will indicate what service guarantees can still be given with limited resources.

Unlimited Resources. Fig. 3 shows the model checking results for a varying overloading factor k with no resource limits. As indicated by Fig. 3(a), ASV$^+$SR can sustain the expected client success ratio at a certain percentage. Even for an overloading factor of $k = 32$, a success ratio around 95% can be achieved. Compared to an overloading factor of $k = 4$, a 7-fold decrease in provisioned servers is observed (Fig. 4(a)), achieving a stable success ratio of only around 3% less. Fig. 3(b) shows that the same is true for the average TTS. ASV$^+$SR outperforms the ASV protocol, and furthermore achieves stable availability, for all performance indicators. However, this comes at the cost of provisioning new servers. Fig. 4(a) shows how many servers are provisioned. The results indicate that the factor k defines a trade-off between the cost and the performance of stable availability. SR by itself ($k = 1$) with unlimited resources (not shown in the figures) would



(a) Client success ratio (b) Average TTS

Fig. 5. Performance of the ASV$^+$SR protocol with a load factor of $k = 4$ and limited resources

provide stable availability at a level as if no attack has happened, but would provision 134 servers for 200 attackers. Note that fluctuations in the results, e.g., the average TTS in case of 60 attackers being lower than the average TTS in case of 40 attackers, are due to the provisioning of a discrete number of servers.

Limited Resources. Fig. 5 shows the model checking results for an overloading factor of $k = 4$ and a limit m of either 5 or 10 servers that the SR meta-object can provision. As indicated by Fig. 5(a), the success ratio can still be kept at a high level under the assumption of limited resources. In fact, the protocol behaves just as in the case of unlimited resources up to the point where more servers than the limit would be needed to keep the success ratio stable. After that point, the protocol behaves like the original ASV protocol (but with the equivalent of a more powerful server) and the success ratio decreases. Nevertheless, it decreases more slowly since now 5, respectively 10, servers handle the incoming requests compared to the single server in the ASV case. Fig. 5(b) shows that the average TTS behaves in a way similar to that of the success ratio. We only checked these properties for an overloading factor of $k = 4$; for higher values of k, the attack level at which stable availability is lost is higher and the rate at which the quality subsequently decreases differs by a constant factor.

5 Related Work and Concluding Remarks

Here we discuss related work on defenses against DoS attacks and their formal analysis. Related work on modular meta-object architectures for distributed systems, and on statistical model checking and quantitative properties has been respectively discussed in Sects. 2.1 and 2.2.

There exist several approaches to formal patterns (see e.g. [10]); ours is different by focusing on executable specifications, quantitative analysis, and the combination of formal and informal aspects. The standard book on security patterns [25] does not discuss DoS defenses, although some of its patterns (such as reflection, replication and filtering) can be related to our patterns.

Defenses against DoS attacks use various mechanisms. An important class of defenses use *currency-based mechanisms*, where a server under attack demands payment from clients in some appropriate "currency" such as actual *money*, *CPU cycles* (e.g., by solving a puzzle), or, as in the case of ASV, *bandwidth*. The earliest bandwidth-based defense proposed was Selective Verification (SV) [14]. Adaptive bandwidth-based defenses include both ASV [15], and the auction-based approach in [28].

Regarding formalizations and analyses of DoS resistance of protocols, a general cost-based framework was proposed in [20]; an information flow characterization of DoS-resistance was presented in the cost-based framework of [16]; and [1] used observation equivalence and a cost-based framework to analyze the availability properties of the JFK protocol. Other works on formal analysis of availability properties use branching-time logics [30,18]. Our own work is part of a recent approach to the formal analysis of DoS resistance using statistical model checking. The first paper in this direction used probabilistic rewrite theories to analyze the DoS-resistance of the SV mechanism when applied to the handshake steps of TCP [3]. ASV itself, applied to client-server systems, was formally specified in rewriting logic and was analyzed this way in [7]. The formalization of ASV in rewriting logic as a meta-object was first presented in [5]. Likewise, cookies have been formalized in rewriting logic as a meta-object for DoS defense in [8].

In this paper we have presented a formal pattern-based approach to the design and mathematical analysis of security mechanisms of Cloud services. We have shown that formal patterns can help deal with security issues and that formal analysis can help evaluate patterns in various contexts. In particular, we have specified dynamic server replication (SR) and the ASV protocol as formal patterns in the executable rewriting logic language Maude. By formally composing the two patterns we have obtained the new pattern ASV$^+$SR. We have analyzed properties of the ASV$^+$SR pattern using the statistical model checker PVeStA, and were able to show as our main result that, unlike the two original patterns, ASV$^+$SR achieves stable availability in presence of a large number of attackers at reasonable cost, which can be predictably controlled by the choice of the overloading parameter.

Our current results rely on two simplifications: The client-server communication consists of a stateless request-reply interaction and the replication of servers is only able to add but not to delete servers. As next steps, we plan to refine the patterns to cope with the winding-down of resources at the end of a DoS attack and with more complex client-server interactions where the server has to preserve state. Moreover, in this paper we have only studied quantitative properties of the patterns; it would be very interesting and useful to analyze also qualitative properties. In [8] it is shown that adding cookies to a client-server system preserves all safety properties. We conjecture that the same holds for the ASV and ASV$^+$SR protocols. Finally, we plan to continue with our pattern-based approach and to build a collection of formal patterns for security mechanisms.

References

1. Abadi, M., Blanchet, B., Fournet, C.: Just Fast Keying in the Pi Calculus. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 340–354. Springer, Heidelberg (2004)
2. Agha, G., Frolund, S., Panwar, R., Sturman, D.: A linguistic framework for dynamic composition of dependability protocols. IFIP, pp. 345–363 (1993)
3. Agha, G., Gunter, C., Greenwald, M., Khanna, S., Meseguer, J., Sen, K., Thati, P.: Formal modeling and analysis of DoS using probabilistic rewrite theories. In: FCS (2005)
4. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. ENTCS 153(2), 213–239 (2006)
5. AlTurki, M.: Rewriting-based formal modeling, analysis and implementation of real-time distributed services. PhD thesis, University of Illinois (2011)
6. AlTurki, M., Meseguer, J.: PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011)
7. AlTurki, M., Meseguer, J., Gunter, C.: Probabilistic modeling and analysis of DoS protection for the ASV protocol. ENTCS 234, 3–18 (2009)
8. Chadha, R., Gunter, C.A., Meseguer, J., Shankesi, R., Viswanathan, M.: Modular Preservation of Safety Properties by Cookie-Based DoS-Protection Wrappers. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 39–58. Springer, Heidelberg (2008)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
10. Dong, J., Alencar, P.S.C., Cowan, D.D., Yang, S.: Composing pattern-based components and verifying correctness. JSS 80, 1755–1769 (2007)
11. Eckhardt, J.: A Formal Analysis of Security Properties in Cloud Computing. Master's thesis, LMU Munich (2011)
12. Erl, T.: SOA Design Patterns. Prentice Hall (2008)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
14. Gunter, C., Khanna, S., Tan, K., Venkatesh, S.: DoS Protection for Reliably Authenticated Broadcast. In: NDSS (2004)
15. Khanna, S., Venkatesh, S., Fatemieh, O., Khan, F., Gunter, C.: Adaptive Selective Verification. In: IEEE INFOCOM, pp. 529–537 (2008)
16. Lafrance, S., Mullins, J.: An Information Flow Method to Detect Denial of Service Vulnerabilities. JUCS 9(11), 1350–1369 (2003)
17. Wirsing, M., et al.: Sensoria Patterns: Augmenting Service Engineering. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 170–190. Springer, Heidelberg (2008)
18. Mahimkar, A., Shmatikov, V.: Game-based Analysis of Denial-of-Service Prevention Protocols. In: IEEE CSFW, pp. 287–301 (2005)
19. MasterCard. MasterCard Statement (September 2011), http://www.businesswire.com/news/home/20101208005866/en/MasterCard-Statement
20. Meadows, C.: A Formal Framework and Evaluation Method for Network Denial of Service. In: IEEE CSFW (1999)

21. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. TCS 96(1), 73–155 (1992)
22. Meseguer, J., Talcott, C.: Semantic Models for Distributed Object Reflection. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
23. Mühlbauer, T.: Formal Specification and Analysis of Cloud Computing Management. Master's thesis, LMU Munich (2011)
24. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. HOSC 20(1–2), 161–196 (2007)
25. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns. Wiley (2005)
26. Sen, K., Viswanathan, M., Agha, G.: On Statistical Model Checking of Stochastic Systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005)
27. W3C. Request-Response Message Exchange Pattern (September 2011), http://www.w3.org/TR/2003/PR-soap12-part2-20030507/#singlereqrespmep
28. Walfish, M., Vutukuru, M., Balakrishnan, H., Karger, D.R., Shenker, S.: DDoS defense by offense. In: ACM SIGCOMM, pp. 303–314 (2006)
29. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. JIC 204(9), 1368–1409 (2006)
30. Yu, C.-F., Gligor, V.: A Specification and Verification Method for Preventing Denial of Service. IEEE T-SE 16(6), 581–592 (1990)

Loose Programming with PROPHETS

Stefan Naujokat, Anna-Lena Lamprecht, and Bernhard Steffen

Dortmund University of Technology, Chair for Programming Systems, Dortmund, D-44227, Germany
{stefan.naujokat,anna-lena.lamprecht,bernhard.steffen}@cs.tu-dortmund.de

Abstract. Loose programming is an extension to graphical process modeling that is tailored to automatically complete underspecified (loose) models using a combination of data-flow analysis and LTL synthesis. In this tool demonstration we present PROPHETS[1], our current implementation of the loose programming concept. The first part of the demonstration focuses on the preparative domain modeling, where a domain expert annotates the available services with semantic (ontological) information. The second part is then concerned with the actual loose programming, where a process modeler orchestrates the services without having to care about technical details like correct typing, interface compatibility, or platform-specific details. The orchestrated process skeletons are treated as loose service orchestrations that are automatically completed to running applications.

1 Introduction

In service-oriented software development approaches, the specification of (business) processes usually requires detailed knowledge of the available services, their behavior and capabilities. Our concept of loose programming [1] aims at providing easy access to and experimentation with (often unmanageably large) libraries of services. With loose specification, process designers are given the opportunity to sketch their intents roughly, while the backing data-flow analysis and linear-time synthesis handle the concretization automatically.

PROPHETS[1] extends the graphical modeling framework jABC [2] by the loose programming concepts. To enable loose specification and synthesis on a given library of services, semantic information on the services, i.e., a *domain model* is needed. Therefore, there are two user roles defined to work with PROPHETS: While the *domain expert* provides information on services and data types, the *process developer* uses it to semi-automatically create workflows.

In the following, Section 2 explains the domain modeling concepts by means of a simple example domain. Then, Section 3 presents how the domain model is applied for the synthesis of loosely specified processes. Section 4 concludes with a short discussion of related and future work.

[1] PROPHETS is available for download at http://prophets.cs.tu-dortmund.de. This site also provides technical documentation and further information on loose programming.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 94–98, 2012.

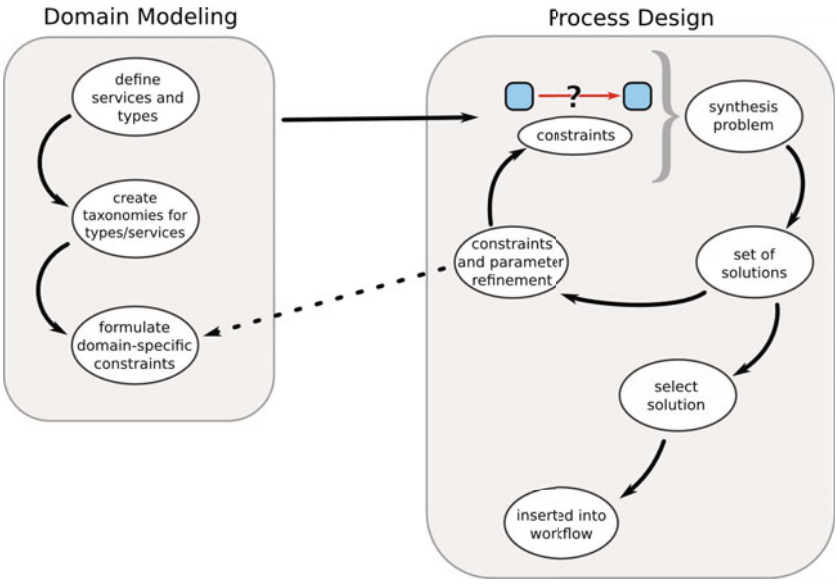


Fig. 1. The workflows of the two user roles involved in loose programming

2 Modeling the Domain

The here presented domain literally corresponds to the 'Hello World' example common to all programming languages. Our domain consists of three services: 'SayHello' sends a message, while 'Understand' receives one, with the language of both being configurable. Naturally, the latter service can only understand the message if it is in its known language. Therefore, the third service, 'Translate', can convert a message from one language into another. Unfortunately, not all language combinations are directly translatable. Only translations from one country's language to its direct neighbors' languages are valid (here limited to Western Europe). In our example scenario, the process developer wants to model a process that sends a message in one language and receives it in another, but he is not familiar with the geography of Europe. The domain expert has this information and provides the semantic annotations to the three services as well as (possibly) further constraints for the composition of services.

Setting up the domain model consists of three major steps (see Fig. 1):

1. At first, the domain expert has to create the service definition file. This mainly requires the identification of symbolic names for types and services, and the behavioral description of the services in terms of their input and output types. Multiple possible type combinations for one service (as it is the case here) simply lead to multiple entries in the service definition file.
2. Secondly, the domain expert may define taxonomies on the types and services. Although this is not strictly required, it may be useful for further structuring of the domain. Here, it might make sense to group all 'Translate'

service instances into one group, all 'SayHello' service instances into another etc.. The types (which are languages in this example) can, for example, be grouped according to language families.

3. Finally, the domain expert may define general domain-specific knowledge by global formulas that are used as constraints for every synthesis. In this example it makes sense to prevent the synthesis from utilizing any of the 'SayHello' services as part of the solution. This becomes necessary, because the synthesis algorithm tries to solve the loose specification by satisfying the input requirements of the target SIB. As the 'SayHello' services have no input requirements, they can be used anywhere to produce every language. The solution problem would be solved formally, but nothing has been actually translated. Such a solution would not be acceptable for the process developer.

3 Process Synthesis

Process design with the jABC (cf. Fig. 2) consists of taking SIBs[2] from the SIB library (A), placing them at the graph canvas (B) and connecting them with directed labeled edges (branches) according to the flow of control. Configuration (i.e. setting parameters) of SIB instances is done using the SIB inspector (C). A model that is defined in this way can then directly be executed and debugged with the integrated interpreter (D). In addition to this kind of complete specification, the PROPHETS plugin enables the process developer to mark one or

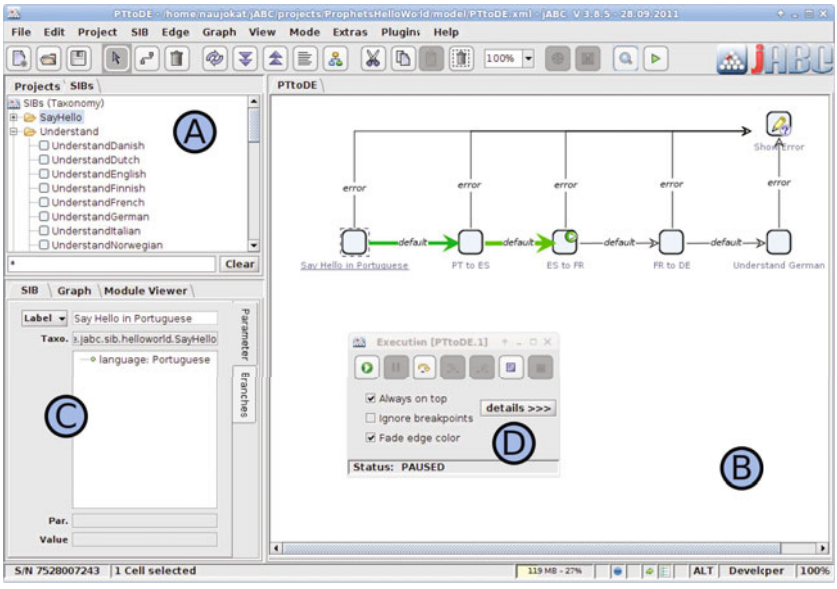


Fig. 2. Overview of jABC GUI elements

more branches as *loosely specified*. PROPHETS' synthesis is then applied to each of the loose branches to replace them by concrete realizations (see Fig. 3)

The plugin determines the start types for the synthesis automatically by performing a data-flow analysis on the process model. The types that are available at the source of the loosely specified branch are used as initial state for the synthesis. As goal types the synthesis uses the input types of the loosely specified branch's target SIB.

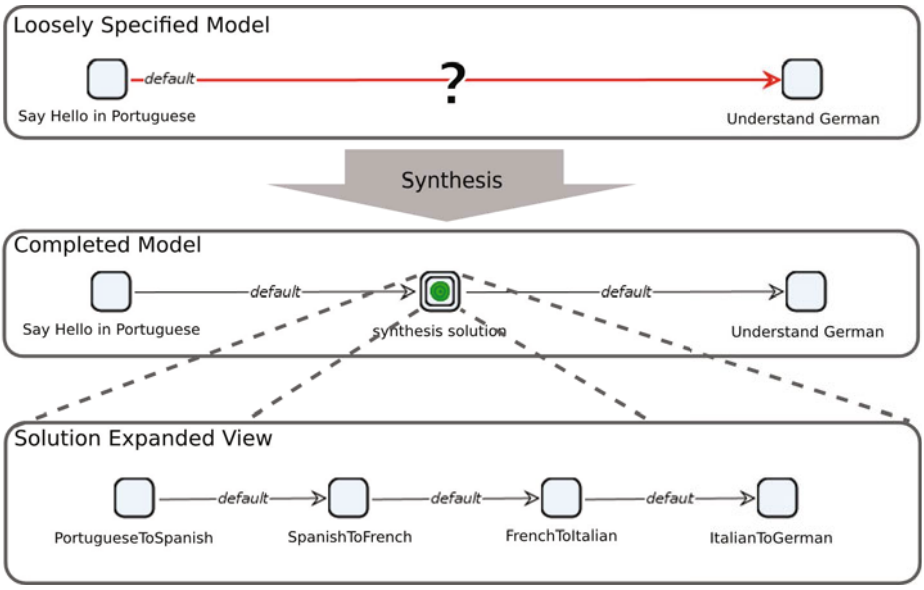


Fig. 3. Loosely specified process and completed model after synthesis

In addition to the inferred start and goal types, the synthesis can be guided by constraints in SLTL [3,4]. However, the expertise of a process designer in the jABC usually covers rather knowledge on business processes than software programming, and likewise we assume that specification of process requirements with formulas in temporal logic is beyond his interests. Therefore, PROPHETS incorporates a concept for template-based constraint specifications. The templates, which can easily be defined and extended by the domain expert, present a description of the constraint in plain natural language to the process designer. The description contains variable parts, which are translated into drop-down boxes for the process designer to assign values.

A process designer can also profit from a specified domain without using the synthesis feature. If a PROPHETS service definition exists, a jABC model can be automatically verified. The plugin then checks if all SIBs have their required

[2] A *SIB* (*Service Independent Building Block*) forms a wrapper for any kind of service that is used in the jABC.

types (input types) available on execution, whatever execution path might lead to this SIB. This is done by a combination of the previously mentioned data-flow analysis and GEAR [5], the model-checking-plugin for the jABC.

4 Conclusion

The here presented example is kept very simple on purpose, so that despite the limited space in this paper, we can elaborate on both of the two basic concepts when working with loose programming. More complex domains, especially in the context of bioinformatics analysis workflows, have shown the applicability of our approach [6,7]. Furthermore, the flexible architecture allows one to change (and even synthesize) the synthesis process itself in order to adapt to special needs of the domain in question [8]. In fact, PROPHETS supports self-application: loosely defined synthesis processes can be completed and executed. Subject of ongoing research are the improvement of the synthesis performance with domain-specific heuristics as well as further concepts for the automatic creation of the domain model, e.g. by learning from service logs or exploiting structural information about the service domain.

References

1. Lamprecht, A.L., Naujokat, S., Margaria, T., Steffen, B.: Synthesis-Based Loose Programming. In: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC). (September 2010)
2. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-Driven Development with the jABC. In: Hardware and Software, Verification and Testing. Volume 4383 of LNCS. Springer Berlin / Heidelberg (2007) 92–108
3. Steffen, B., Margaria, T., Freitag, B.: Module Configuration by Minimal Model Construction. Technical report, Universität Passau (1993)
4. Steffen, B., Margaria, T., von der Beeck, M.: Automatic synthesis of linear process models from temporal constraints: An incremental approach. In ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS'97) (1997)
5. Bakera, M., Margaria, T., Renner, C., Steffen, B.: Tool-supported enhancement of diagnosis in model-driven verification. Innovations in Systems and Software Engineering **5** (2009) 211–228
6. Lamprecht, A.L., Naujokat, S., Steffen, B., Margaria, T.: Constraint-Guided Workflow Composition Based on the EDAM Ontology. In: Proc. of the Workshop on Semantic Web Applications and Tools for Life Sciences. Volume 698., Berlin, CEUR Workshop Proceedings (December 2010)
7. Lamprecht, A.L., Naujokat, S., Margaria, T., Steffen, B.: Semantics-based composition of EMBOSS services. Journal of Biomedical Semantics **2**(Suppl 1) (2011)
8. Naujokat, S., Lamprecht, A.L., Steffen, B.: Tailoring Process Synthesis to Domain Characteristics. In: Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS). (2011)

Schedule Insensitivity Reduction

Vineet Kahlon

NEC Labs, Princeton, USA

Abstract. The key to making program analysis practical for large concurrent programs is to isolate a small set of interleavings to be explored without losing precision of the analysis at hand. The state-of-the-art in restricting the set of interleavings while guaranteeing soundness is partial order reduction (POR). The main idea behind POR is to partition all interleavings of the given program into equivalence classes based on the partial orders they induce on shared objects. Then for each partial order at least one interleaving need be explored. POR classifies two interleavings as non-equivalent if executing them leads to different values of shared variables. However, some of the most common properties about concurrent programs like detection of data races, deadlocks and atomicity as well as assertion violations reduce to control state reachability. We exploit the key observation that even though different interleavings may lead to different values of program variables, they may induce the same control behavior. Hence these interleavings, which induce different partial orders, can in fact be treated as being equivalent. Since in most concurrent programs threads are *loosely coupled*, i.e., the values of shared variables typically flow into a small number of conditional statements of threads, we show that classifying interleavings based on the control behaviors rather than the partial orders they induce, drastically reduces the number of interleavings that need be explored. In order to exploit this loose coupling we leverage the use of dataflow analysis for concurrent programs, specifically numerical domains. This, in turn, greatly enhances the scalability of concurrent program analysis.

1 Introduction

Verification of concurrent programs is a hard problem. A key reason for this is the behavioral complexity resulting from the large number of interleavings of transitions of different threads. While there is a substantial body of work devoted to addressing the resulting state explosion problem, a weakness of existing techniques is that they do not fully exploit structural patterns in real-life concurrent code. Indeed, in a typical concurrent program threads are *loosely coupled* in that there is limited interaction between values of shared objects and control flow in threads. For instance, data values written to or read from a shared file typically do not flow into conditional statements in the file system code. What conditional statements may track, for instance, are values of status bits for various files, e.g., whether a file is currently being accessed, etc. However, such status bits affect control flow in very limited and simplistic ways.

One of the main reasons why programmers opt for limited interaction between shared data and control in threads is the fundamental fact that concurrency is complex. A deep interaction between shared data and control would greatly complicate the debugging process. Secondly, the most common goal when creating concurrent programs is to exploit parallelism. Allowing shared data values to flow into conditional statements

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 99–114, 2012.

would require extensive use of synchronization primitives like locks to prevent errors like data races thereby killing parallelism and adversely affecting program efficiency.

An important consequence of this loose coupling of threads is that even though different interleavings of threads may results in different values of shared variables, they may not induce different program behaviors in that the control paths executed may remain unchanged. Moreover, for commonly occurring correctness properties like absence of data races, deadlocks and atomicity violations, we are interested only in the control behavior of concurrent programs. Indeed, data race detection in concurrent programs reduces to deciding the temporal property $\mathsf{EF}(c_1 \wedge c_2)$, where c_1 and c_2 are control locations in two different threads where the same shared variable is accessed and disjoint sets of locks are held. Similarly, checking an assertion violation involving an expression *expr* over control locations as well as program variables, can be reduced to control state reachability of a special location *loc* resulting via the introduction of a program statement of the form `if(expr) GOTO loc;` . Thus structural patterns in real-life programs as well as in commonly occurring properties are best exploited via reduction techniques that preserve control behaviors of programs rather than the actual behavior defined in terms of program states.

The state-of-the-art in state space reduction for concurrent program analysis is Partial Order Reduction (POR) [3,8,9]. The main idea behind POR is to partition all interleavings of the given program into equivalence classes based on the partial orders they induce on shared objects. Then for each partial order at least one interleaving need be explored. However, a key observation that we exploit is that because of loose coupling of threads even if different interleavings result in different values of shared (and local) variables, they may not induce different control behaviors. In order to capture how different interleavings may lead to different program behaviors, we introduce the notion of *schedule sensitive* transitions. Intuitively, we say that dependent transitions t and t' are schedule sensitive if executing them in different relative orders affects the behavior of the concurrent program, i.e., changes the valuation of some conditional statement that is dependent on t and t'. POR would explore both relative orders of t and t' irrespective of whether they induce different control behaviors or not whereas our new technique explores different relative orders of t and t' only if they induce different control behaviors. In other words, POR classifies interleavings with respect to global states, i.e., control locations *as well as* the values of program variables, as opposed to just control behavior. However, classifying computations based solely on control behaviors raises the level of abstraction at which partial orders are defined which results in the collapse of several different (state defined) partial orders, i.e., those inducing the same control behavior. This can result in drastic state space reduction.

The key challenge in exploiting the above observations for state space reduction is that deducing schedule insensitivity requires us to reason about program semantics, i.e., whether different interleavings could affect valuations of conditional statements. In order to carry out these checks statically, precisely and in a tractable fashion we leverage the use of dataflow flow analysis for concurrent programs. We show that schedule insensitivity can be deduced in a scalable fashion via the use of numerical invariants like ranges, octagons and polyhedra [7,2]. Then by exploiting the semantic notion of schedule insensitivity we show that we can drastically reduce the set of interleavings that need be explored over and above POR.

2 Motivation

Consider a concurrent program $\mathcal{P}$ comprised of the two threads T_1 and T_2 shown in fig 1(a) accessing shared variable *sh*. Suppose that we are interested in the reachability of the global control state (a_4, b_4). Since all transitions write to the same shared variable, i.e., *sh*, each of the transitions a_1, a_2 and a_3 is dependent with each of b_1, b_2 and b_3 except for the pair (a_3, b_3) both of which are read operations. As a result, in applying POR we would need to explore all interleavings of the local transitions of the two threads except a_3 and b_3. This results in the transition digram shown in fig. 1(b) where a pair of the form (c_1, c_2) indicates that thread T_i is at location c_i but hasn't executed the statement at c_i. A downward arrow to the left (right) signifies a move by T_1 (T_2).

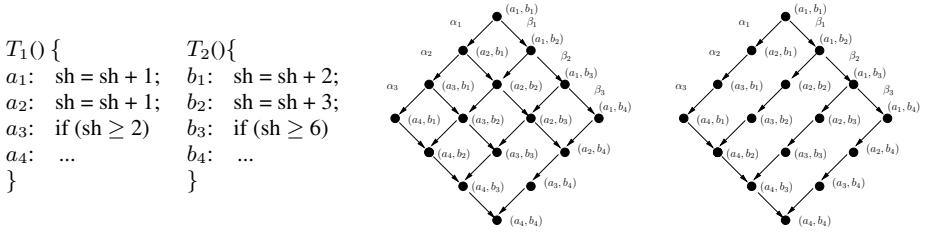


Fig. 1. (a) (b) (c)

However, if we track the values of the shared variable *sh* (assuming it was initialized to 0), we see that at global states (a_3, b_1), (a_3, b_2), (a_3, b_3) and (a_3, b_4), $sh \geq 2$ as a result of which the if-condition at location a_3 of T_1 always evaluates to *true*. This leads to the key observation that even though the statements a_i and b_j, where $i \neq 3$ and $j \neq 3$, are dependent and executing them in different order results in different values of *sh*, it does not affect the valuation of the conditional statement at a_3. Thus with respect to a_3 we need not explore different interleavings of the operations of T_1 and T_2. In fact it suffices to explore just one interleaving, i.e., a_1, a_2, a_3, b_1, b_2 and b_3. Consider now the conditional statement *sh* $\geq$ *6* at b_3. The guard evaluates to *false* in state (a_1, b_3) but evaluates to *true* in each of the states (a_2, b_3) and (a_3, b_3). Starting from state (a_1, b_1), we see that we can reach the global state (a_2, b_3) where *sh* $\geq$ *6* and the state (a_1, b_3) where *sh* $<$ *6*. Thus at (a_1, b_1), we need to explore paths starting with the transition $(a_1, b_1) \rightarrow (a_2, b_1)$ as well as those starting with $(a_1, b_1) \rightarrow (a_1, b_2)$. This is because executing one of these transitions may result in the conditional statement b_3 evaluating to *true* and executing the other may result in it evaluating to *false*. Similarly, from state (a_1, b_2) we need to explore paths starting via both its out-going transitions.

On reaching state (a_2, b_1), however, we see that all interleavings lead either to (a_2, b_3) or to (a_3, b_3) and at both of these states $sh \geq 6$, i.e., the conditional statement at b_3 evaluates to *true*. In other words, starting at state (a_2, b_1) the precise interleaving that is executed does not matter with respect to the valuation of b_3. We would therefore like to explore just one of these interleavings leading to (a_4, b_4). Hence starting at global state (a_2, b_1) we explore just one successor. We choose to explore the one resulting from the transition fired by T_1. Using a similar reasoning, we can see that it suffices to allow only T_1 to execute in each of the states (a_2, b_2) and (a_3, b_2). Furthermore, at the states

(a_4, b_1), (a_4, b_2) we have no choice but to execute T_2. Similarly, at the states (a_1, b_4) and (a_2, b_4) we have no choice but to execute T_1. This leads to the transition graph shown in fig. 1(c) clearly demonstrating the reduction (as compared to fig. 1(b)) in the set of interleavings that need be explored.

In order to exploit the above observations, we need to determine for each state (a_i, b_j) in the transaction graph and each conditional statement *con* reachable from (a_i, b_j), whether *con* either evaluates to *true* along all interleavings starting at (a_i, b_j) or evaluates to *false* along all such interleavings. In general, this is an undecidable problem. On the other hand, in order for our technique to be successful our method needs to be scalable to real-life programs. Dataflow analysis is ideally suited for this purpose. Indeed, in our example if we carry out range analysis, i.e., track the possible range of values that *sh* can take, we can deduce that at the locations (a_3, b_1), (a_3, b_2) and (a_3, b_3), *sh* lies in the ranges $[2, 2]$, $[4, 4]$ and $[7, 7]$, respectively. From this it follows easily that the conditional statement at a_3 always evaluates to *true*. It has recently been demonstrated that not only ranges but even more powerful numerical invariants like octagons [7] and polyhedra [2] can be computed efficiently for concurrent programs all of which can be leveraged to deduce schedule insensitivity. A key point is that exploiting numerical invariants to falsify or validate conditional statements offers a good trade-off between precision and scalability. This allows us to filter out interleavings efficiently which can, in turn, be leveraged to make model checking more tractable.

3 System Model

We consider concurrent systems comprised of a finite number of processes or threads where each thread is a deterministic sequential program written in a language such as C. Threads interact with each other using communication/synchronization objects like shared variables, locks and semaphores.

Formally, we define a concurrent program $\mathcal{CP}$ as a tuple $(\mathcal{T}, \mathcal{V}, \mathcal{R}, s_0)$, where $\mathcal{T} = \{T_1, ..., T_n\}$ denotes a finite set of threads, $\mathcal{V} = \{v_1, ..., v_m\}$ a finite set of shared variables and synchronization objects with v_i taking on values from the set V_i, $\mathcal{R}$ the transition relation and s_0 the initial state of $\mathcal{CP}$. Each thread T_i is represented by the control flow graph of the sequential program it executes, and is denoted by the pair (C_i, R_i), where C_i denotes the set of control locations of T_i and R_i its transition relation. A global state s of $\mathcal{CP}$ is a tuple $(s[1], ..., s[n], v[1], ..., v[m]) \in \mathcal{S} = C_1 \times ... \times C_n \times V_1 \times ... \times V_m$, where $s[i]$ represents the current control location of thread T_i and $v[j]$ the current value of variable v_j. The global state transition digram of $\mathcal{CP}$ is defined to be the standard interleaved composition of the transition diagrams of the individual threads. Thus each global transition of $\mathcal{CP}$ results by firing a local transition t of the form (a_i, g, u, b_i), where a_i and b_i are control locations of some thread $T_i = (C_i, R_i)$ with $(a_i, b_i) \in R_i$; g is a guard which is a Boolean-valued expression on the values of local variables of T_i and global variables in $\mathcal{V}$; and u is a set of operations on the set of shared and local variables of T_i that encodes how the value of these variables are modified. Formally, an operation *op* on variable v is a partial function of the form $IN \times V \rightarrow OUT \times V$, where V is the set of possible values of v and IN and OUT are, respectively, the set of possible input and output values of the operation. The notation $op(in, v_1) \rightarrow (out, v_2)$ denotes execution of operation *op* of v with input value in yielding output *out* while changing the value of v from v_1 to v_2. Given a transition (a_i, g, u, b_i), the set of operations appearing in g and u are said to be *used* by t and are

denoted by $used(t)$. Also, for transition $t : (a_i, g, u, b_i)$, we use $pre(t)$ and $post(t)$ to denote control locations a_i and b_j, respectively. A transition $t = (a_i, g, u, b_i)$ of thread T_i is enabled in state s if $s[i] = a_i$ and guard g evaluates to true in s. If $s[i] = a_i$ but g need not be true in s, then we simply say that t is *scheduled* in s. We write $s \xrightarrow{t} s'$ to mean that the execution of t leads from state s to s'. Given a transition $t \in \mathcal{T}$, we use $proc(t)$ to denote the process executing t. Finally, we note that each concurrent program $\mathcal{CP}$ with a global state space $\mathcal{S}$ defines the global transition system $A_G = (\mathcal{S}, \Delta, s_0)$, where $\Delta \subseteq \mathcal{S} \times \mathcal{S}$ is the *transition relation* defined by follows: for $s, s' \in Q$, $(s, s') \in \Delta$ if $\exists t \in \mathcal{T} : s \xrightarrow{t} s'$; and s_0 is the initial state of $\mathcal{CP}$. For ease of exposition, in this paper we consider concurrent programs with only two threads although our techniques extend easily to programs with multiple threads.

4 Schedule Insensitivity Reduction

The state-of-the-art in state space reduction for concurrent program analysis is Partial Order Reduction (POR) [3,8,9]. POR classifies computations based solely on the partial orders they induce. These partial orders are defined with respect to global states, i.e., control locations *as well as* the values of program variables, as opposed to just control behavior. However, classifying computations based solely on control behavior raises the level of abstraction at which partial orders are defined which results in the collapse of several different (state defined) partial orders, i.e., those inducing the same control behavior. Whereas (ideally) POR would explore at least one computation per partial order, the goal of our new reduction is to explore only one computation for all these collapsed partial orders. This can result in drastic state space reduction.

Concurrent Def-Use Chains and Control Dependency. Control flow within a thread is governed by valuations of conditional statements. However, executing thread transitions accessing shared objects in different orders may result in different values of these shared objects resulting in different valuations of conditional statements of threads and hence different control paths being executed. Note that the valuation of a conditional statement *cond* will be so affected only if the value of a shared variable *flows* into *cond*. This dependency is captured using the standard notion of a *def-use chain*. A *definition* of a variable v is taken to mean an assignment (either syntactic or semantic, e.g., via a pointer) to v. A *definition-use chain (def-use chain)* consists of a definition of a variable in a thread T and all the uses, i.e., read accesses, reachable from that definition in (a possibly different) thread T' without any other intervening definitions. Note that due to the presence of shared variables a def-use chain may, depending on the scheduling of thread operations, span multiple threads. Thus different interleavings can affect the valuation of a conditional statement *cond* only if there is a def-use chain starting from an operation writing to a shared variable sh and leading to *cond*. This is formalized using the notion of *control dependency*.

Definition. (Control Dependency). *We say that a conditional statement cond at location loc of thread T is control dependent on an assignment statement st of thread T' (possibly different from T) if there exists a computation x of the given concurrent program leading to a global state with T at location loc such that there is a def-use chain from st to cond along x.*

Schedule Insensitivity. In order to capture how different interleavings may lead to different program behaviors, we introduce the notion of schedule sensitive (or equivalently schedule insensitive) transitions. Intuitively, we say that transitions t and t' of two different threads are *schedule sensitive* if executing them in different relative orders affects the behavior of the concurrent program, i.e., changes the valuation of some conditional statement that is control dependent on t and t'. Formally,

Definition (Schedule Sensitive Operations). *Let OP be the set of operations on variable var. Then $\mathsf{Sen} \subseteq OP \times OP \times \mathcal{S}$ is a schedule sensitivity relation for var if for $s \in \mathcal{S}$ and op_1, $op_2 \in OP$, the following holds: if v is the value of var in s then for all possible inputs in_1 and in_2 we have*

- $(op_1, op_2, s) \notin \mathsf{Sen}$ *(op_1 and op_2 are schedule insensitive in s) implies that* $(op_2, op_1, s) \notin \mathsf{Sen}$,
- *if $op_1(in_1, v)$ is defined and $op_1(in_1, v) \rightarrow (out_1, v_1')$, then $op_2(in_2, v)$ is defined if and only if $op_2(in_2, v_1')$ is defined; and*
- *if $op_1(in_1, v)$ and $op_2(in_2, v)$ are defined, then each conditional statement cond that is control dependent on op_1 or op_2 and is scheduled in state $t \in \mathcal{S}$ either evaluates to true along all paths of the given concurrent program leading from s to t or it evaluates to false along all such paths.*

Definition (Schedule Insensitive Transitions). *Two transitions t_1 and t_2 are schedule insensitive in state s if*

- *the threads executing t_1 and t_2 are different, and*
- *either t_1 and t_2 are independent in s, or for all $op_1 \in used(t_1)$ and $op_2 \in used(t_2)$, if op_1 and op_2 are operations on the same shared object then op_1 and op_2 are schedule insensitive in s, i.e., $(op_1, op_2, s) \notin \mathsf{Sen}$.*

In the above definition we use the standard notion of (in)dependence of transitions as used in the theory of partial order reduction (see [3]). The motivation behind defining schedule insensitive transitions is that if in a global state s, transitions t_1 and t_2 of threads T_1 and T_2, respectively, are dependent then we need to consider interleavings where t_1 and t_2 are executed in different relative orders only if there exists a conditional statement *cond* such that *cond* is control dependent on both t_1 and t_2 and its valuation is affected by executing t_1 and t_2 in different relative orders, i.e., t_1 and t_2 are schedule sensitive in s.

We next define the notion of *control equivalent* computations which is the analogue of Mazurkiewicz equivalent computations for schedule sensitive transitions.

Definition (Control Equivalent Computations). *Two computations x and y are said to be control equivalent if x can be obtained from y by repeatedly permuting adjacent pairs of schedule insensitive transitions, and vice versa.*

Note that control equivalence is a coarser notion of equivalence than Mazurkiewicz equivalence in that Mazurkiewicz equivalence implies control equivalence but the reverse need not be true. That is precisely what we need for more effective state space reduction than POR.

5 Deducing Schedule Insensitivity

In order to exploit schedule insensitivity for state space reduction we need to provide an effective, i.e., automatic and lightweight, procedure for deciding schedule insensitivity of a pair of transitions. By definition, in order to infer whether t_1 and t_2 are schedule sensitive, we have to check whether there exists a conditional statement *cond* satisfying the following: (i) **Control Dependence:** of *cond* on t_1 and t_2, (ii) **Reachability:** *cond* is enabled in a state t reachable from s, and (iii) **Schedule Sensitivity:** there exist interleavings from s leading to states with different valuations of *cond*.

In order to carry out these checks statically, precisely and in a tractable fashion we leverage the use of dataflow flow analysis for concurrent programs. As was shown in the motivation section, by using range analysis, we were able to deduce schedule insensitivity of the local states (a_i, b_j), where $i \in [2..3]$ and $j \in [1..3]$ which enabled us to explore only one transition from each of them. We can, in fact, leverage even more powerful numerical invariants like octagons [7] and polyhedra [2].

Transaction Graph. In order to deduce control dependence, reachability and schedule sensitivity, we exploit the notion of a *transaction graph* which has previously been used for dataflow analysis of concurrent programs (see [4]). The main motivation behind the notion of a transaction graph is to capture thread interference, i.e., how threads could affect dataflow facts at each others locations. This is because, in practice, concurrent programs usually do not allow unrestricted interleavings of local operations of threads. Typically, synchronization primitives like locks and Java-style wait/notifies, are used in order to control accesses to shared data or introduce causality constraints. Additionally, the values of shared variables may affect valuations of conditional statements which, in turn, may restrict the allowed set of interleavings. The allowed set of interleavings in a concurrent program are determined by control locations in threads where context switches occur. In order to identify these locations the technique presented in [4] delineates *transactions*. A transaction of a thread is a maximal atomically executable piece of code, where a sequence of consecutive statements in a given thread T are *atomically executable* if executing them without any context switch does not affect the outcome of the dataflow analysis at hand. Once transactions have been delineated, the thread locations where context switches need to happen can be identified as the start and end points of transactions. The transactions of a concurrent program are encoded in the form of a *transaction graph* the definition of which is recalled below.

Definition (Transaction Graph) [4]. *Let $\mathcal{CP}$ be a concurrent program comprised of threads T_1,..., T_n and let C_i and R_i be the set of control locations and transitions of the CFG of T_i, respectively. A transaction graph $\Pi_{\mathcal{CP}}$ of $\mathcal{CP}$ is defined as $\Pi_{\mathcal{CP}} = (C_{\mathcal{CP}}, R_{\mathcal{CP}})$, where $C_{\mathcal{CP}} \subseteq C_1 \times ... \times C_n$ and $R_{\mathcal{CP}} \subseteq (C_1, ..., C_n) \times (C_1, ..., C_n)$. Each edge of $\Pi_{\mathcal{CP}}$ represents the execution of a transaction by a thread T_i, say, and is of the form $(l_1, \ldots, l_i, \ldots, l_n) \rightarrow (m_1, \ldots, m_i, \ldots, m_n)$ where (a) starting at the global state $(l_1, ..., l_n)$, there is an atomically executable sequence of statements of T_i from l_i to m_i, and (b) for all $j \neq i$, $l_j = m_j$.*

Note that this definition of transactions is quite general, and allows transactions to be inter-procedural, i.e., begin and end in different procedures, or even begin and end inside loops. Also, transactions are not only program but also analysis dependent.

Our use of transaction graphs for deducing schedule insensitivity, is motivated by several reasons. First, transaction graphs allow us to carry out dataflow analysis for the

concurrent program at hand which is crucial in reasoning about schedule insensitivity. Secondly, transaction graphs already encode reachability information obtained by exploiting scheduling constraints imposed by both synchronization primitives as well as shared variables. Finally, the transaction graph encodes concurrent def-use chains which we use in inferring control dependency. In other words, transaction graphs encodes all the necessary information that allows us to readily decide schedule sensitivity.

Transaction Graph Construction. We now recall the transaction graph construction [4] which is an iterative refinement procedure that goes hand-in-hand with the computation of numerical invariants (steps 1-9 of alg. 1). In other words, the transaction graph construction and computation of numerical invariants are carried out simultaneously via the same procedure.

First, an initial set of (coarse) transactions are identified by using scheduling constraints imposed by synchronization primitives like locks and wait/notify and ignoring the effects of shared variables (step 3-7 of alg. 1). This step is essentially classical POR carried out over the product of the control flow graphs of the given threads. This initial synchronization-based transaction delineation acts as a bootstrapping step for the entire transaction delineation process. These transactions are used to compute the initial set of numerical (ranges/octagonal/polyhedral) invariants. Note that once a (possibly coarse) transaction graph is generated dataflow analysis can be carried out exactly as for sequential programs. However, based on these sound invariants, it may be possible to falsify conditional statements that enable us to prune away unreachable parts of the program (Step 8) (see [4] for examples). We use this sliced program, to re-compute (via steps 3-7) transactions based on synchronization constraints which may yield larger transactions. This, in turn, may lead to sharper invariants (step 8). The process of progressively refining transactions by leveraging synchronization constraints and sound invariants in a dovetailed fashion continues till we reach a fix-point.

Deducing Schedule Insensitivity. The transaction graph as constructed via the algorithm described in [4] encodes transactions or context switch points as delineated via a refinement loop that dovetails classical POR and slicing induced by numerical invariants. In order to incorporate the effects of schedule insensitivity we refine this transaction delineation procedure to avoid context switches induced by pairs of transitions of different threads that are dependent yet schedule insensitive.

The procedure for schedule insensitive transaction graph construction is formalized as alg. 1. Steps 1-9 of alg. 1 are from the original transaction delineation procedure given in [4]. In order to collapse partial orders by exploiting schedule insensitivity, we introduce the additional steps 10-32. We observe that given a state (l_1, l_2) of the transaction graph, a context switch is required at location l_1 of thread T_1 if there exists a global state (l_1, m_2) reachable from (l_1, l_2) such that l_1 and m_2 are schedule sensitive. This is because executing l_1 and m_2 in different orders may lead to different program behaviors. Since a precise computation of the schedule sensitivity relation is as hard as the verification problem, in order to determine schedule insensitivity of (l_1, m_2), we use a static over-approximation of the schedule sensitivity relation defined as follows:

Definition (Static Schedule Sensitivity). *Transitions t_1 and t_2 scheduled at control locations n_1 and n_2 of threads T_1 and T_2, respectively, are schedule insensitive at state (n_1, n_2) of the transaction graph if for each conditional statement* cond *such that*

Algorithm 1. Construction of Schedule Insensitive Transaction Graph

repeat
  Initialize $W = \{(in_1, in_2)\}$, where in_j is the initial state of thread T_j.
  repeat
    Remove a state (l_1, l_2) from W and add it to $Processed$
    Compute the set *Succ* of successors of (l_1, l_2) via POR by exploiting synchronization constraints **(Synchronization Constraints)**
    Add all states of $Succ$ not in $Processed$ to W.
  until W is empty
  Compute numerical invariants on the resulting synchronization skeleton to slice away unreachable parts of the program **(Shared Variable Constraints)**
until transactions cannot be refined further
repeat
  for each state (l_1, l_2) of Π **do**
    $control\_oblivious = true$
    for each global state (l_1, m_2) where m_2 and l_1 are dependent **do**
      for each conditional state *cond* scheduled at state (r_1, r_2), say, **do**
        if (r_1, r_2) is reachable from (l_1, m_2) **then**
          if *cond* is control dependent with l_1 and m_2 **then**
            if $inv_{(r_1,r_2)}$ is the invariant at location (r_1, r_2) and $\neg((inv_{(r_1,r_2)} \Rightarrow cond) \vee (inv_{(r_1,r_2)} \wedge cond = false))$ **then**
              $control\_oblivious = false$
            end if
          end if
        end if
      end for
    end for
    if $control\_oblivious$ **then**
      for each predecessor (k_1, l_2) of (l_1, l_2) in Π **do**
        for each successor (n_1, l_2) of (l_1, l_2) in Π **do**
          remove (l_1, l_2) as a successor of (k_1, l_2) and add (n_1, l_2) as a successor.
        end for
      end for
    end if
  end for
until no more states can be sliced

- cond *is reachable from* (n_1, n_2) *in the transaction graph (***Reachability***),*
- *there are concurrent def-use chains in the transaction graph from both* n_1 *and* n_2 *to* cond *(***Control Dependence***),*
- cond *either evaluates to* true *along all paths of the transaction graph from* (n_1, n_2) *to* cond *or it evaluates to* false *along all such paths (***Schedule Insensitivity***).*

Using dataflow analysis, these checks can be carried out in a scalable fashion.

Checking Reachability and Control Dependency. For our reduction to be precise it is important that while inferring schedule insensitivity we only consider conditional statements *cond* that are reachable from (l_1, m_2). As discussed before, reachability of global states is governed both by synchronization primitives and shared variable values and by using numerical invariants we can infer (un)reachability efficiently and with high precision. Importantly, this reachability information is already encoded in the transition

relation of the transaction graph. In order to check control dependence of *cond* on l_1 and m_2, we need to check whether there are def-use chains from a shared variable v written to at locations l_1 and m_2 to a variable u accessed in the conditional statement *cond* at location r_1 or r_2, where state (r_1, r_2) of the transaction graph is reachable from (l_1, l_2). Note that all states that have been deduced as unreachable via the use of numerical invariants and synchronization constraints have already been sliced away via step 8 of alg. 1. Thus it suffices to track def-use chains along the remaining paths (step 14) in the transaction graph starting at (l_1, l_2) (step 15). This can be accomplished in exactly the same way as in sequential programs - the only difference being that we do it along paths in the transaction graph so that def-use chains can span multiple threads.

Checking Schedule Insensitivity. Next, in order to deduce that a conditional statement *cond* scheduled in state (r_1, r_2) either evaluates to *true* along all paths from (l_1, m_2) to (r_1, r_2) or evaluates to false along all such paths, we leverage numerical invariants computed in step 8 of alg. 1. Let $inv_{(r_1,r_2)}$ be the (range, octagonal, polyhedral) invariant computed at (r_1, r_2). Then if *cond* is either falsified, i.e., $cond \wedge inv_{(r_1,r_2)} = false$ or *cond* is validated, i.e., $inv_{(r_1,r_2)} \Rightarrow cond$, the valuation of conditional statements in (r_1, r_2) are independent of the path from (l_1, m_2) to (r_1, r_2) (step 17). In order to check schedule-insensitivity of (l_1, m_2), we need to carry out the above check for every conditional statement that is reachable from (l_1, m_2) and has a def-use chain from both l_1 and m_2 to *cond*. If there exists no such conditional statement then we can avoid a context switch at location l_1 of thread T_1 (steps 24-30) thereby collapsing partial orders in the transaction graph.

Scalability Issues. A key concern in using transactions graphs for deducing schedule insensitivity is the state explosion resulting from the product construction. However, in practice, the transaction graph construction is very efficient due to three main reasons. First, in building the transaction graph we take the product over control locations and not local states of threads. Thus for k threads the size of the transaction graph is at most n^k, where n is the maximum number of lines of code in any thread. Secondly, when computing numerical invariants we use the standard technique of variable clustering wherein two variables u and v occur in a common cluster if there exists a def-use chain along which both u and v occur. Then it suffices to build the transaction graph for each cluster separately. Moreover, for clusters that contains only local thread variables there is no need to build the transaction graph as such variables do not produce thread dependencies. Thus cluster induced slicing can drastically cut down on the statements that need to be considered for each cluster and, as a result, the transaction graph size. Finally, since each cluster typically has few shared variables, POR (step 5) further ensures that the size of the transaction graph for each cluster is small. Finally, it is worth keeping in mind that the end goal of schedule insensitivity reduction is to help model checking scale better and in this context any transaction graph construction will likely be orders of magnitude faster than model checking which remains the key bottleneck.

6 Enhancing Symbolic Model Checking via Schedule Insensitivity

We show how to exploit schedule insensitivity for scaling symbolic model checking.

Schedule Insensitivity versus Partial Order Reduction. In order to illustrate the advantage of schedule insensitivity reduction we start by briefly recalling monotonic partial order reduction, a provably optimal symbolic partial order reduction technique. The

technique is optimal in that it ensures that exactly one interleaving is explored for every partial order induced by computations of the given program. Using schedule insensitivity we show how to enhance monotonic POR by further collapsing partial orders over and above those obtained via MPOR.

The intuition behind MPOR is that if all transitions enabled at a global state are independent then we need to explore just one interleaving. This interleaving is chosen to be the one in which transitions are executed in increasing (monotonic) order of their thread-ids. If, however, some of the transitions enabled at a global state are dependent than we need to explore interleavings that exercise both relative orders of these transitions which may violate the *natural* monotonic order. In that case, we allow an out-of-order-execution, viz., a transition tr' with larger thread-id than tr and dependent with tr to execute before tr.

Example. Consider the example in fig. 1. If we ignore dependencies between local transitions of threads T_1 and T_2 then MPOR would explore only one interleaving namely the one wherein all transitions of T_1 are executed before all transitions of T_2, i.e., the interleaving $\alpha_1\alpha_2\alpha_3\beta_1\beta_2\beta_3$ (see fig. 1(b)). Consider now the pair of dependent operations (a_1, b_1) accessing the same shared variable sh. We need to explore interleavings wherein a_1 is executed before b_1, and vice versa, which causes, for example, the out-of-order execution $\beta_1\alpha_1\alpha_2\alpha_3\beta_2\beta_3$ where transition β_1 of thread T_2 is executed before transition α_1 of thread T_1 even though the thread-id of β_1 is greater than the thread-id of α_1. MPOR guarantees that exactly one interleaving is explored for each partial order generated by dependent transitions.

When exploiting schedule insensitivity, starting at a global control state (c_1, c_2) an out-of-order execution involving transitions tr_1 and tr_2 of thread T_1 and T_2, respectively, is enforced only when (i) tr_1 and tr_2 are dependent, *and* (ii) tr_1 and tr_2 are schedule dependent starting at (c_1, c_2). Note that the extra condition (ii) makes the criterion for out-of-order execution stricter. This causes fewer out-of-order executions and further restricts the set of partial orders that will be explored over and above MPOR.

Going back to our example, we see that starting at global control state (a_2, b_2), transitions a_2 and b_2 are dependent as they access the same shared variable. Thus MPOR would explore interleavings wherein a_2 is executed before b_2 ($\alpha_1\beta_1\alpha_2\alpha_3\beta_2\beta_3$) and vice versa ($\alpha_1\beta_1\beta_2\alpha_2\alpha_3\beta_3$). However as shown in sec. 2, a_2 and b_2 are schedule insensitive and so executing a_2 and b_2 in different relative orders does not generate any new behavior. Thus we only explore one of these orders, i.e., a_2 executing before b_2 as *thread-id*$(a_2) = 1 < 2 =$*thread-id*(b_2). Thus after applying SIR, we see that starting at (a_2, b_2) only one interleaving, i.e., $\alpha_2\alpha_3\beta_2\beta_3$, is explored.

Implementation Strategy. Our strategy for implementing SIR is as follows:

1. We start by reviewing the basics of SAT/SMT-based bounded model checking.
2. Next we review the MPOR implementation wherein the scheduler is constrained so that it does not explore all enabled transitions as in the naive approach but only those that lead to the exploration of new partial orders via a monotonic ordering strategy as discussed above.
3. Finally we show how to implement SIR by further restricting the scheduler to explore only those partial orders that are generated by schedule sensitive dependent transitions. This is accomplished via the same strategy as in MPOR - the only difference being that we allow out-of-order executions between transitions that are not just dependent but also schedule sensitive.

Bounded Model Checking (BMC). Given a multi-threaded program and a reachability property, BMC can check the property on all execution paths of the program up to a fixed depth K. For each step $0 \leq k \leq K$, BMC builds a formula Ψ such that Ψ *is satisfiable iff there exists a length-k execution that violates the property*. The formula is denoted $\Psi = \Phi \wedge \Phi_{prop}$, where Φ represents all possible executions of the program up to k steps and Φ_{prop} is the constraint indicating violation of the property (see [1] for more details about Φ_{prop}). In the following, we focus on the formulation of Φ. Let $V = V_{global} \cup \bigcup V_i$, where V_{global} are global variables and V_i are local variables in T_i. For every local (global) program variable, we add a state variable to V_i (V_{global}). We add a pc_i variable for each thread T_i to represent its current program counter. To model nondeterminism in the scheduler, we add a variable sel whose domain is the set of thread indices $\{1, 2, \ldots, n\}$. A transition in T_i is executed only when $sel = i$.

At every time frame, we add a fresh copy of the set of state variables. Let $v^i \in V^i$ denote the copy of $v \in V$ at the i-th time frame. To represent all possible length-k interleavings, we first encode the transition relations of individual threads and the scheduler, and unfold the composed system exactly k time frames.

$$\Phi := \; I(V^0) \wedge \bigwedge_{i=0}^{k} (SCH(V^i) \wedge \bigwedge_{j=1}^{n} TR_j(V^i, V^{i+1}))$$

where $I(V^0)$ represents the set of initial states, SCH represents the constraint on the scheduler, and TR_j represents the transition relation of thread T_j. Without any reduction, $SCH(V^i) := true$, which means that sel takes all possible values at every step. This default SCH considers all possible interleavings. SIR can be implemented by adding constraints to SCH to remove redundant interleavings.

MPOR Strategy. As discussed before, the broad intuition behind MPOR is to execute location transitions of threads in increasing orders of their thread-ids unless dependencies force an out-of-order execution. In order to characterize situations where we need to force an out-of-order execution we use the notion of a *dependency chain*.

Definition (Dependency Chain) *Let t and t' be transitions such that $t <_x t'$, i.e., t is executed before t' along computation x. A dependency chain along x starting at t is a (sub-)sequence of transitions $tr_{i_0}, ..., tr_{i_k}$ fired along x, where (a) $i_0 < i_1 < ... < i_k$, (b) for each $j \in [0..k-1]$, tr_{i_j} is dependent with $tr_{i_{j+1}}$, and (c) there does not exist a transition fired along x between tr_{i_j} and $tr_{i_{j+1}}$ that is dependent with tr_{i_j}.*

For transitions t and t' fired along x, we use $t \Rightarrow_x t'$ to denote the fact that there is a dependency chain from t to t' along x. Then the MPOR strategy is as follows:

MPOR Strategy. *Explore only those computation x such that for each pair of transitions tr and tr' such that $tr' <_x tr$ we have $tid(tr') > tid(tr)$ only if either (i) $tr' \Rightarrow_x tr$, or (ii) there exists a transition tr'' such that $tid(tr'') < tid(tr)$, $tr' \Rightarrow_x tr''$ and $tr' <_x tr'' <_x tr$.*

Schedule Insensitivity Reduction. For implementing SIR, we only need to consider partial orders induced by those pairs of conflicting transitions that are schedule sensitive. This motivates the following definition.

Definition (Schedule-Dependency Chain) *Let t and t' be transitions fired along a computation x such that $t <_x t'$. A schedule-dependency chain along x starting at t is*

a (sub-)sequence of transitions $tr_{i_0}, ..., tr_{i_k}$ fired along x, where (a) $i_0 < i_1 < ... < i_k$, (b) for each $j \in [0..k-1]$, tr_{i_j} is schedule-dependent with $tr_{i_{j+1}}$, and (c) there does not exist a transition fired along x between tr_{i_j} and $tr_{i_{j+1}}$ that is schedule-dependent with tr_{i_j}.

For transitions t and t' fired along x, we use $t \Rightarrow_x^s t'$ to denote that the fact that there is a schedule-dependency chain from t to t' along x. Note that the difference between the above definition and that of a Dependency chain is that the above definition is more restrictive as it only consider chains over dependent transitions only if they are schedule-dependent. As a result is leads to exploration of fewer partial orders which in turn enhances scalability of state space exploration. Then the SIR strategy is as follows:

SIR. *Explore only those computations such that for each pair of transitions tr and tr' such that $tr' <_x tr$ we have $tid(tr') > tid(tr)$ only if either (i) $tr' \Rightarrow_x^s tr$, or (ii) there exists a transition tr'' such that $tid(tr'') < tid(tr)$, $tr' \Rightarrow_x^s tr''$ and $tr' <_x tr'' <_x tr$.*

Encoding SIR. In order to implement our technique, we need to track schedule dependency chains in a space efficient manner. Our encoding to track schedule dependency chains is similar to the one for tracking dependency chains in MPOR except that we consider schedule sensitivity as opposed to dependency of transitions in building these chains. In order to track schedule dependency chains, for each pair of threads T_i and T_j, we introduce a new variable SDC_{ij} defined as follows.

Definition. *$SDC_{il}(k)$ is 1 or -1 accordingly as there is a dependency chain or not, respectively, from the last transition executed by T_i to the last transition executed by T_l at or before time step k. If no transition has been executed by T_i up to time step k, $SDC_{il} = 0$.*

Updating SDC_{ij}. If at time step k thread T_i is executing transition tr, then for each thread T_l, we check whether the last transition executed by T_l is schedule sensitive with tr. To track that we introduce the dependency variables DEP_{li} defined below.

Definition. *$DEP_{li}(k)$ is* true *or* false *accordingly as the transition being executed by thread T_i at time step k is dependent with the last transition executed by T_l, or not. Note that $DEP_{ii}(k) = 1$ always holds (due to control conflict).*

For MPOR these dependency variables are enough to track dependency chains. However even if two transitions are dependent they might still be schedule insensitive. To carry out this additional check, we introduce the schedule sensitivity variables

Definition. *$SS_{li}(k)$ is* true *or* false *accordingly as the transition of thread T_i being executed at time step k is schedule sensitive with the last transition executed by T_l, or not. Note that $SS_{ii}(k)$ always holds true.*

We now show how the SDC variables are updated. If $(DEP_{li}(k) = 1) \wedge (SS_{li}(k) = true)$ and if $SDC_{jl}(k-1) = 1$, i.e., there is a schedule dependency chain from the last transition executed by T_j to the last transition executed by T_l, then this schedule dependency chain can be extended to the last transition executed by T_i, i.e., tr. In that case, we set $DC_{ji}(k) = 1$. Also, since we track schedule dependency chains only from the last transition executed by each thread, the schedule dependency chain corresponding to T_i needs to start afresh and so we set $SDC_{ij}(k) = -1$ for all $j \neq i$. To sum up, the updates are as follows.

$$
\begin{array}{ll}
SDC_{ii}(k) = 1 & \\
SDC_{ij}(k) = -1 & \text{when } j \neq i \\
SDC_{ji}(k) = 0 & \text{when } j \neq i \text{ and } SDC_{jj}(k-1) = 0 \\
SDC_{ji}(k) = \bigvee_{l=1}^{n} (SDC_{jl}(k-1) = 1 & \\
\qquad \wedge DEP_{li}(k) \wedge SS_{li}(k)) & \text{when } j \neq i \text{ and } SDC_{jj}(k-1) \neq 0 \\
SDC_{pq}(k) = SDC_{pq}(k-1) & \text{when } p \neq i \text{ and } q \neq i
\end{array}
$$

Scheduling Constraint. Next we introduce the scheduling constraints variables S_i, where $S_i(k)$ is *true* or *false* based on whether thread T_i can be scheduled to execute or not, respectively, at time step k in order to ensure quasi-monotonicity. Then we conjoin the following constraint to SCH:

$$\bigwedge_{i=1}^{n} (sel^k = i \Rightarrow S_i(k))$$

We encode $S_i(k)$ (where $1 \leq i \leq n$) as follows:
$S_i(0) = true$ and
for $k > 0$, $S_i(k) = \bigwedge_{j>i} (SDC_{ji}(k) \neq -1 \vee \bigvee_{l<i} SDC_{jl}(k-1) = 1)$

In the above formula, $SDC_{ji}(k) \neq -1$ encodes the condition that either a transition by thread T_j, where $j > i$, hasn't been executed up to time k, i.e., $SDC_{ji}(k) = 0$, or if it has then there is a schedule-dependency chain from the last transition executed by T_j to the transition of T_i enabled at time step k, i.e., $SDC_{ji}(k) = 1$. If these two cases don't hold and there exists a transition tr' fired by T_j before the transition tr of T_i enabled at time step k, then in order for quasi-monotonicity to hold, there must exist a transition *tr"* fired by thread T_l, where $l < i$, after tr' and before tr such that there is a schedule-dependency chain from tr' to tr'' which is encoded as $\bigvee_{l<i} SDC_{jl}(k-1) = 1$.

All we need to show now is how to encode the DEP and SS variables. The dependency variables are encoded exactly as in MPOR (see [5] for details). Thus as a final step we show how to encode the SS variables.

Encoding SS. For encoding SS variables we use the schedule insensitive transaction graph constructed in sec 5. In order to decide whether transitions $c_i \rightarrow d_i$ and $c_j \rightarrow d_j$ of threads T_i and T_j are schedule sensitive it suffices to check whether there exist paths in the transaction graph wherein c_i is executed before c_j along one and vice versa along the other. Note that since SIR allows context switching only at locations where shared variables are accessed, we can restrict ourselves to locations c_i and c_j satisfying this property. Moreover since we are interested only in the schedule (in)sensitivity of dependent transitions we can further assume that the statements at c_i and c_j are dependent.

To encode SS_{ij} we first compute the set *SS-Pairs*$_{ij}$ of all pairs (c_1, c_2) such that (i) c_1 and c_2 belong to threads T_i and T_j, (ii) there exists a pair of dependent transitions of the form $tr_1 : c_1 \rightarrow d_1$ and $tr_2 : c_2 \rightarrow d_2$, and (iii) there exist paths in the schedule insensitive transaction graph wherein c_1 is executed before c_2 along one and vice versa along the other. The sets *SS-Pairs*$_{ij}$ can be enumerated via a single traversal of the transaction graph. Then $SS_{ij} = \bigvee_{(c,d) \in SS-Pairs_{ij}} ((pc_i = c) \wedge (pc_j = d))$

Table 1. Model Checking Data Race Warnings (Timings are in seconds and memory in MBs)

Witness #	Shared Vars	Relevant Sh. Vars	Transaction Graph	MPOR		SIR	
				Time	Mem	Time	Mem
jfs_dmap : 1	6	1	0.01	0.02	59	0.01	12
ctrace : 1	19	12	10	2	62	1	43
ctrace : 2	19	12	14	10 hr	1.2G	3hr	0.5G
ctrace : 3	19	12	12	2303	733	1800	560
autofs : 1	7	2	0.05	1.14	60	0.5	30
autofs : 2	7	2	0.07	128	144	43	85
ptrace : 1	3	1	20	844	249	502	191
raid : 1	6	0	-	26.13	75	7.1	21
raid : 2	6	0	-	179	156	20	41
raid : 3	6	0	-	32.19	87	5	29
raid : 4	6	0	-	4.15	61	3	19
raid : 5	6	0	-	9.30	59	2	24
raid : 6	6	0	-	70	116	12	23
ipoib : 1	10	2	0.02	0.1	58	0.1	58
ipoib : 2	10	2	0.02	0.1	59	0.1	59
ipoib : 3	10	2	0.04	0.1	58	0.1	57
ipoib : 4	10	2	0.03	0.3	59	0.3	59

7 Implementation and Experimental Results

In previous work [6] we used static analysis to produce data race warnings for a suite of Linux device drivers downloaded from the Linux Kernel Archives. Each warning produced via static analysis is a pair (l_1, l_2) of control locations in different threads where the same shared variable is accessed with at least one of the access being a write operation and disjoint sets of locks are held. In order to decide whether (l_1, l_2) is a true date race we have to decide whether there exists a reachable global state of the given program with thread T_i at control location l_i.

We compare the time taken and memory used for MPOR [5] and SIR. For each of the six drivers, the property checked is reachability of control locations corresponding to data race warnings. Columns 1 and 2 report the total number and the number of relevant shared variables, respectively. Here a shared variable is said to relevant is there is a def-use chain starting at some write of v and leading to a conditional statement of some thread. Clearly we need to consider conflicts only for the relevant shared variables. Note that typically, the number of relevant shared variables is considerably less than the total number of shared variables thereby pointing to the utility of SIR. Column 3 gives the time taken for transaction graph construction using our new SIR algorithm. Note that the overhead of this step is small. Also, for examples that contain no relevant shared variables, e.g., *raid*, this step is unnecessary as we know a priori that only one interleaving need be explored. The model checking statistics for MPOR and SIR are shown in columns 4-5 and 6-7, respectively. Clearly, both the time taken and memory used when applying SIR is significantly less than when MPOR is used. Our experiments were conducted on a workstation with 2.8 GHz Xeon processor and 4GB memory.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
2. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among the variables of a program. In: ACM POPL, pp. 84–97 (January 1978)
3. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
4. Kahlon, V., Sankaranarayanan, S., Gupta, A.: Semantic Reduction of Thread Interleavings in Concurrent Programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 124–138. Springer, Heidelberg (2009)
5. Kahlon, V., Wang, C., Gupta, A.: Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 398–413. Springer, Heidelberg (2009)
6. Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and Accurate Static Data-Race Detection for Concurrent Programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 226–239. Springer, Heidelberg (2007)
7. Miné, A.: A New Numerical Abstract Domain Based on Difference-Bound Matrices. In: Danvy, O., Filinski, A. (eds.) PADO-II. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
8. Peled, D.: Combining partial order reductions with on-the-fly model checking. In: Formal Aspects of Computing, vol. 8, pp. 39–64 (1996)
9. Valmari, A.: A stubborn attack on state explosion. Formal Methods in System Design 1(4) (1992)

Adaptive Task Automata: A Framework for Verifying Adaptive Embedded Systems

Leo Hatvani, Paul Pettersson, and Cristina Seceleanu

Mälardalen University, 721 23, Västerås, Sweden
{leo.hatvani,paul.pettersson,cristina.seceleanu}@mdh.se

Abstract. We present a framework for modeling and analysis of adaptive embedded systems, based on the model of timed automata with tasks. The model is extended with primitives allowing modeling of adaptivity, by testing the potential schedulability of a given task, in the context of the set of currently enqueued tasks. This makes it possible to describe adaptive embedded systems, in which decisions to admit further tasks or take other measures of adaptivity is based on available CPU resources, external, or internal events. We show that this model can be encoded in the framework of timed automata, and hence that the problem is decidable. We also validate the framework, by using the Uppaal tool.

1 Introduction

Adaptive embedded systems are embedded systems that must be capable of dynamic reconfiguration, to adapt to e.g., changes in available resources, user- or application-driven mode changes, or modified quality of service requirements. The possibility to adapt provides flexibility that extends the area of operation of embedded systems and potentially reduces resource consumption, but also poses challenges in many aspects of systems development, including system modeling, scheduling, and analysis.

In embedded systems, tasks are usually assumed to execute periodically according to classical real-time scheduling methods, such as rate monotonic scheduling, other fixed priorities, earliest deadline first, or first-in first-out [5]. For systems with non-periodic tasks or non-deterministic task behaviors fewer general results exist. Automata models have been proposed to relax some of the assumptions on the arrival patterns of tasks. In the model of *task automata* (or timed automata with tasks) [8,10], the release patterns of tasks are modeled using *timed automata* [1], such that a set of tasks with known parameters is released at the time point an automaton location is reached. It has been shown that the corresponding schedulability problem for this bigger class of possible release patterns is decidable, i.e., the problem of checking if, for all possible traces of a task automata, the tasks released are schedulable (or not), assuming a given scheduling policy. It has also been shown how to generate code from task automata, such that a modeled system can be realized on a hardware platform running e.g., WxWorks [3,4]. The theory is implemented in the Times tool [2].

J, de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 115–129, 2012.

On the another hand, many results exist for formal verification of adaptive embedded system models specified in high level languages such as UML Statecharts, as enumerated by Schaefer [13]. Another set of results describes application of formal verification of schedulability to: multiprocessor systems [14], satellite systems [11], or providing generalized frameworks for schedulability analysis [7]. All of these studies have one thing in common: the non-schedulability of the system can be determined only after a task misses its deadline, and thus the information is not present soon enough, such that it can be used to avoid entering such state.

In this work, we propose a framework for modeling and analysis of *adaptive real-time embedded systems*, based on the model of task automata, and assuming a single CPU preemptive environment. We extend the model with primitives allowing modeling of adaptivity based on the schedulability of the set of currently released tasks (i.e., the ready queue), if further tasks are released. In particular, we propose to add a schedulability predicate that can be used as a conjunct of a timed automaton guard. The predicate evaluates to true at a given time point, iff the current ready queue, extended with zero or more specified tasks, is schedulable with a given scheduling policy. This allows for modeling of e.g., adaptive embedded systems in which decisions to admit further tasks are based on available CPU resources, or systems in which tasks with high quality of service can occasionally be replaced with alternative lower quality tasks, when the CPU load is too high.

	P	T	D	C
t_0	3	10	10	5
t_1	2	15	6	4
t_1'	1	–	6	1

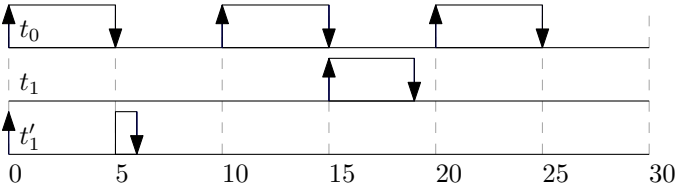


Fig. 1. A trace of a task set with adaptable task t_1

As a small example of the proposed model, consider a system with two tasks t_0, t_1, and t_1', where t_1' is a version of t_1 with lower quality of service, which requires less CPU time. The task parameters are given in Fig. 1: P is priority, T is period, D is deadline, and C is computation time. Since $P_0 > P_1 > P_1'$, task t_0 will be executed periodically without being preempted. We assume t_1 will be admitted only if it has a chance to complete before deadline, otherwise t_1' is released. The system is schedulable, and will release t_0 every 10 ms, and will try to release t_1 every 15ms. If t_1 cannot be released at that time point, due to interference from t_0, task t_1' will be released. Modeled in our extended task automata model, we can check schedulability, verify how many times out of k task t_1' replaces t_1, and interpret a simulated trace as static cyclic scheduler for the system.

As our main result, we show that the schedulability problem and other reachability properties of the proposed model are decidable for fixed priority scheduling policies. Our encoding of the problem is based on previous results of Fersman et.al. [8,10], in which it is shown how given task automata can be encoded and

analyzed as a network of timed automata. However, in comparison to the previous work, our type of adaptive systems cannot rely completely on encoding the scheduler and explore the state space to check if the system is schedulable or not. Instead, we need to check in advance if a system is schedulable, or will be schedulable with the potential release of one or several additional tasks.

The rest of this paper is organized as follows: in the next section, we describe preliminaries, in Section 3 adaptive scheduling policies encompassed by the model, and in Section 4 our main result, the encoding. In Section 5, we give some examples, and conclude the paper in Section 6.

2 Preliminaries: Task Automata

Our model of *adaptive task automata* is based on the model of *task automata* (or *timed automata with tasks*) [8,10,12], which extends the model of timed automata with a notion of tasks. A *timed automata* [1] is simply a finite state automata extended with a finite set of real-valued clocks. The edges of timed automata are labeled with Boolean combinations of simple clock constraints, events, and a *reset set* of clocks, specifying a subset of the clocks to be reset when the edge is taken. In the model of task automata, the idea is to associate each location of a timed automaton with a an executable program, called *task*, which is assumed to be released when the location is reached. Each task is assumed to be associated with given parameters such as execution time, hard deadline, priority, etc. It is possible to interpret a task automaton as an abstract model of a running system, in which the underlying timed automata describes the time points at which possible events occur, and the location-associated tasks, triggered by the occurring event.

Syntax. Let $\mathcal{T}$ ranged over by $t_0, \ldots, t_n$ denote a finite set of task types. Each task type may have different instances over time, however, we will assume, without lack of generality, that at each time point there is at most one instance of each task type released. Each task type is associated with a a triple of natural numbers $t_i(C_i, D_i, P_i)$, where C_i is the task's computation time, D_i its relative deadline (relative from the release time point), and P_i its priority. Further, let Act ranged over by a, b etc, denote the set of action labels, and $\mathcal{C}$ ranged over by $x_0, \ldots, x_n$ the finite set of real-valued clocks. We use $\mathcal{B}(\mathcal{C})$ ranged over by g to denote the set of conjunctive formulas of constraints, called clock constraints, of the form $x_i \sim n$ and $x_i - x_j \sim m$, where $\sim \in \{\leq, <, >, \geq\}$, and n and m are natural numbers.

Definition 1. *[10] A* task automaton *over* Act, $\mathcal{C}$, *and* $\mathcal{T}$ *is a tuple* $\langle L, l_0, E, I, M \rangle$, *where* L *is a set of location ranged over by* $l_0, \ldots l_n$, $l_0 \in L$ *is the initial location,* $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times \text{Act} \times 2^{\mathcal{C}} \times L$ *is the set of edges,* $I : L \mapsto \mathcal{B}(\mathcal{C})$ *is a function assigning each location with a location invariant, and* $M : L \hookrightarrow \mathcal{T}$ *is a partial function assigning locations with tasks.* □

Semantics. Like in standard timed automata, a task automaton may perform two types of actions. A *delay transition* corresponds to progression of time and execution of the released task with the highest priority, and idling lower priority tasks waiting to run. An *action transitions* corresponds to taking an enabled edge (one whose guard evaluates to true given the current clock values), and (possibly) releasing a task associated with the location reached.

A state of a task automaton is a triple $\langle l, u, q\rangle$, where l is the current control location, $u : \mathcal{C} \mapsto \mathbb{R}_{\geq 0}$ is a function mapping clocks to non-negative real values, and q is the current ready queue of tasks. The task queue is formed as: $[t_i(c_i, d_i), \ldots, t_j(c_j, d_j)]$, where t_i is the task type, c_i is the remaining computation time, and d_i the relative deadline. A scheduling function, such as fixed priority or earliest deadline first, is a function Sch sorting the task queue w.r.t. the task parameters. For instance, $[t_1(1, 2), t_2(2.5, 4)]$ is sorted according to fixed priority, if $P_1 > P_2$. Note that a scheduling policy can be either preemptive or non-preemptive, depending on whether the first queue position can be changed (preemptive) or not (non-preemptive).

To define the semantics, we also need a function $\mathsf{Run}_{\mathsf{Sch}}$ that takes a task queue q and a non-negative real-number δ, and returns the result of executing q for δ time units, with the given scheduling function Sch (e.g., $\mathsf{Run}_{\mathsf{FPS}}([t_1(1, 2), t_2(2.5, 4)], 2) = [t_2(1.5, 2)]$, for a fixed priority scheduling fuction $\mathsf{Run}_{\mathsf{FPS}}$).

Definition 2. *[10] Given a task automata $\langle L, l_0, E, I, M\rangle$ with an initial state $\langle l_0, u_0, q_0\rangle$, and a scheduling strategy Sch, the semantics is a transition system defined as:*

- $\langle l, u, q\rangle \xrightarrow{a}_{\mathsf{Sch}} \langle l', r(u), \mathsf{Sch}(M(l') :: q)\rangle$ *if* $l \xrightarrow{g,a,r} l' \in E$ *and* $u \models g$
- $\langle l, u, q\rangle \xrightarrow{\delta}_{\mathsf{Sch}} \langle l, u \oplus \delta, \mathsf{Run}_{\mathsf{Sch}}(q, \delta)\rangle$ *if* $(u \oplus \delta) \models I(l)$

where $r(u)$ is 0 for all $x_i \in r$ and $u(x_i)$ otherwise, $t :: q$ is the result of merging t with q, and $u \oplus \delta$ is the result of adding δ to all clock values in u. □

Schedulability. Verification problems of the above model, with non-preemptive and preemptive tasks, have been already investigated in [10,12]. Here we briefly review the notions of reachability and schedulability. A state $\langle l, u, q\rangle$ is *reachable* with a given scheduling policy Sch, if $\langle l_0, u_0, q_0\rangle\ (\longrightarrow_{\mathsf{Sch}})^* \langle l, u, q\rangle$, where $\longrightarrow_{\mathsf{Sch}}$ is $\xrightarrow{a}_{\mathsf{Sch}}$ or $\xrightarrow{\delta}_{\mathsf{Sch}}$. Further, a state $\langle l, u, q\rangle$ with $q = [t_0(c_0, d_0), \ldots, t_n(c_n, d_n)]$ is defined as *deadline-missed*, if there is some $0 \leq i \leq n$ such that $c_i > 0$ and $d_i \leq 0$. A task automaton A is defined to be *non-schedulable with* Sch iff a deadline-missed state is reachable with Sch. Otherwise, A is considered to be *schedulable with* Sch. In general, A is said to be *schedulable* if it is schedulable with some scheduling strategy Sch. The problem of checking schedulability of task automata with preemptive tasks is proven to be decidable in [10].

3 Adaptive Task Automata

In this section, we describe the model of *adaptive task automata*, which extends the model of timed automata for adaptivity. Our aim is to enable modeling of adaptivity based on the schedulability of the set of currently released tasks,

and the effect of potentially releasing additional tasks for execution. In terms of modeling, the extension consists of a set of predicates for schedulability test, which can be used in conjunction with other guards on edges of task automata. As a main result of this paper, we will also show how the resulting model can be encoded as timed automata, and hence, that reachability and schedulability checking are decidable.

Definition 3. *Given a task automaton state* $\langle l, u, q\rangle$*, with* $q = [t_0(c_0, d_0), \ldots, t_n(c_n, d_n)]$*, and two distinct tasks,* t_i *and* t_j*, let* $\mathcal{P}$ *be the set of predicates* $\{inqueue/1, sched/1, sched/2\}$ *satisfied as follows:*

$\langle l, u, q\rangle \models inqueue(t_i)$ *if* $t_i \in q$

$\langle l, u, q\rangle \models sched(t_i)$ *if* $(\sum_{j=0}^{i} c_j) \leq d_i \wedge inqueue(t_i) \vee \langle l, u, \mathsf{Sch}(t_i :: q)\rangle \models sched(t_i) \wedge \neg inqueue(t_i)$

$\langle l, u, q\rangle \models sched(t_i, t_j)$ *if* $inqueue(t_i) \wedge \langle l, u, \mathsf{Sch}(t_j :: q)\rangle \models sched(t_i)$

□

We say that t_i is *active* in state $\langle l, u, q\rangle$ if $\langle l, u, q\rangle \models inqueue(t_i)$. In the rest of the paper, we will omit $\langle l, u, q\rangle$ if the context is obvious. Intuitively, $sched(t_i)$ is true in a state, if t_i will meet its deadline, given that q is executed according to Sch. We say that t_i is *schedulable* if $sched(t_i)$. Similarly, $sched(t_i, t_j)$ is true in a state, if t_i is schedulable even if t_j is released (added to q).

We now define the model of adaptive task automata. Let $\mathcal{B}(\mathcal{P} \cup \mathcal{C})$ denote the set of conjunctive formulas of clock constraints in $\mathcal{B}(\mathcal{C})$, and predicates in $\mathcal{P}$.

Definition 4 (Adaptive Task Automata). *An* adaptive task automaton *over* Act*,* $\mathcal{C}$*, and* $\mathcal{T}$ *is a tuple* $\langle L, l_0, E', I, M\rangle$*, where* L*,* l_0*,* I*,* M *are defined as in task automata in Definition 1. The set of edges is defined as:* $E' \subseteq L \times \mathcal{B}(\mathcal{P} \cup \mathcal{C}) \times \text{Act} \times 2^{\mathcal{C}} \times L$*.* □

Hence, the set of guards of the edges is extended to conjunctions of clock constraints and the predicates of Definition 3.

Example 1. The adaptive task automaton shown in Fig. 2 describes the release pattern of the task t_1 and corresponding backup task t_1' from Fig. 1. The automaton consists of a clock x, and three states: Start, Release t_1, and Release t_1'. The edge from state Start to the states releasing tasks t_1 or t_1' is immediate, given the invariant $x \leq 0$ of state Start. The choice of the next state is regulated by the evaluation of the respective guards on the edges, $sched(t_1)$ or $sched(t_1') \wedge \neg sched(t_1)$, respectively. Once one of the Release $\{t_1, t_1'\}$ states is entered, the corresponding task is released, and the automaton spends the rest of the period in that state, before returning to start and resetting the clock x. Note that a third edge from Start to an error location, taken in case when none of the alternatives can be released, has been omitted from the figure for simplicity.

Derived predicates. The predicates defined above can be used to derive several other useful predicates, including:

- $sched\_all = (\bigwedge_i inqueue(t_i) \Rightarrow sched(t_i))$,
- $sched\_all(t_i) = (\bigwedge_j inqueue(t_j) \Rightarrow sched(t_j, t_i))$.

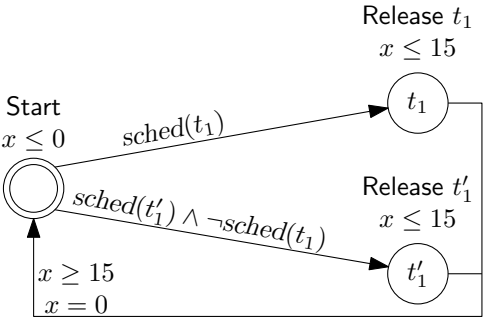


Fig. 2. Adaptive task automata for the task t_1 from the Example 1

The predicate *sched_all* evaluates to *true*, in case all tasks in the queue are schedulable, assuming scheduling policy Sch. The second predicate holds if all the tasks in the queue are schedulable, if task t_i is released. We will make use of the above derived predicates in an example presented in Section 5.

4 Encoding of the Adaptive Task Automata

In this section, we present an encoding of the task release automata, the scheduler, and the task queue, as timed automata models. The encoding is presented in terms of the variables that are used to model the execution of tasks. Based on these variables, the predicate *sched*() is encoded, and finally, an encoding of a fixed priority scheduler is presented.

Modeling a task set execution in timed automata requires tracking of several values for each executed task instance. To establish if a task has executed in time, we keep track of the amount of time that the task has been executing, and the amount of time that has passed since the task has been released. By using these values, and comparing them to the computation times and relative deadlines of the tasks, we can establish if a task is able to complete successfully, or not.

Our encoding is based on, and combines ideas introduced by Fersman et al. [8,9]. The following variables are used for each task t_i:

- c_i - a clock that resets every time the predicate $(\exists t_j \mid inqueue(t_j) \wedge P_j \geq P_i)$ changes value from false to true, where P_i and P_j are priorities of tasks t_i and t_j respectively;
- d_i - a clock reset when the task t_i is released;
- r_i - an integer variable (of bounded domain) that contains a sum of the computation times C_i of all tasks of higher or equal priority to task t_i, which have been released since c_i has been last reset.

The use of these variables will be exemplified on the scenario illustrated in Fig. 3. Four task instances are released: t_1 (at time point 4), t_2 (at time point 1), and

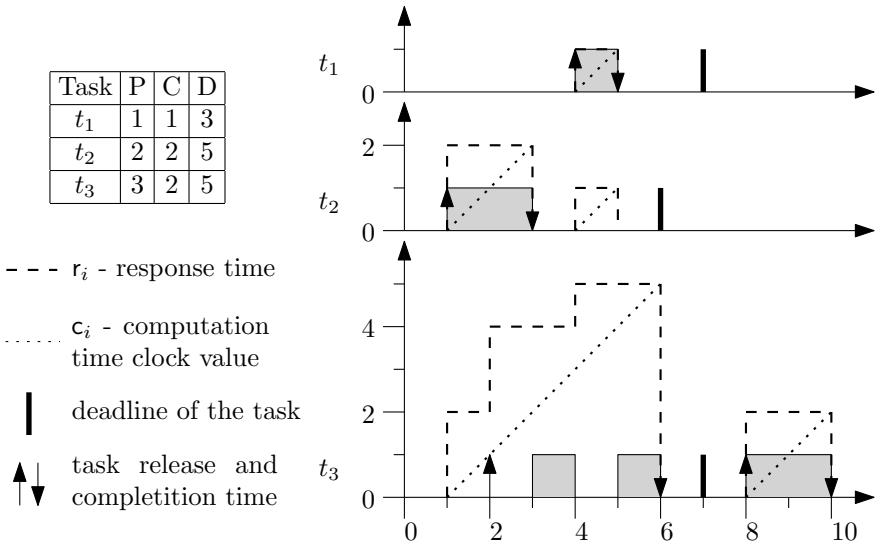


Fig. 3. Tracking of essential variables for each task

t_3 (at time points 2 and 8). The task parameters and the values of variables r_i, and clocks c_i, over time, are also given in the figure. Clocks d_i are left out for clarity, but the point where they would become equal to the corresponding value D_i is marked with thick vertical bars.

The variables and clocks of all tasks are reset at the release of the first task t_2, at time point 1. As t_2 is released, its computation time (2) is added to all the r_i of tasks with lower or equal priority to t_2, i.e., $\mathsf{r}_2 = \mathsf{r}_2 + 2 = 2$ and $\mathsf{r}_3 = \mathsf{r}_3 + 2 = 2$.

A task completes its execution when $\mathsf{c}_i = \mathsf{r}_i$. In our case, this happens first at time point 3, when $\mathsf{r}_2 = \mathsf{c}_2$. However, before this, task t_3 is released at time point 2, so r_3 is increased by 2, the computation time of task t_3. The only clock reset at this time is d_3, to start measuring time until its relative deadline.

At time point 4, task t_1 is released, causing the reset of all its variables, and those of task t_2 (according to how c_i is updated). Variables r_1, r_2, and r_3 are increased by 1 (the computation time of task t_3), to 1, 1, and 5, respectively.

We now focus on task t_3. Observe that the difference $\mathsf{r}_3 - \mathsf{c}_3$ for task t_3 represents the time left until t_3 completes its execution (assuming no higher priority task is released). The time left to its deadline is given by $D_3 - \mathsf{d}_3$. Comparing the two values, we get the amount of time that the task can be delayed without missing its deadline, and hence, as long as the inequality holds, the task will meet its deadline. The values are illustrated in Fig. 4. In fact, at time x, there is enough time to execute a higher priority task for 2 time units, since $\mathsf{r}_3 - \mathsf{c}_3 + 2 \leq D_3 - \mathsf{d}_3$. When task t_1 is later released, we already know that task t_3 can finish at time 6, i.e., 1 time unit before its deadline.

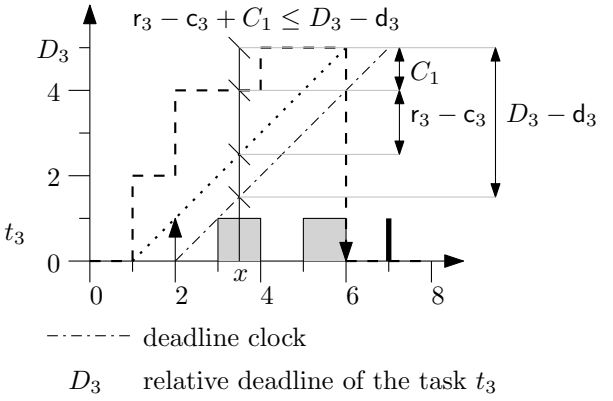


Fig. 4. Visual explanation of the schedulability predicate encoding

4.1 Encoding the Predicate *sched*()

Given the variables introduced above, and given that there is a possible scheduler model (introduced in the next section), we encode the predicate *sched*() as follows:

$$sched(t_i) = \begin{cases} \mathsf{r}_i - \mathsf{c}_i \leq D_i - \mathsf{d}_i & \text{if } inqueue(t_i) \\ \mathsf{r}_i - \mathsf{c}_i + C_i \leq D_i - \mathsf{d}_i & \text{if } \neg inqueue(t_i) \wedge P_{run} > P_i \\ C_i \leq D_i & \text{if } \neg inqueue(t_i) \wedge P_{run} < P_i \end{cases}$$

where t_{run} refers to the currently executing task.

The first case has been explained in the previous section, note that it covers all cases where $t_i = t_{run}$, since $inqueue(t_{run})$ is invariantly true. In case the task of interest (t_i) has not been released yet, its computation time is not included in the expression $\mathsf{r}_i - \mathsf{c}_i \leq D_i - \mathsf{d}_i$, so this gives rise to the second case. In case the task is not yet released, and it has higher priority than the currently running task, it will execute immediately, and its schedulability is then only depending on computation time being shorter than the deadline, hence the third case. This case cannot be covered by the second case, since the clocks are considered inactive at this point, and can only be reset and not read.

The implementation of the scheduler requires a strict ordering between the tasks. We have introduced that ordering by assuming unique task priorities. Together with the requirement of single task instance per task, this makes $P_i = P_j$ lead to an error state, and it is therefore not considered.

The derivation of the schedulability predicate that tests the schedulability of task t_i, based on the release of task t_j, can be done from the second case above, by replacing c_i with a new computation time C_j. This provides the following predicate that tests whether the task t_i is schedulable, if task t_j is released:

$$sched(t_i, t_j) = \begin{cases} \mathsf{r}_i - \mathsf{c}_i + C_j \leq D_i - \mathsf{d}_i & \text{if } P_i < P_j \wedge inqueue(t_i) \\ \mathsf{r}_i - \mathsf{c}_i \leq D_i - \mathsf{d}_i & \text{if } P_i > P_j \wedge inqueue(t_i) \\ \text{false} & \text{if } \neg inqueue(t_i) \end{cases}$$

The second case of this predicate holds when the task that we want to release will not influence the measured task.

4.2 Encoding the Fixed Priority Scheduler

We have devised a model of a fixed priority scheduler, to support our approach to the verification of adaptive embedded systems. This encoding enables us to simulate the passage of time in the model, and simultaneously, keep track of response times of tasks in the queue. This is required for an on-line analysis of schedulability. Next, we give the scheduler's encoding high-level description, yet omitting some details due to lack of space.

High Level Description. The model consists of three locations with identified, different roles: Idle, Busy and Error, as shown in the overview Fig. 5. The corresponding locations can also be found in the Fig. 6.

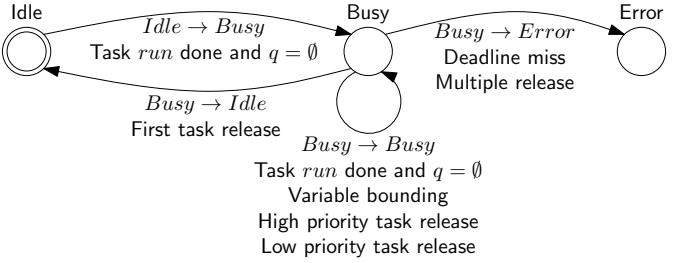


Fig. 5. A high level overview of the scheduler and queue encoding in timed automata

The scheduler and queue timed automaton model starts in the Idle location. As soon as some task is released, the location changes to Busy, and if an error occurs, the model switches to the Error location. Otherwise, the model loops in the Busy location, for as long as there are tasks in the queue. The addition of the Error location makes it possible to easily distinguish between an error in the schedule, and a deadlock in the task release model.

The queue is implemented such that each task t_i has attribute $inqueue_i$. This attribute indicates whether or not the task is present in the ready queue and is therefore directly tied to the $inqueue(t_i)$ predicate.

The initial location of the model is Idle. The model can be in this location only when there are no tasks in the queue, and no task is being executed. As soon as one of the tasks is released (added to the queue), the model changes its location to Busy, via the First task release edge. The consequence of taking this edge is that all of the clocks and variables are reset, in order to initiate a new cycle of execution. After that, the variables related to the release of the first task are updated (detailed explanation of variable updates is presented in section 4.3).

When the automaton is in the Busy location, it means that a task instance is being executed on the CPU. Since the model does not implement any task blocking mechanism, the situation when there are tasks in the queue, but none is executing, cannot occur.

The Busy location wraps in on itself in multiple edges. Many of these edges are restricted to execute at the same time point. This is enforced by an invariant on the Busy location (shown in dotted box in Fig. 6). The model uses variable

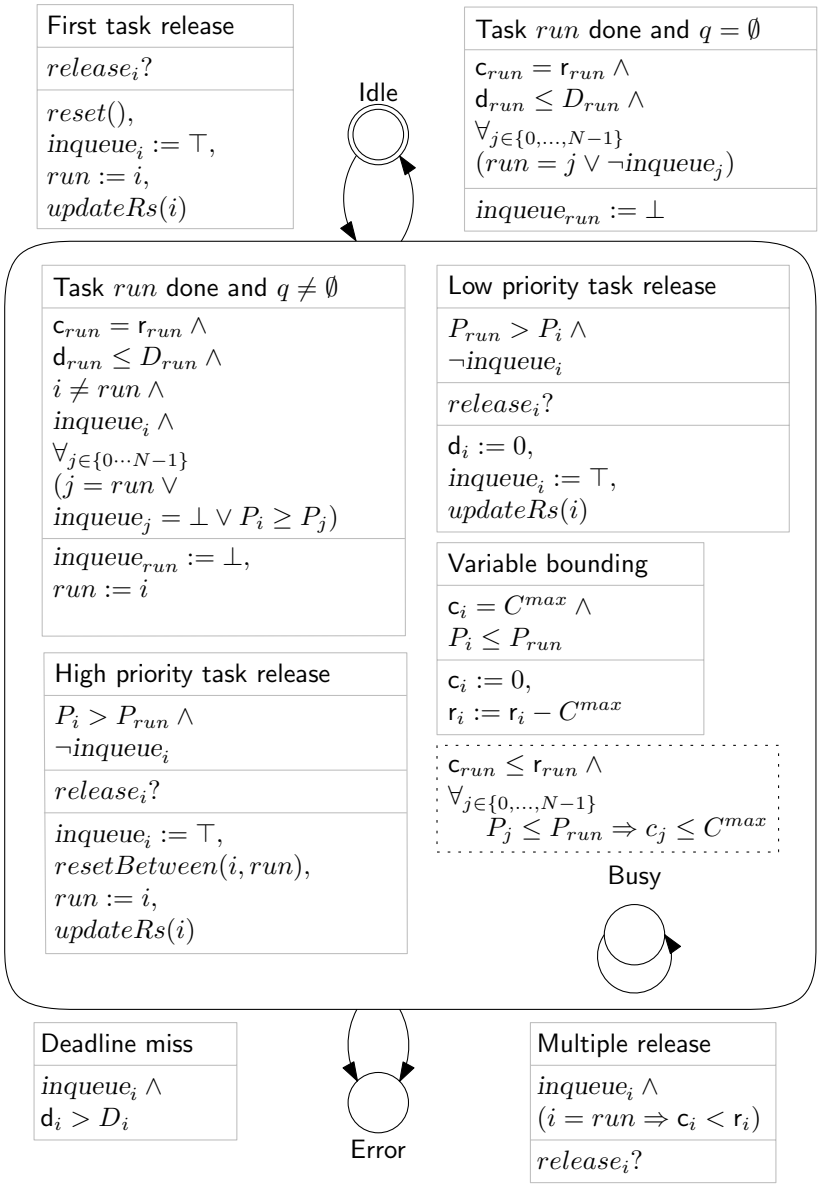


Fig. 6. The full model of scheduler and queue. The boxes represent transitions described by (in order from top to bottom): name, guard predicate, synchronization expression, and assignment. If one of the values is nil it is not shown.

i to represent classes of edges that are instantiated for every task in the task set. For instance, if there are five task types in the task set, there will be five Variable bounding edges, one for each task type. Below, we enumerate the classes of edges looping in the Busy location:

- Task *run* done and $q \neq \emptyset$ - After the current task has completed its execution, this current task, denoted by the value of the *run* variable, is removed from the queue, and a next task is chosen for execution, out of those currently in the queue. The choice of the next task is done by selecting the edge corresponding to a task that has higher priority than all of the other tasks.
- High priority task release - It releases a new task into the queue, which preempts the currently running task. The release changes the status of the currently executing task, sets a new value of the variable *run*, and resets the currently inactive variables that have lower or equal priority than the new task.
- Low priority task release - If the new task is not of higher priority then the currently running task, it is then just placed in the queue. Its variables are already active, so only the deadline clock d_i is reset.
- Variable bounding - Due to the nature of timed automata, it is required that the variables in the model have upper and lower bounds. This process is explained in detail in section 4.3.

Last but not least, we need to consider the possibilities for the model to switch to the Error location. In such a case, there are two classes of edges and, once again, they are iterated over all tasks:

- Deadline miss edge is taken when a task misses its deadline, that is, the deadline clock becomes greater then the value of the relative deadline.
- Multiple release edge is taken when a task is released, but it is already in the queue.

Finally, the edge "Task *run* done and $q = \emptyset$" is taken when the last task in the queue is completed, and there are no more tasks left. We remove the currently running task from the queue and return to the Idle location.

4.3 Variable Bounding

To be able to verify timed automata models, all of the variables, including clocks, have to be bound. To bound variables in this model, we have introduced a loop on the Busy location, named Variable bounding. This loop is executed for each individual task t_i, whenever its total computation time reaches a certain value C^{max}. It reduces the total computation time c_i to zero, and subtracts C^{max} from the corresponding response time variable r_i, thus not influencing the delta $\mathsf{r}_i - \mathsf{c}_i$. By doing this, we ensure that the total computation time is always lower or equal to C^{max}, and that the response time variable is kept bound to $C^{max} + D^{max}$, within a working system. C^{max} can be any value greater or equal to the maximum of computation times in the current task set, and D^{max} is the maximum of deadlines in the task set. If the response time becomes greater than $C^{max} + D^{max}$, we can guarantee that the task will breach its deadline, and the model becomes unschedulable.

Theorem 1. *The problem of checking the schedulability of the system, modeled using adaptive task automata, is decidable.*

Proof Sketch. Due to space limitation, we give only a proof sketch here. In this section, we have presented a way of encoding adaptive task automata using timed automata, featuring a fixed priority scheduler. Since all of the variables in the model are bounded, and the problem of decidability of bounded timed automata with subtraction has been already proved decidable [10], the problem of decidability of checking schedulability in this particular case follows straightforwardly. □

5 Examples

To further illustrate the benefits that the system designers could get from using our model, we have analyzed two example systems, one synthetic, and one based on real world ideas.

5.1 Admission Control - A Synthetic Example

This example demonstrates the usage of the *sched_all*(t_i) predicate, for a given task t_i. We assume a system with two tasks, t_1 and t_2, where each has an alternative version of itself, t'_1 and t'_2, respectively. The task parameters are shown in Fig. 7; parameter J represents the task's jitter value. For instance, the task t_1 will be released every 10 time units, but can be up to 2 time units late.

Task	P	C	D	T	J
t_1	4	5	5	10	2
t'_1	3	3	5	–	2
t_2	2	3	7	10	5
t'_2	1	2	7	–	5

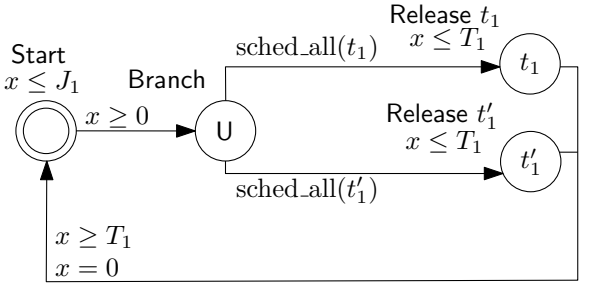


Fig. 7. Task set and adaptive task automata model for the synthetic example

Fig. 7 shows the task automaton corresponding to t_1; the one of t_2 is similar, hence we omit it. For the task t_1, the task automaton checks whether all of the other tasks in the system are schedulable if the task t_1 is released. If the tasks are not schedulable, it tries to release the alternative variant of the task: t'_1. The two task automata instances are modeled as timed automata, and communicate with the scheduler via channels. The order between the preferred and alternative variant of the tasks, respectively, is ensured by using channel priorities [6]. For these models, we have proven that the system would never run into the **Error** state of the scheduler, and that (all of) the variants of the tasks will be eventually released. Proving that the system will never get into the Error state is the most demanding on the UPPAAL prover, and it required about 0.08 seconds CPU time, and 42MB memory on a dual-core 3.0GHz CPU, equipped with 4GB of RAM.

5.2 Smartphone Task Management Example

The second example has been adapted from an idealized smartphone operating system. Modern smartphone devices support multitasking, yet have quite limited resources available for realizing their functionality. We propose a scheduler-level solution that enables a phone to adapt to the current situation fluently, by dynamically restricting the quality of service provided to the user.

The basic assumption is that the software in the smart phone is being executed in cycles. A series of short tasks that handle different applications are being executed each cycle. The applications that we have chosen for this example are: phone call, video call, and multimedia. The user can turn any of these applications on, or off, at arbitrary moments. The switch status of the application will not be immediately reflected in the active task set, but the task set will change during the next cycle, instead.

Table 1. Set of tasks for the smartphone example

	P	T	D	C	Description
t_{cl}	5	10	10	4	Call
t_{vc}	4	10	10	3	Video Chat
t_{mm}	3	10	10	7	Multimedia: max quality
t'_{mm}	2	–	10	4	Multimedia: medium quality
t''_{mm}	1	–	10	3	Multimedia: low quality

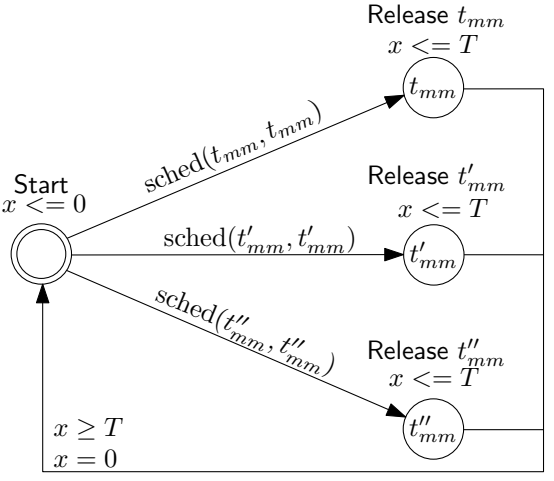


Fig. 8. Adaptive task automaton model for the smartphone example

We have modeled the smartphone as an adaptive task automaton, and then implemented it as timed automata. The system model relies on a fixed priority scheduler. Tasks t_{cl} (phone call), and t_{vc} (video call), are described by "periodic release" automata, whereas task t_{mm} (multimedia) is modeled using the adaptive

task release automaton presented in Fig. 8. The automaton has been modeled using priorities [6], to remove nondeterminism from the execution.

Once the system has been modeled, a full verification of schedulability becomes possible. As previously, verification of not reaching the Error state has been the most demanding and, required about 0.03s and 34MB of RAM memory.

6 Conclusion

In this paper, we have proposed a framework for formal modeling and scheduling of adaptive embedded systems, which relies on a task automata description of the system (tasks and scheduler). In order to check at each task's release time point whether the system is schedulable, or will be with the potential release of other additional tasks, we have introduced a set of schedulability predicates to be used in the guards of the task automata model.

The encodings and on-line schedulability tests that we have devised can be seen as model-level means of predicting, at release time-moments, the timeliness behavior of real-time tasks with very general release patterns, which are stored in the ready queue. Our liberal adaptive task automata model, enhanced with predicates for schedulability test, lets one perform on-line adaptations that decide to admit or not certain tasks, depending on their respective adherence to the desired real-time requirements, that is, meeting their deadlines. The salient result of our work is the decidability of reachability and schedulability of adaptive task automata, by showing that the resulting model can be encoded in the timed automata framework.

The power of our approach resides exactly in the fact that the task selection strategy is specified as a predicate on clocks and integers. As it stands now, that is, assuming fixed priority schedulers, the model is compatible with any scheduler that has fixed ordering between the tasks, once the tasks are released. As with every formalized approach, there are some potentially useful-to-solve unexplored issues, which need further attention. For instance, it would be interesting to check on the consequences of allowing a task set to run, even if, based on our schedulability tests, we decide that it misses its deadline at the current time point. Another problem that deserves investigation is the possibility of releasing more than one task at a time, and verify the resulting model.

We also consider to extend the method to cater also for other schedulers than fixed-priority, for instance, Earliest-Deadline-First (EDF) schedulers. Nevertheless, although, as for now, our technique is restricted to fixed-priority schedulers, it can already decide on task executions at run-time, but has also the potential of manipulating the queue of released tasks, in the sense of switching ready tasks' priorities, if the case, removing certain tasks from the queue, etc., all based on possible further additions to the schedulability predicates.

The final avenue to explore would be along investigating the efficiency of our approach, when handling real-world industrial case study.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Times: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 60–72. Springer, Heidelberg (2004)
3. Amnell, T., Fersman, E., Pettersson, P., Sun, H., Yi, W.: Code synthesis for timed automata. Nordic Journal of Computing 9(4), 269–300 (2002)
4. Åsberg, M., Nolte, T., Pettersson, P.: Prototyping and code synthesis of hierarchically scheduled systems using times. Journal of Convergence (Consumer Electronics) 1(1), 77–86 (2010)
5. Buttazzo, G.C.: Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications. Kulwer Academic Publishers (1997)
6. David, A., Håkansson, J., Larsen, K., Pettersson, P.: Model Checking Timed Automata with Priorities Using DBM Subtraction. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 128–142. Springer, Heidelberg (2006)
7. David, A., Illum, J., Larsen, K., Skou, A.: Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1. CRC Press (2011/12/27 (2009)
8. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)
9. Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Schedulability analysis of fixed-priority systems using timed automata. Theor. Comput. Sci. 354, 301–317 (2006)
10. Fersman, E., Pettersson, P., Yi, W.: Timed Automata with Asynchronous Processes: Schedulability and Decidability. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 67–82. Springer, Heidelberg (2002)
11. Mikučionis, M., Larsen, K., Rasmussen, J., Nielsen, B., Skou, A., Palm, S., Pedersen, J., Hougaard, P.: Schedulability Analysis Using Uppaal: Herschel-Planck Case Study. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 175–190. Springer, Heidelberg (2010)
12. Norström, C., Wall, A., Yi, W.: Timed automata as task models for event-driven systems. In: Sixth International Conference on Real-Time Computing Systems and Applications, RTCSA 1999, pp. 182–189 (1999)
13. Schaefer, I.: Integrating Formal Verification into the Model-Based Development of Adaptive Embedded Systems. Ph.D. thesis, TU Kaiserslautern, Kaiserslautern, Germany (October 2008) ISBN 978-3-89963-862-2
14. Yu, F., Li, G., Xiong, N.: Schedulability analysis of multi-processor real-time systems using uppaal. In: 2010 2nd International Conference on Information Science and Engineering (ICISE), pp. 1–6 (December 2010)

Verified Resource Guarantees for Heap Manipulating Programs

Elvira Albert[2], Richard Bubel[1], Samir Genaim[2], Reiner Hähnle[1], and Guillermo Román-Díez[3]

[1] CSE, Chalmers University of Technology, Sweden
[2] DSIC, Complutense University of Madrid (UCM), Spain
[3] DLSIIS, Technical University of Madrid (UPM), Spain

Abstract. Program properties that are automatically inferred by static analysis tools are generally not considered to be completely trustworthy, unless the tool implementation or the results are formally verified. Here we focus on the formal verification of *resource guarantees* inferred by automatic cost analysis. Resource guarantees ensure that programs run within the indicated amount of resources which may refer to memory consumption, to number of instructions executed, etc. In previous work we studied formal verification of inferred resource guarantees that depend only on integer data. In realistic programs, however, resource consumption is often bounded by the size of *heap-allocated* data structures. Bounding their size requires to perform a number of structural heap analyses. The contributions of this paper are (i) to identify what exactly needs to be verified to guarantee sound analysis of heap manipulating programs, (ii) to provide a suitable extension of the program logic used for verification to handle structural heap properties in the context of resource guarantees, and (iii) to improve the underlying theorem prover so that proof obligations can be automatically discharged.

1 Introduction

Formally proving the correctness of software can be crucial for many applications, e.g., in safety-critical systems. There are two possible approaches to certifying the correctness of software, (1) either perform full-blown verification of the correctness of the system or (2) alternatively validate its results for every execution. In the case of static analyzers, the first alternative is a daunting task, among other things, because of the sophisticated algorithms used for the analysis and their evolution over time. In this paper, we adopt the second alternative based on constructing a validating tool [14] which, after every run of the analyzer, formally (and automatically) confirms that the results are correct and, optionally, generates correctness proofs. Such proofs can then be translated to independently checkable *certificates* in the proof-carrying code style [6,13].

Resource usage analysis aims at (over-)approximating the amount of resources (time, memory, etc.) required to run a program in terms of its input arguments. COSTA [1,2] is a cost analyzer which allows the user to select a particular resource

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 130–145, 2012.

(among those available in the system) and automatically generate *resource usage upper bounds* from Java bytecode (and hence Java) programs. Correctness of the techniques that COSTA implements is proven at the theoretical level, but the tool has not been formally verified. Thus, there is no guarantee that correctness is realized by the implementation. In recent work [3], we have proposed a fully automatic process of obtaining *verified resource guarantees* by using KeY [5], a state-of-the-art theorem prover for Java programs, for verifying that the upper bounds inferred by COSTA are correct. In essence, the COSTA and KeY systems cooperate in such a way that KeY produces formal correctness proofs for the different intermediate results used to obtain the upper bounds. When the resource guarantees depend only on data of integer type, this cooperation results in a fully automatic tool for producing verified resource guarantees.

However, it is often the case that resource guarantees depend on the structural properties of *dynamically allocated data*, e.g., the resource consumption of executing a loop that traverses a list is typically a function of the length of such a list. Resource analysis needs to keep track of how the size of data structures changes along the execution. For this purpose, COSTA integrates as an additional component the *path-length* analysis [17]. The path-length of a non-cyclic data structure is the length of the maximal path starting from the root, i.e., its depth. Inferring the path-length property also requires proving *acyclicity* of data structures and keeping track of possible *sharing* between pointers.

The main achievement of this paper is the extension of [3] to handle heap manipulating programs. In particular: (1) we identify the structural properties inferred by COSTA which need to be verified and extend the *Java Modeling Language* (JML) by suitable new constructs; (2) we extend the program logic used during verification by additional theories for structural heap properties including acyclicity or disjointness of heap regions. Extensive work with implementation and improvement of the proof-search strategies for the newly introduced theories was required to achieve a high degree of automation; (3) we formalize faithfully the notion of maximal path-length of an acyclic data structure in KeY's logic. This theory is equipped with lemmas that match the requirements of the path-length analysis performed in COSTA; and (4) realizing the cooperation between COSTA and KeY has required a number of non-trivial extensions of both systems.

The paper is organized as follows: Sec. 2 recalls the framework of [3]; Sec. 3 presents the additional components that need to be verified for carrying out the extension; Sec. 4 describes how the KeY logic has been extended to express and verify structural heap properties and path-length assertions; experimental results are presented in Sec. 5; and Sec. 6 concludes and discusses related work.

2 The Framework: Verification of Resource Guarantees

In this section we review the verification framework for upper bounds (UBs) as proposed in [3] which does not take heap-allocated data structures into account. Sec. 2.1 describes the components involved in a resource guarantees analysis while Sec. 2.2 details the formal verification of these components with KeY.

```
1 void scoreBoard(int[][] v) {
2   //@ ghost int v_len =v.length
3   int i=0, j=0;
4   //@ assert (j=0 ∧ i=0 ∧ v.length=v_len)
5   //@ ghost int i1=i, j1=j, v_len1=v.length
6   //@ ghost int i2=i, j2=j, v_len2=v.length
7   //@ decreases ( (v.length − i) ≥ 0 ? (v.length − i) : 0)
8   //@ loop_invariant (i2=0 ∧ i2=i ∧ v_len2≥0) ∨ (i2=0 ∧ i≥1 ∧ v_len2≥i)
9   while (i < v.length) {
10    j=0;
11    //@ assert (v_len1>i ∧ i1=i ∧ j=0)
12    //@ ghost int i3=i, j3=j, v_len3=v.length
13    //@ ghost int i4=i, j4=j, v_len4=v.length
14    //@ decreases ( (i − j) ≥ 0 ? (i − j) : 0)
15    //@ loop_invariant (j4=0 ∧ j4=j ∧ i4=i) ∨ (j4=0 ∧ j≥1 ∧ i4≥j ∧ i4=i)
16    while (j < i) {
17      v[i][j]=i + j;
18      j++;
19      //@ assert (j=j3+1 ∧ i3=i);
20      //@ set i3=i, j3=j, v_len3=v.length
21    }
22    i++;
23    //@ assert (v.length1>i ∧ i=i1+1)
24    //@ set i1=i, j1=j, v_len1=v.length
25  }
26 }
```

Fig. 1. COSTA's output for a simple example working on integer data

2.1 Inference of Resource Guarantees

Cost analyzers [1,2] usually infer UBs for each iterative and recursive construct (loops) and then compose the results in order to obtain UBs for the methods of interest. W.l.o.g., we focus on polynomial UBs which are the result of composing simple loops, but the same components are used to infer UBs for programs with logarithmic and exponential complexities. Intuitively, in order to infer an UB for a single loop, we infer an UB A on the worst-case cost of a single execution of its body and an UB I on the number of iterations that it can perform. Then, $A * I$ is an UB for the loop. To infer A and I COSTA relies on the program analysis components described below that provide the necessary information. The results are provided by COSTA as JML annotations that KeY will attempt to verify.

Ranking functions. For each loop, COSTA infers as UB on the number of iterations a linear function I from the loop variables to $\mathbb{N}$ which is strictly decreasing at each iteration. Ranking functions are of the form $\mathsf{nat}(\ell)$, where $\mathsf{nat}(\ell) = \max(0, \ell)$, which can be translated to the JML annotation "*//@ decreasing* $\ell > 0 \ ? \ \ell : 0$".

Example 1. Consider the method `scoreBoard()` given in Fig. 1, where two nested loops are used to initialize some matrix values. For the inner loop COSTA infers at line 14 the ranking function $f(i,j) = \mathtt{nat}(i-j)$ which safely bounds the number of iterations. For the outer loop, the number of iterations is bounded by the ranking function that appears in line 7 which involves the length of the array.

Loop invariants. Loop invariants, together with size relations, are needed to compute the worst-case cost A of executing one loop iteration. For each loop in the program, COSTA infers an invariant φ that involves the loop variables $\bar{v}$ and auxiliary variables $\bar{w}$ such that each w_i represents the initial value of v_i. The JML annotation for this invariant consists of one line defining all $\bar{w}$ as ghost variables ("*//@ ghost int* $w_1 = v_1$*;...; int* $w_n = v_n$", lines 6, 13 in Fig. 1) and one line for the loop invariant ("*//@ loop_invariant* φ", lines 8, 15 in Fig. 1).

Example 2. Consider the invariant for the outer loop at line 8. The left disjunct corresponds to first visit to that program point, and the right disjunct to visit it after executing the loop body at least once. Note that separating the invariant into these two cases results in a more precise UB, and in addition helps KeY in verifying the invariant. We declared as ghost variable in line 6 such that i_2,j_2 and v_len_2 correspond to the initial value of `i`, `j` and `v.length` when entering the loop for the first time. The invariant states that `i` is always smaller than or equal to the initial value of `v.length` ($i \leq v\_len_2$) This is essential to bound the worst-case cost of the loop, since the cost of each iteration depends on `i`.

Size relations. Given a fragment of code (a scope), COSTA infers size relations between the values of the variables at a certain program point of interest within the scope and their initial values when entering the scope. This allows composing the cost of the different code fragments. In particular, for each loop (or method call), COSTA infers the relation φ between the values of variables before a loop (or call) entry and the entry of its parent scope. Suppose that the loop (or call) is at line L_l, its parent scope starts at line L_p, $\bar{v}$ are the variables of interest at line L_l, and $\bar{w}$ represent their values at line L_p. Then we add the JML annotation "*//@ ghost int* $w_1 = v_1$*;...; int* $w_n = v_n$;" immediately after line L_p to capture the values of $\bar{v}$ at line L_p, and the JML annotation "*//@ assert* φ" immediately before line L_l to state that the relation φ must hold at the program point.

Example 3. Let us demonstrate the need for size relations: (1) during cost analysis, the cost of the outer loop is inferred first in terms of the values of `i` and `v.length` before entering the loop, and later is transformed to be in terms of the length of the input array. For this, COSTA uses the size relation at line 4 which relates the values at that program point to those at line 2 using the corresponding ghost variables; (2) similarly, the cost of the inner loop is first inferred in terms of the values of `i` and `j` before entering the loop, and later is transformed to be in terms of their values when entering the outer loop. Assuming that i_1, j_1 and v_len_1 are respectively the value of `i`, `j`, and `v.length`, line 11 includes the size relation required to do such transformation. Note that since these code

fragments appear inside a loop, the values of i_1, j_1 and v_len_1 should be updated in each iteration. This is done by defining and initializing them at line 5 (for the first iteration) and modifying them in each iteration at the end of the loop (line 24). The size relation at line 23 is used by COSTA to synthesize a ranking function, this also helps KeY in proving that it is indeed a ranking function; and (3) lines 12, 19 and 20 encode the size relation of the inner loop.

Upper Bounds. In the verification phase it suffices to prove the correctness of the inferred ranking functions, loop invariants, and size relations: based on these, it is straightforward to compute an UB for the method by applying parametric integer programming (PIP) to obtain A and then just multiply $I * A$.

Example 4. We start from the innermost loop at line 16. Assuming that executing the condition costs (at most) c_1 instructions, and that the cost of each iteration (i.e., the loop body) is c_2 instructions, then it is clear that $\mathsf{nat}(i_4 - j_4) * (c_1 + c_2) + c_1$ is an UB on the cost of this loop. Next, we move to the outer loop at line 9. Let us assume that the cost of the comparison is (at most) c_3 instructions, the code at line 10 costs c_4 instructions, and the code at line 22 is c_5 instructions. Then, the cost of each iteration of this loop is $c_3 + c_4 + \mathsf{nat}(i_4 - j_4) * (c_1 + c_2) + c_1 + c_5$, where the highlighted subexpression is the cost of the inner loop. Note that each iteration might have a different cost, since $i_4 - j_4$ is not the same for all iterations. The solution is to find the worst-case cost A in terms of v_len_2, i_2, j_2 such that $A \geq i_4 - j_4$ in all iterations. Then, $\mathsf{nat}(v\_len_2 - i_2) * [c_3 + c_4 + \mathsf{nat}(A) * (c_1 + c_2) + c_1 + c_5] + c_3$ is an UB for the loop. To find such A, COSTA solves the PIP problem of maximizing the objective function $i_4 - j_4$ w.r.t. the loop invariant (line 8) and the size relations (line 11) where v_len_2, i_2, j_2 are the parameters. This produces an expression in terms of v_len_2, i_2, j_2 which is greater than or equal to $i_4 - j_4$ in all iterations of the loop. In our example, it is $A = v\_len_2 - 1$. We finally can compute the cost of the `scoreBoard` method. Assume that the cost of line 3 is c_6, then the cost of the method is $c_6 + \mathsf{nat}(v\_len_2 - i_2) * [c_3 + c_4 + \mathsf{nat}(v\_len_2 - 1) * (c_1 + c_2) + c_1 + c_5] + c_3$. We need to express this UB in terms of the input parameter v_len. For this, COSTA maximizes (using PIP) $v\_len_2 - i_2$ and $v\_len_2 - 1$ w.r.t. the size relation at line 4 and, respectively, obtains v_len and $v\_len - 1$. Therefore, $c_6 + \mathsf{nat}(v\_len) * [c_3 + c_4 + \mathsf{nat}(v\_len - 1) * (c_1 + c_2) + c_1 + c_5] + c_3$ is the UB for `scoreBoard`.

2.2 Verification by Symbolic Execution

The program logic used by KeY is *JavaCard Dynamic Logic* (JavaDL) [5], a first-order dynamic logic with arithmetic. JavaDL extends sorted first-order logic by a program modality $\langle\cdot\rangle\cdot$. Let p denote a sequence of executable Java statements and ϕ an arbitrary JavaDL formula, then $\langle\mathsf{p}\rangle\phi$ is a formula which states that program p terminates and in its final state ϕ holds. A typical formula looks like

$$\mathsf{i} \doteq i0 \wedge \mathsf{j} \doteq j0 \rightarrow \overbrace{\langle \mathsf{i=j-i;j=j-i;i=i+j;} \rangle}^{\mathsf{p}} (\mathsf{i} \doteq j0 \wedge \mathsf{j} \doteq i0)$$

where i, j are program variables represented as *non-rigid* constants. Non-rigid constants and functions are state-dependent: their value can be changed by programs. The *rigid* constants $i0, j0$ are state-independent: their value cannot be changed. The formula above says that if program p is executed in a state where i and j have values $i0$, $j0$, then p terminates *and* in its final state the values of the variables are swapped. To reason about JavaDL formulas, KeY employs a sequent calculus whose rules perform *symbolic execution* of the programs in the modalities. Here is a typical rule:

$$\text{ifSplit}\ \frac{\Gamma, b \Longrightarrow \langle \{\mathsf{p}\}\mathsf{rest}\rangle\phi, \Delta \quad \Gamma, \neg b \Longrightarrow \langle \{\mathsf{q}\}\mathsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{if\ (b)\ \{p\}\ else\ \{q\}\ rest}\rangle\phi, \Delta}$$

As values are symbolic, it is in general necessary to split the proof whenever an implicit or explicit case distinction is executed. It is also necessary to represent the *symbolic* values of variables throughout execution. This becomes apparent when statements with side effects are executed, notably assignments. The assignment rule in JavaDL looks as follows:

$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathsf{x} := \mathsf{val}\}\langle \mathsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{x = val;\ rest}\rangle\phi, \Delta}$$

The expression in curly braces in the premise is called *update* and is used in KeY to represent symbolic state changes. An *elementary* update $loc := val$ is a pair of a program variable and a value. The meaning of updates is the same as that of an assignment, but updates can be composed in various ways to represent complex state changes. Updates u_1, u_2 can be composed into *parallel updates* $u_1 \| u_2$. In case of clashes (updates u_1, u_2 assign different values to the same location) a last-wins semantics resolves the conflict. This reflects left-to-right sequential execution. Apart from that, parallel updates are applied simultaneously, i.e., they do not depend on each other. Update application to a formula/term e is denoted by $\{u\}e$ and forms itself a formula/term.

Verifying Size Relations. JML annotations are proven to be valid by symbolic execution. For example, in the method scoreBoard() one starts with execution of the variable declarations. Ghost variable declarations and assignments to ghost variables (*//@ set var=val;*) are treated like Java assignments. If a JML assertion "assert φ;" is encountered during symbolic execution, the proof is split: the first branch must prove that the assertion formula φ holds in the current symbolic state; the second branch continues symbolic execution. In the scoreBoard example, a proof split occurs before entering each loop. This verifies the size relations among variables as derived by COSTA and encoded in terms of JML assertions.

Verifying Invariants and Ranking Functions. Verification of the loop invariants and ranking functions obtained from COSTA is achieved with a tailored loop invariant rule that has a variant term to ensure termination:

$$\text{loopInv}\ \frac{\begin{array}{ll}(i) & \Gamma \Longrightarrow Inv \wedge dec \geq 0, \Delta \\ (ii) & \Gamma, \{\mathcal{U}_A\}(b \wedge Inv \wedge dec \doteq d0) \Rightarrow \\ & \quad \{\mathcal{U}_A\}\langle \mathsf{body}\rangle(Inv \wedge dec < d0 \wedge dec \geq 0), \Delta \\ (iii) & \Gamma, \{\mathcal{U}_A\}(\neg b \wedge Inv) \Longrightarrow \{\mathcal{U}_A\}\langle \mathsf{rest}\rangle\phi, \Delta\end{array}}{\Gamma \Longrightarrow \langle \mathsf{while\ (b)\ \{\ body\ \}\ rest}\rangle\phi, \Delta}$$

Inv and *dec* are obtained, respectively, from the loop_invariant and decreasing JML annotations generated by COSTA. Premise (i) ensures that invariant *Inv* is valid just before entering the loop and that the variant *dec* is non-negative. Premise (ii) ensures that *Inv* is preserved by the loop body and that the variant term decreases strictly monotonic while remaining non-negative. Premise (iii) continues symbolic execution upon loop exit. The integer-typed variant term ensures loop termination as it has a lower bound (0) and is decreased by each loop iteration. Using COSTA's derived ranking function as variant term obviously verifies that the ranking function is correct. The update $\mathcal{U}_A$ assigns to all locations whose values are potentially changed by the loop a fixed, but unknown value. This allows using the values of locations that are unchanged in the loop during symbolic execution of the body.

Contracts. COSTA also infers *contracts* which specify pre- and post-conditions on the input and output arguments of each method. Contracts are useful for modular verification in KeY.

3 Upper Bounds for Heap Manipulating Programs

When input arguments of a method are of reference type, its UB is usually not specified in terms of the concrete values within the data structures, but rather in terms of some *structural* properties of the involved data structures. For example, if the input is a list, then the UB would typically depend on the length of the list instead of the concrete values in the list.

Example 5. Consider the program in Fig. 2 where class `List` implements a linked list as usual. For method `insert`, COSTA infers the UB $c_1 * \mathsf{nat}(x) + c_2$ where x refers to the length of `x`, and c_1/c_2 are constants representing the cost of the instructions inside/before & after the loop. The UB depends on the length of `x`, because the list is traversed at lines 16–19.

The example shows that cost analysis of heap manipulating programs requires inferring information on how the size of data structures changes during the execution, similar to the invariants and size-relations that are used to describe how the values of integer variables change. To do so, we first need to fix the meaning of "size of a data structure". We use the path-length measure which maps data structures to their *depth*, such that the depth of a cyclic data structure is defined to be ∞. Recall that the depth of a data structure is the maximum number of nodes (i.e. objects) on a path from the root to a leaf. Using this size measure, COSTA infers invariants and size relations that involve both integer and reference variables, where the reference variables refer to the depth of the corresponding data structures. Once the invariants are inferred, synthesizing the UBs follows the same pattern as in Sec. 2. In the following, we identify the essential information of the path-length analysis (and related analyses) that must be verified later by KeY.

```
1 //@ requires \acyclic(x)
2 //@ ensures \acyclic(\result)
3 //@ ensures \depth(\result) ≤ \depth(x) + 1
4 public static List insert(List x, int v) {
5     //@ ghost List x0 = x;
6     List p = null;
7     List c = x;
8     List n = new List(v, null);
9     //@ ghost List c0 = c
10    //@ assert \depth(n) = 1 ∧ \depth(c0) = \depth(x0)
11    //@ decreasing \depth(c)
12    //@ loop_invariant \depth(c0) ≥ \depth(c)
13    //@ loop_invariant \acyclic(n) ∧ \acyclic(p) ∧ \acyclic(x) ∧ \acyclic(c)
14    //@ loop_invariant \disjoint({n, x})∧\disjoint({n, c})∧\disjoint({n, p})
15    //@ loop_invariant !\reachPlus(p, x) ∧ !\reachPlus(n, x) ∧ !\reach(n, p)
16    while ( c != null ∧ c.data < v) {
17        p = c;
18        c = c.next;
19    }
20    if ( p == null ) {
21        n.next = x;
22        x = n;
23    } else {
24        n.next = c;
25        p.next = n;
26    }
27    return x;
28 }
```

Fig. 2. The running example, with (partial) JML annotations

3.1 Path-Length Analysis

Path-length analysis is based on abstracting program states to linear constraints that describe the corresponding path-length relations between the different data structures. For example, the linear constraint $x < y$ represents all program states in which *the depth of the data structure to which* `x` *points is smaller than the depth of the data structure to which* `y` *points*. Starting from an initial abstract state that describes the path-length relations of the initial concrete state, the analysis computes path-length invariants for each program point of interest. In order to verify the path-length information with KeY, we have extended JML with the new keyword `\depth` that gives the depth of a data structure to which a reference variable points. In particular, for invariants, size-relations, and contracts, if the corresponding constraints include a variable x, corresponding to a reference variable `x`, we replace all occurrences of x by `\depth(x)`.

Example 6. We explain the various path-length relations inferred by COSTA for the method `insert` of Fig. 2, and how they are used to infer an UB. Due to space

limitations, we only show the annotations of interest. For the loop at lines 16–19, COSTA infers that the depth of the data structure to which `c` points decreases in each iteration. Since the depth is bounded by 0, it concludes that $\texttt{nat}(c)$ is a ranking function for that loop. As a part of the loop invariant, COSTA infers that $c_0 \geq c$ where c_0 refers to the depth of the data structure to which `c` points before entering the loop and c to the depth of the data structure to which `c` points after each iteration. Using this invariant, together with the knowledge that the depth of c_0 equals to the depth of `x`, we have that $c_1 * \texttt{nat}(x) + c_2$ is an UB for `insert` (since the maximum value of `c` is exactly `x`). Another essential relation inferred by the path-length analysis (captured in the `ensures` clause in line 3) is that the depth of the list returned by `insert` is smaller than or equal to the depth of `x` plus one. This is crucial when analyzing a method that uses `insert` since it allows tracking the size of the list after inserting an element.

Path-length relations are obtained by means of a fixpoint computation which (symbolically) executes the program over abstract states. As a typical example, executing `x=y.f` adds the constraint $x' < y$ to the abstract state if the variable `y` *points to an acyclic* data structure, and $x' \leq y$ otherwise. On the other hand, executing `x.f=y` adds the constraints $\bigwedge\{z' \leq z + y \mid$ `z` might share with `x`$\}$ if it is guaranteed that `x` does not become cyclic after executing this statement. This is because, in the worst case, `x` might be a leaf of the corresponding data-structure pointed to by `z`, and thus the length of its new paths can be longer than the old ones at most by y. Obviously, to perform path-length analysis, we require information on (a) whether a variables certainly points to an acyclic data structure; and (b) which variables might share common regions in the heap.

3.2 Cyclicity Analysis

The cyclicity analysis of COSTA [9] infers information on which variables *may* point to (a)cyclic data structures. This is essential for the path-length analysis. The analysis abstracts program states to sets of elements of the form: (1) $\texttt{x}\rightsquigarrow\texttt{y}$ which indicates that starting from `x` one *may* reach (with at least one step) the object to which `y` points; (2) $\circlearrowleft^{\texttt{x}}$ which indicates that `x` *might* point to a cyclic data structure; and (3) $\texttt{x}\diamond\texttt{y}$ which indicates that `x` *might* alias with `y`.

Starting from an abstract state that describes the initial reachability, aliasing and cyclicity information, the analysis computes invariants (on reachability, aliasing and cyclicity) for each program point of interest by means of a fixpoint computation which (symbolically) executes the program instructions over the abstract states. For example, when executing `y=x.f`, then `y` inherits the cyclicity and reachability properties of `x`; and when executing `x.f=y`, then `x` becomes cyclic if before the instruction the abstract state included $\circlearrowleft^{\texttt{y}}$, $\texttt{y}\rightsquigarrow\texttt{x}$, or $\texttt{y}\diamond\texttt{x}$.

On the verification side, to make use of the inferred cyclicity relations, we extend JML by the new keyword `\acyclic` which *guarantees* acyclicity. In contrast to COSTA, JML and KeY use shape predicates with *must*-semantics. Acyclicity information is then added in JML annotations at entry points of contracts and loops where we specify all variables which are guaranteed to be acyclic. For loop

entry points as invariants (as in line 13) and for contracts as pre- and postconditions (as in lines 1, 2). To make use of the reachability relations we extend JML by the new keyword `\reachPlus(x, y)`, which indicates that `y` *must* be reachable from `x` in at least one step, and use the standard keyword `\reach(x, y)` which indicates that `y` *must* be reachable from `x` in zero or more steps (i.e., they might alias). The *may*-information of COSTA about reachability and aliasing is then added as *must*-predicates in JML (in loop entries and contracts) as follows: let A be the set of judgments inferred by COSTA for a given program point, then we add `!\reachPlus(x, y)` whenever $\texttt{x} \rightsquigarrow \texttt{y} \notin A$, and we add `!\reach(x, y)` whenever $\texttt{x} \rightsquigarrow \texttt{y} \notin A \wedge \texttt{x} \diamond \texttt{y} \notin A$ (for example, in line 15).

3.3 Sharing Analysis

Knowledge on possible sharing is required by both path-length and cyclicity analyses. The sharing analysis of COSTA is based on [15] where abstract states are sets of pairs of the form $\texttt{x} \bullet \texttt{y}$ which indicate that `x` and `y` might share a common region in the heap. The sharing invariants are propagated from an initial state by means of a fixpoint computation to the program points of interest. For example, when executing `y=x.f`, the variable `y` will only share with anything that shared with `x` (including `x` itself); on the other hand, when executing `x.f=y`, the variable `x` keeps its previous sharing relations, and in addition it might share with `y` and anything that shared with `y` before.

Obviously, KeY needs to know about the sharing information inferred by COSTA to verify acyclicity and path-length properties. To this end, we extended JML by the new keyword `\disjoint` which states that its argument, a set of variables, *does not share* any common region in the heap (for example, in line 14).

4 Verification of Path-Length Assertions

Structural heap properties, including acyclicity, reachability and disjointness, are essential both for path-length analysis and for the verification of path-length assertions. However, while the path-length analysis performed by COSTA maintains cyclicity and sharing, the complementary properties are used as primitives on the verification side. The reason is that the symbolic execution machinery of KeY starts with a completely unspecified heap structure that subsequently is refined using the inferred information about acyclicity and disjointness. In the following we explain how structural heap properties are formalized in the dynamic logic (JavaCard DL) used in this paper and implemented in KeY [5].

4.1 Heap Representation

First we briefly explain the logical modeling of the heap in JavaCard DL.[1] The heap of a Java program is represented as an element of type *Heap*. The *Heap*

[1] Note that this is *not* the heap model described in earlier publications on KeY such as [5]. In the present paper we use an explicit heap model based on [18].

data type is formalized using the theory of arrays and associates locations to values. A location is a pair (o, f) of an object o and a field f. The *select* function allows to access the value of a location in a heap h by $select(h, o, f)$. The complementary update operation which establishes an association between a location (o, f) and a value val is $store(h, o, f, val)$. To improve readability, when the heap h it is clear from the context, we use the familiar notation $o.f$ and $o.f := val$ instead of select and store expressions. Based on this heap model, we define a rule for symbolic execution of field assignments (cf. the assign rule in Sec. 2.2). It simply updates the global heap program variable with the updated heap object:

$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathsf{heap} := store(\mathsf{heap}, \mathsf{o}, \mathsf{f}, \mathsf{v})\}\langle \mathsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{o.f} = \mathsf{v};\ \mathsf{rest}\rangle\phi, \Delta}$$

4.2 Predicates for Structural Heap Properties

For the sake of readability, in Sec. 3, we gave simplified versions of the predicates `\depth`, `\acyclic`, `\reach`, `\reachPlus` and `\disjoint` as compared to the actual implementation. In reality, these predicates have an extra argument that restricts their domain to a given set of fields. For example, instead of $\texttt{\textbackslash depth}(x)$ we might actually have $\texttt{\textbackslash depth}(\{x.\texttt{next}\}, x)$ which refers to the depth of `x` considering only those paths that go through the field `next`. A syntactic analysis infers automatically a safe approximation of these sets of fields by taking the fields explicitly used in the corresponding code fragment.

Ultimately, the various structural heap properties are reduced to reachability between objects which, therefore, must be expressible in the underlying program logic. The counterpart of JML's `\reach` predicate in JavaCard DL is

$$\texttt{\textbackslash reach} : \mathit{Heap} \times \mathit{LocSet} \times \mathit{Object} \times \mathit{Object} \times \mathit{int}$$

and expresses *bounded reachability* (or n-reachability): an object e is n-reachable from an object s with respect to a heap h and a set of locations l (of type $LocSet$) if and only if there exists a sequence $s = o_1 o_2 \cdots o_n = e$ where $o_{i+1} = o_i.f_i$ and $(o_i, f_i) \in l$ for all $0 < i < n$. The predicate $\texttt{\textbackslash reach}(h, l, s, e, n)$ is formally defined as $n \geq 0 \wedge s \neq \mathsf{null} \wedge ((n \doteq 0 \wedge s \doteq e) \vee \exists f.(o, f) \in l \wedge \texttt{\textbackslash reach}(h, l, s.f, e, n-1))$. As a consequence, from null nothing is reachable and also null cannot be reached.

Location sets in JavaCard DL are formalized in the data type $LocSet$ which provides constructors and the usual set operations (see [18] for a full account). Here we need only three location set constructors: the constructor *empty* for the empty set, the constructor $singleton(o, f)$ which takes an object o and a field f and constructs a location set with location (o, f) as its only member, and the constructor $allObjects(f)$ which stands for the location set $\{(o, f) \mid o \in Object\}$.

Example 7. $\texttt{\textbackslash reach}(h, allObjects(\mathsf{next}), head, last, 5)$ is evaluated to true iff the object $last$ is reachable from object $head$ in five steps by a chain of next fields.

Based on `\reach` we could directly axiomatize structural heap predicates such as $\texttt{\textbackslash acyclic}(h, l, o)$ or $\texttt{\textbackslash disjoint}(h, l, o, u)$. Instead we prefer to reduce structural heap predicates to $\texttt{\textbackslash reachPlus}(h, l, o, u)$ which is the counterpart of the JML

function of the same name in Sec. 3.2 and expresses reachability in at least one step. This has several advantages over using \reach: (1) the definition of predicates such as \acyclic does not use the step parameter of the \reach predicate and one would use existential quantification to eliminate it which impedes automation; and (2) for \reachPlus$(h, l, o, u)$ to hold one has to perform at least one step using a location in l. This renders the definition of properties such as \acyclic less cumbersome as the zero step case has been excluded.

The predicate \reachPlus can defined with the help of \reach and this definition can be used if necessary, however, in the first place we use a separate axiomatization of \reachPlus. This helps to avoid (or at least to delay as long as possible) the reintroduction of the step parameter and, hence, an additional level of quantification. For space reasons, we do not give the calculus rules for the axioms and auxiliary lemmas of the structural heap predicates like \acyclic and \disjoint (which are not too surprising). Instead, we describe in the following section one central difficulty that arises when reasoning about structural heap properties and how we solved it to achieve higher automation.

4.3 Field Update Independence

When reasoning about structural heap predicates one often ends up in a situation where one has to prove that a heap property is still valid after updating a location on the heap, i.e, after executing one or several field assignments. For instance, we might know that \acyclic$(h, l, u)$ holds and have to prove that after executing the assignment o.f=v; the formula \acyclic$(store(h, o, f, v), l, u)$ holds.

A precise analysis of the effect of a field update is expensive and makes automation significantly harder. As it is common in this kind of situation, it helps to *optimize the common case*. In the present context, this means to decide in most cases efficiently that a field assignment does not effect a heap property at all. This is sufficiently achieved by two simple checks:

1. The expression $singleton(o, f) \subseteq l$ checks whether an updated location $o.f$ is in the location set l of the heap property to be preserved. This turns out to be inexpensive for most (if not all) practically occurring cases. Whenever this check fails, the resulting *store* can be removed from the argument of the heap property. For instance, an assignment o.data=5 to the data field of a list does not change the list structure which depends solely on the next field. In that case we can rewrite \acyclic$(store(h, \mathsf{o}, \mathsf{data}, 5), l, u)$ to \acyclic$(h, l, u)$.
2. To check whether an object o whose field has been updated is reachable from one of the other mentioned objects, is more expensive than the previous one, but still cheaper than a full analysis. For example, we can check whether the object o is reachable from object u in case of \acyclic$(store(h, o, f, v), l, u)$. If the answer is negative we can again discard the store expression.

4.4 Path-Length Axiomatization

In general, the JML assertions generated by COSTA refer to the path-length of a data structure o as \depth$(l, o)$ where l is the location set restricting the depth

Table 1. Statistics for the Generation and Checking of Resource Guarantees

Bench	Certificate Generation				Cert. Size		Generation/Checking		
	$\mathbf{T}_{heap}$	$\mathbf{T}_{ana}$	$\mathbf{T}_{jml}$	$\mathbf{T}_{ver}$	**Nod**	**Br**	$\mathbf{T}_{gen}$	$\mathbf{T}_{check}$	%
`traverse`	14	36	2	2300	1208	52	2338	1100	47.05
`create`	54	150	8	3100	1499	47	3258	1400	42.97
`insert`	282	374	16	40800	19252	636	41190	5800	14.08
`indexOf`	26	86	4	5900	2439	67	5990	1800	30.05
`reverse`	72	130	8	20900	14206	673	21038	3400	16.16
`array2List`	62	154	8	2600	1457	37	2762	1400	50.69
`copy`	76	132	10	22600	14147	673	22742	3100	13.63
`searchtree`	142	202	6	3700	2389	97	3908	1500	38.38

to certain locations. This JML function is mapped to the JavaCard DL function $\backslash\texttt{depth}(h, l, o)$ which is evaluated to the maximal path-length of o in heap h using only locations from l. Its axiomatization is based on the n-reachability predicate $\backslash\texttt{reach}$ expressing that there exists an object u reachable in $\backslash\texttt{depth}(h, l, o)$ steps and that there is no object z reachable from o in more than $\backslash\texttt{depth}(h, l, o)$ steps. This definition is not used by default by the theorem prover, instead, automated proof search relies mainly on a number of lemmas that state more useful higher-level properties. For instance, given a term like $\backslash\texttt{depth}(store(h, o, f, v), l, u)$ there is a lemma which checks that o is reachable from u and some acyclicity requirements. If that is positive then the lemma allows us to use the same approximation for $\backslash\texttt{depth}$ in case of a heap update as detailed in Sec. 3.1.

5 Experimental Results

The implementation of our approach required the following non-trivial extensions to COSTA and KeY: (1) generate and output in COSTA the JML annotations `\depth`, `\acyclic` and `\disjoint` so that KeY can parse them; (2) synthesize suitable proof obligations in JavaCard DL that ensure correctness of the resource analysis; (3) axiomatize the JML `\depth`, `\acyclic` and `\disjoint` functions in KeY as described in Sec. 4 and implement heuristics for automation; and (4) implement heuristic checks in KeY that allow fast verification of the common case as described in Sec. 4.4. The resulting extended versions of KeY and COSTA are available for download from `http://fase2012.hats-project.eu`.

Table 1 shows first experiments using a set of representative programs that perform common list operations as well as searching for an element in a binary tree. The experiments were performed using an Intel Core2 Duo at 2.53GHz with 4Gb of RAM running Linux 2.6.32. Columns $\mathbf{T}_{heap}$, $\mathbf{T}_{ana}$ and $\mathbf{T}_{jml}$ show, respectively, the times (in milliseconds) taken by COSTA to perform the heap analysis (cyclicity, sharing and path-length), to execute the whole analysis (heap and other analyses performed by COSTA), and to generate the JML annotations. Column $\mathbf{T}_{ver}$ shows the time taken by KeY to verify the JML annotations generated by COSTA. The size of the generated proofs is indicated by their number of nodes **Nod** and branches **Br**. Column $\mathbf{T}_{gen}$ shows the total time taken to generate the

proof ($\mathbf{T}_{ana}$+$\mathbf{T}_{jml}$+$\mathbf{T}_{ver}$) and $\mathbf{T}_{check}$ shows the time taken by KeY to check the validity of the proof. The last column (%) shows the ratio $\mathbf{T}_{check}/\mathbf{T}_{gen}$.

Our preliminary experiments show already that a proof-carrying code approach to resource guarantees can be realized using COSTA and KeY with both certificate generation and checking being fully automatic. In our framework the code originating from an untrusted *producer* should be bundled with the proof generated by COSTA + KeY for a given resource consumption. Then the code *consumer* can check locally and automatically with KeY whether the claimed resource guarantees are verified. As expected, checking an existing proof with KeY takes on average only around 30% of the time to produce it.

6 Conclusions and Related Work

This paper describes the combination of a state-of-the-art resource analyzer (COSTA) and a formal verification tool (KeY) to automatically infer *and verify* resource guarantees that depend on the size of heap-allocated data structures in Java programs. The distribution of work among the two systems is as follows: COSTA generates ranking functions, invariants, as well as size relations, and outputs them as extended JML annotations of the analyzed program; KeY then verifies the resulting proof obligations in its program logic and produces proof certificates that can be saved and reloaded.

Many software verification tools including KeY [5], Why [8], VeriFast [16], or Dafny [12] rely on automatic theorem proving technology. While most of these systems are expressive enough to model and prove heap properties of programs, such proofs are far from being automatic. The main reason is that functional verification of heap properties requires complex invariants that cannot be found automatically. In addition, automated reasoning over heap-allocated symbolic data is far less developed than reasoning over integers or arrays.

With this paper we show that the automation built into a state-of-the-art verification system is sufficient to reason successfully about resource-related heap properties. The main reasons for this are: (a) the required invariants are inferred automatically in the resource analysis stage; (b) a limited and carefully axiomatized signature for heap properties expressed in logic is used. This confirms the findings of the SLAM project [4] that existing verification technology can be highly automatic for realistic programs and a restricted class of properties.

There exist several other cost analyzers which automatically infer resource guarantees for different programming languages [10,11]. However, none of them formally proves the correctness of the upper bounds they infer. An exception is [6], which verifies and certifies resource consumption (for a small programming language and not for heap properties). For the particular case of memory resources, [7] formally certifies the correctness of the static analyzer. We have taken the alternative approach of certifying the correctness of the upper bounds that the tool generates. This is not only much simpler, but has the additional advantage that the generated proofs can act as resource certificates.

Acknowledgments. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, the Spanish Ministry of Science and Innovation (MICINN) under th TIN-2008-05624 *DOVES* project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning 46(2), 161–203 (2011)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
3. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: Verified Resource Guarantees using COSTA and KeY. In: Proc. of PEPM 2011, pp. 73–76. ACM Press (2011)
4. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The Static Driver Verifier Research Platform. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 119–122. Springer, Heidelberg (2010)
5. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
6. Crary, K., Weirich, S.: Resource Bound Certification. In: POPL 2005, pp. 184–198. ACM Press (2000)
7. de Dios, J., Peña, R.: Certification of Safe Polynomial Memory Bounds. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 184–199. Springer, Heidelberg (2011)
8. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
9. Genaim, S., Zanardini, D.: The Acyclicity Inference of COSTA. In: Workshop on Termination (WST 2010) (July 2010)
10. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: precise and efficient static estimation of program computational complexity. In: Proc. of POPL 2009, pp. 127–139. ACM (2009)
11. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 287–306. Springer, Heidelberg (2010)
12. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
13. Necula, G.: Proof-Carrying Code. In: POPL 1997, ACM Press (1997)
14. Pnueli, A., Siegel, M.D., Singerman, E.: Translation Validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
15. Secci, S., Spoto, F.: Pair-Sharing Analysis of Object-Oriented Programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 320–335. Springer, Heidelberg (2005)

16. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An Automatic Verifier for Java-Like Programs Based on Dynamic Frames. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 261–275. Springer, Heidelberg (2008)
17. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for java bytecode based on path-length. ACM Trans. Program. Lang. Syst. 32(3) (2010)
18. Weiß, B.: Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction. PhD thesis, KIT (2011)

An Operational Decision Support Framework for Monitoring Business Constraints

Fabrizio Maria Maggi[1,⋆], Marco Montali[2,⋆⋆], and Wil M.P. van der Aalst[1]

[1] Eindhoven University of Technology, The Netherlands
{f.m.maggi,w.m.p.v.d.aalst}@tue.nl
[2] KRDB Research Centre, Free University of Bozen-Bolzano, Italy
montali@inf.unibz.it

Abstract. Only recently, *process mining* techniques emerged that can be used for *Operational decision Support* (OS), i.e., knowledge extracted from event logs is used to handle running process instances better. In the process mining tool ProM, a generic OS service has been developed that allows ProM to dynamically interact with an external information system, receiving streams of events and returning meaningful insights on the running process instances. In this paper, we present the implementation of a novel business constraints monitoring framework on top of the ProM OS service. We discuss the foundations of the monitoring framework considering two logic-based approaches, tailored to Linear Temporal Logic on finite traces and the Event Calculus.

Keywords: Declare, process mining, monitoring, operational decision support.

1 Introduction

Process mining has been traditionally applied on historical data that refers to past, complete process instances. Recently, the exploitation of process mining techniques has been extended to deal also with running process instances which have not yet been completed. In this setting, process mining provides *Operational decision Support* (OS), giving meaningful insights that do not only refer to the past, but also to the present and the future [1]. In particular, OS techniques can be used to: *check* the current state of affairs detecting deviations between the actual and the expected behavior; *recommend* what to do next; *predict* what will happen in the future evolution of the instance.

In order to enable the effective development of OS facilities, the widely known process mining framework ProM 6 [2] incorporates a backbone for OS [3]. Here,

⋆ This research has been carried out as a part of the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). The project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.
⋆⋆ This research has been partially supported by the NWO "Visitor Travel Grant" initiative, and by the EU Project FP7-ICT ACSI (257593).

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 146–162, 2012.

all the common functionalities needed for OS are implemented, such as management of requests coming from external information systems, dynamic acquisition and correlation of incoming partial execution traces (representing the evolution of process instances), and interaction with different process instances at the same time. The OS backbone relies on a client-server architecture. The client is exploited by an external stakeholder to send a partial trace to ProM and ask queries related to OS. On the server side, an OS service (running inside ProM) takes care of coordinating the available OS functionalities in order to answer such queries. Multiple *OS providers* that encapsulate specific OS functionalities can be developed and dynamically registered to the OS service.

In this work, we present the implementation of a novel runtime compliance verification framework on top of the ProM OS. The framework is called *Mobucon* (*Mo*nitoring *bu*siness *con*straints) and its focus is to dynamically *check* the compliance of running process instances with business constraints, detecting deviations and measuring the *degree of adherence* between the actual and the expected behavior.

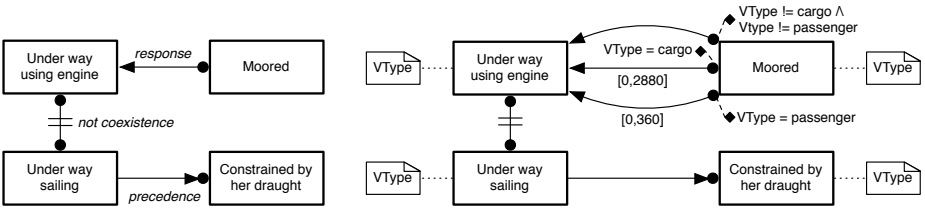
Given a business constraints reference model and a partial trace characterizing the running execution of a process instance, Mobucon infers the status of each business constraint. In particular, it produces a constantly updated snapshot about the state of each business constraint, reporting whether it is currently violated. Consequently, it determines whether the process instance is currently complying with the reference model or not. Beside this, other meaningful insights can be provided to end users, such as, for example, indicators and metrics related to the "degree of compliance", e.g., relating the number of violated constraints with their total number.

The paper is organized as follows. Section 2 presents the *Declare* language [4] and its extension to include metric temporal constraints and constraints on event-related data. The language is declarative and graphical. Moreover, *Declare* has been formalized using a variety of logic-based frameworks, such as Linear Temporal Logic (LTL) with a finite-trace semantics[1] [5,6] and the Event Calculus (EC) [7,8]. Section 3 describes the architecture of our proposed framework. In Sect. 4 and 5, we describe the implementation of two different reasoning engines as OS providers based on LTL and on the EC respectively. We are currently applying our framework to various real-world case studies; in Sect. 6, we report on the monitoring of *Declare* constraints in the context of maritime safety and security. Finally, Sect. 7 includes a comparison of the two approaches and discusses related work and conclusion.

2 Declare

Declare is a declarative, constraint-based process modeling language first proposed in [5,4]. In a constraint-based approach, instead of explicitly specifying all the acceptable sequences of activities in a process, the allowed behavior of

[1] For compactness, in the following we will use the LTL acronym to denote LTL on finite traces.



(a) Basic *Declare* model

(b) *Declare* model augmented with data, data-aware conditions and metric temporal constraints

Fig. 1. Two *Declare* models in the context of maritime safety and security

the process is implicitly specified by means of declarative constraints, i.e., rules that must be respected during the execution. In comparison with procedural approaches, that produce "closed" models, i.e., models where what is not explicitly supported is forbidden, declarative languages are "open" and tend to offer more possibilities for execution. In particular, the modeler is not bound anymore to explicitly enumerate the acceptable executions and models remain compact: they specify the mandatory and undesired behaviors, leaving unconstrained all the courses of interaction that are neither mandatory nor forbidden.

Declare is characterized by a user-friendly graphical front-end and is based on a formal back-end. More specifically, the formal semantics of *Declare* can be specified by using LTL [5,6], abductive logic programming with expectations [6], or the EC [7,8]. These characteristics are crucial for two reasons. First, *Declare* can be used in real scenarios being understandable for end-users and usable by stakeholders with different backgrounds. Second, *Declare*'s formal semantics enable verification and automated reasoning. This is a key aspect in the implementation of monitoring tools for *Declare* models.

Figure 1a shows a simple *Declare* model elicited in the context of a real case study related to the monitoring of vessels behavior in the context of maritime safety and security. We use this example to explain the main concepts. It involves four *events* (depicted as rectangles, e.g., *Under way using engine*) and three *constraints* (shown as arcs between the events, e.g., *not coexistence*). Events characterize changes in the navigational status of each monitored vessel. Constraints highlight mandatory and forbidden behaviors, implicitly identifying the acceptable execution traces that comply with (all of) them. In our case study, a vessel can be either *Under way using engine* or *Under way sailing* but not both, as indicated by the *not coexistence* between such two events. A vessel can be *Constrained by her draught*, but only after being *Under way sailing* (a vessel equipped with an engine cannot be constrained by draught and a sailing vessel cannot be constrained before it is under way). This is indicated by the *precedence* constraint. Finally, after being *Moored* each vessel must eventually be *Under way using engine*, as specified by the *response* constraint.

In [7], an extension of this constraint-based language has been proposed; this extension incorporates also non-atomic activities (i.e., activities whose execution

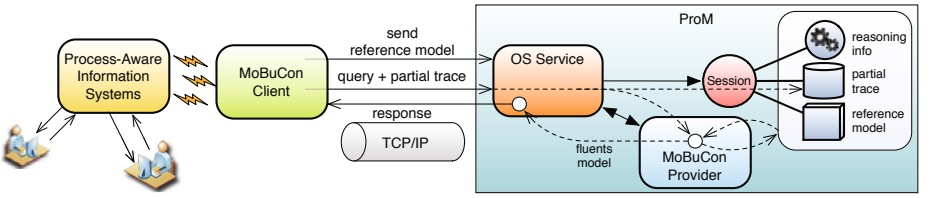


Fig. 2. Mobucon Architecture

is characterized by a life cycle that includes multiple events), event-related data and data-aware conditions and metric temporal constraints (for specifying delays, deadlines and latencies). This extended language is exploited in Fig. 1b to augment the aforementioned constraints with conditions on time and data. More specifically, we assume that each event is equipped with two data: the identifier of the vessel and its type. In particular, the *response* constraint is now differentiated on the basis of the vessel type, introducing different timing requirements (which are specified with the granularity of *minute*). The first *response* constraint indicates that if the type of the vessel is *Passenger ship* and event *Moored* occurs, then *Under way using engine* must eventually occur within 6 hours at most. The second one indicates that if the type of the vessel is *Cargo ship* and *Moored* occurs, then *Under way using engine* must eventually occur within 48 hours. A last standard *response* constraint is employed to capture the behavior of all other vessels, without imposing any deadline. Finally, although not explicitly shown in the diagram, each constraint is applied to events that are associated to the same vessel identifier. This correlation mechanism makes it possible to properly monitor also a unique event streams collecting the evolving behaviors of multiple vessels at the same time.

3 Mobucon Architecture

Figure 2 shows the overall architecture of Mobucon. Mobucon relies on the general architecture of the OS backbone implemented inside ProM 6. Such backbone has been introduced and formalized using colored Petri nets in [3]; in Sect. 3.1, we will therefore sketch some relevant aspects of the general architecture. In Sect. 3.2, we ground the discussion to the specific case of Mobucon, discussing the skeleton of our compliance verification OS provider. The data exchanged between the Mobucon client and provider is illustrated in Sect. 3.3. Finally, in Sect. 3.4, we describe the implemented Mobucon clients. The two concrete instantiations of the Mobucon skeleton in the LTL and EC settings are discussed in Sect. 4 and 5.

3.1 General Architecture

The ProM OS architecture relies on the well-known client-server paradigm. More specifically, the ProM OS service manages the interaction with running process

instances and acts as a mediator between them and the registered specific OS providers.

Sessions are created and handled by the OS Service to maintain the state of the interaction with each running client. To establish a stateful connection with the OS Service, the client creates a session handle for each managed running process instance, by providing host and port of the OS Service. When the client sends a first query related to one of such running instances to the OS service, it specifies information related to the initialization of the connection (such as reference models, configuration parameters, etc.) and to the type of the queries that will be asked during the execution. This latter information will be used by the OS Service to select, among the registered active providers, the ones that can answer the received query. The session handle takes care of the interaction with the service from the client point of view, hiding the connection details and managing the information passing in a lazy way. The interaction between the handle and the service takes place over a TCP/IP connection.

3.2 Mobucon Skeleton

In Mobucon, the interaction between a client and the OS service mainly consists of two aspects. First of all, before starting the runtime compliance verification task, the client sends to the OS service the *Declare* reference model to be used. This model is then placed inside the session by the OS service. The reference model is an XML file that contains the information about events and constraints mentioned in the model. This format is generated by the *Declare* editor (`www.win.tue.nl/declare/`). The client can also set further information and properties. For example, each constraint in the *Declare* reference model can be associated to a specific weight, that can be then exploited to compute metrics and indicators that measure the degree of adherence of the running instance to the reference model.

Secondly, during the execution, the client sends queries about the current monitoring status for one of the managed process instances. The session handle augments these queries with the partial execution trace containing the evolution that has taken place for the process instance after the last request. The OS Service handles a query by first storing the events received from the client, and then invoking the Mobucon provider.

The Mobucon provider recognizes whether it is being invoked for the first time w.r.t. that process instance. If this is the case, it takes care of translating the reference model onto the underlying formal representation. The provider then returns a fresh result to the client, exploiting a reasoning component for the actual result's computation. The reasoning component, as well as the translation algorithm, are dependent on the chosen logical framework (LTL or EC), while the structure of the skeleton is the same for the two approaches. After each query, the generated result is sent back to the OS service, which possibly combines it with the results produced by other relevant providers, finally sending the global response back to the client.

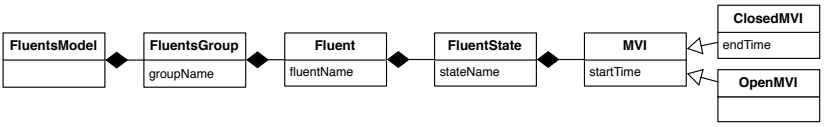


Fig. 3. Fluent model used to store the evolution of constraints

3.3 Exchanged Data and Business Constraints States

We now discuss the data exchanged by the Mobucon client and provider. Note that these data are common to both instantiations of the provider (Mobucon LTL and Mobucon EC). The partial execution traces sent by the client to the OS use the XES format (`www.xes-standard.org/`) for event data. XES is an extensible XML-based standard recently adopted by the IEEE task force on process mining.

The response produced by the Mobucon provider is composed of two parts. The first part contains the temporal information related to the evolution of each monitored business constraint from the beginning of the trace up to now. At each time point, a constraint can be in one state, which models whether it is currently: *satisfied*, i.e., the current execution trace complies with the constraint[2]; *(permanently) violated*, i.e., the process instance is not compliant with the constraint; *pending* (or *possibly violated*), i.e., the current execution trace is not compliant with the constraint, but it is possible to satisfy it by generating some sequence of events. This state-based evolution is encapsulated in a *fluent model* which obeys to the schema sketched in Fig. 3. A fluent model aggregates fluents groups, containing sets of correlated fluents. Each fluent models a multi-state property that changes over time. In our setting, fluent names refer to the constraints of the reference model. The fact that the constraint was in a certain state along a (maximal) time interval is modeled by associating a closed MVI (Maximal Validity Interval) to that state. MVIs are characterized by their starting and ending timestamps. Current states are associate to open MVIs, which have an initial fixed timestamp but an end that will be bounded to a currently unknown future value.

The Mobucon provider also computes the current value of a *compliance indicator* of the running monitored instance. This number gives an immediate feeling about the "degree of adherence" between the instance and the reference model. A low degree of adherence can be interpreted differently depending on the application domain. In general, it is used to classify a process instance as "unhealthy". However, it can also be used to show that a reference model is obsolete and it must be improved to better reflect the reality. The compliance indicator can be computed using different metrics, that can consider the current state of constraints, as well as other information such as the weight of each individual constraint. For example, the compliance indicator shown in Fig. 5a, implemented in Mobucon LTL, is evaluated, at some time t, through the formula $1 - \frac{\sum_i weight_i \# viol_i(t)}{\#events(t) \sum_i weight_i}$, and takes into account the number of violations of each

[2] Mobucon LTL also differentiates between possibly and permanently satisfied.

individual constraint of the reference model ($\#viol_i$) and its weight ($weight_i$). On the other hand, the compliance indicator shown in Fig. 5b, implemented in Mobucon EC, considers the number of violated ($\#viol$) and satisfied ($\#sat$) instances. In particular, at some time t the compliance indicator corresponds to $1 - \frac{\#viol(t)}{\#viol(t)+\#sat(t)}$[3].

3.4 Mobucon Clients

We have developed three Mobucon clients, in order to deal with different settings: (a) manual insertion of the events, (b) replay of a process instance starting from a complete event log, and (c) acquisition of events from an information system. The first two clients are mainly used for testing and experimentation. The last client requires a connection to some information system, e.g., a workflow management system. The three clients differ on how the user is going to provide the stream of events, but all of them include an interface with a graphical representation of the obtained fluent model, showing the evolution of constraints and also reporting the trend of the compliance indicator (Fig. 4).

4 Mobucon LTL

As discussed earlier, there are two Mobucon providers for monitoring business constraints: one based on LTL and one based on the EC. We now describe the LTL-based provider [9]. The basic idea is that a stream of events is monitored w.r.t. a given *Declare* reference model. Each LTL constraint implied by the *Declare* model is translated to a finite state automaton. Moreover, the conjunction of all LTL constraints is also translated to a finite state automaton. The generated automata are used to monitor the behavior. Using the terminology introduced in [9], we call the automaton corresponding to a single *Declare* constraint *local automaton* and the automaton corresponding to their conjunction *global automaton*. Local automata are used to monitor each single constraint in isolation, whereas the global automaton is used to monitor the entire system and detect non-local violations originated by the interplay of multiple constraints.

4.1 Modeling and Implementation

When Mobucon LTL receives a request from a new process instance, it first initializes the session for that instance. In particular, each single constraint of the *Declare* model associated to the session by the client and their conjunction are translated into finite state automata. For the translation, we use the algorithm introduced for the first time in [10] and optimized in [11]. Local and global automata are stored in the session. After that, the provider processes the event (or a collection of events) received with the first request from the client. The following requests will provide again single events or collections of events. The

[3] If $\#viol(t) + \#sat(t) = 0$, then the compliance indicator is defined to be 1.

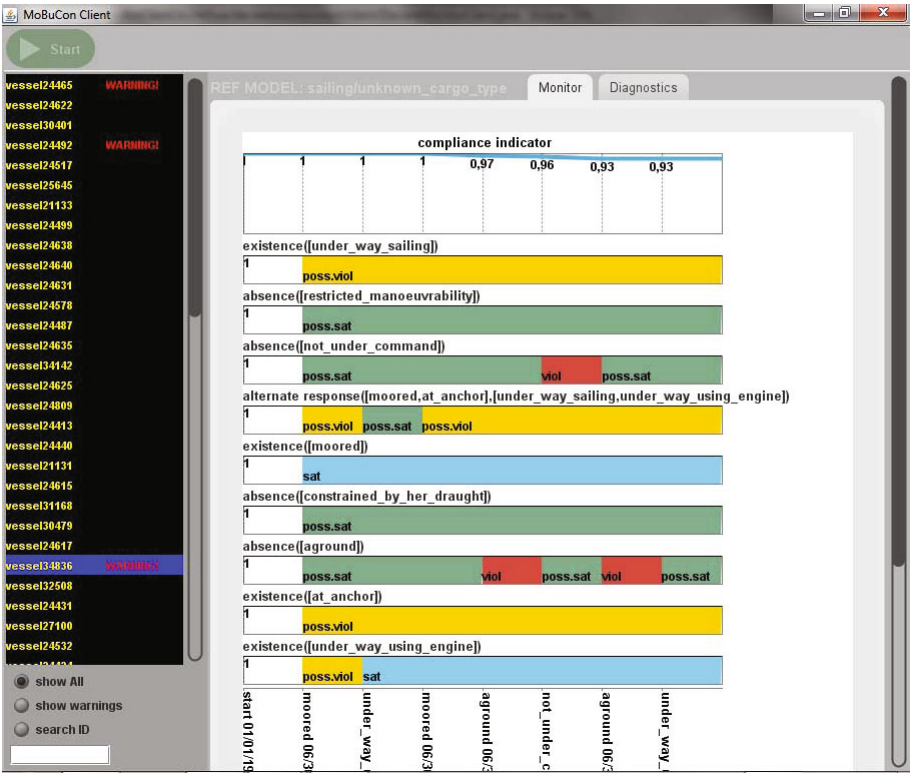


Fig. 4. Screenshot of one of the Mobucon clients

events are processed one by one by using the automata every time retrieved from the session. In this way, the state of each automaton is always preserved from the last request. The set of fluents' MVIs associated to each constraint is recomputed accordingly and returned by the reasoner.

4.2 Approach

Both global and local automata are reduced so that, if a transition violates the automaton from a certain state, this transition does not appear in the list of the outgoing transitions from that state. More specifically, a transition can be positive if it is associated to a single positive label (representing an event, e.g., *moored*), or negative if it is associated to negative labels (e.g., $\neg$*aground*). Positive labels indicate that we follow the transition when exactly the event corresponding to that label occurs, whereas negative labels indicate that we can follow the transition for any event not mentioned. Hence, acceptable events correspond to the label associated to a positive outgoing transition from the current state or to no one of the labels associated to a negative outgoing transition.

The Mobucon LTL provider checks first whether the processed event is acceptable by the global automaton. If the event is allowed, the provider fires

the corresponding transition on the global automaton. In this case, to compute the state of every single constraint in isolation as well, it also fires the transitions corresponding to the processed event on the local automata (note that, if the event is acceptable by the global automaton, it is also acceptable by all local automata). If, after having fired the transition, a local automaton is in an non-accepting state, the corresponding constraint is possibly violated. If a local automaton is in an accepting state, the corresponding constraint is (possibly or permanently) satisfied. To distinguish between possibly and permanently satisfied constraints, the provider checks whether all possible events correspond to a self loop on the current state. If this is the case, the constraint is permanently satisfied, otherwise it is possibly satisfied. If the processed event violates the global automaton, from the point of view of the automata, the violating event is ignored. However, the provider still informs the client that the event caused a violation w.r.t. the reference model. Moreover, it also gives intuitive diagnostics about the violation. Indeed, the global automaton allows the provider to precisely identify which events were permitted instead of the one that caused the violation. This information is derived from the labels of the outgoing negative and positive transitions from the current state in the global automaton.

In some cases, a violation in the global automaton can be directly reduced to a violation of a local automaton. However, in other cases none of the individual local automata is violated as the problem stems from the interplay of multiple constraints [9]. In the latter case, the Mobucon LTL provider is able to identify the conflicting sets of constraints, i.e., the minimal sets of constraints that cause the violation.

5 Mobucon EC

Mobucon EC exploits a reactive EC-based reasoner to provide monitoring facilities. When a first query is received for some process instance, the provider applies a translation algorithm which analyzes the reference model stored in the corresponding session, producing a set of corresponding EC axioms. It then creates a new instance of the reasoner, initializing it with the EC theory obtained from the translation procedure. The reasoner instance is then stored into the session. Every time a new partial trace must be checked, the reasoner is extracted from the session and updated with the new events. This triggers a new reasoning phase in which the previously stored fluents' MVIs are revised and extended. The set of all MVIs is then returned by the reasoner.

In the following, we first sketch how *Declare* constraints, possibly augmented with data and metric temporal aspects, can be tackled by means of EC axioms. We then discuss the implementation of the reasoner.

5.1 Modeling

A comprehensive description of how the EC can be used in the *Declare* setting can be found in [8]. Here, we consider one of the constraints mentioned in Fig. 1b,

namely the *response* constraint over a *Cargo ship*, to give an intuition about such a translation, considering data and metric temporal constraints as well.

Broadly speaking, an EC theory is a logic program which employs special predicates for modeling how fluents change over time, in response of the execution of certain events. For example, $initiates(e, f, t)$ ($terminates(e, f, t)$) is used to say that event e initiates (terminates) f, i.e., makes f true (false), at time t; $holds\_at(f, t)$ is used to run queries over the validity of fluents, in this case verifying whether f is true at time t. For a comprehensive description of the EC, we refer the reader to [12].

In the context of *Declare*, and differently from the LTL-based approach, the runtime characterization of business constraints is not given over the constraints themselves, but is tailored to constraints' instances. A constraint instance represents a specific "grounding" of the constraint inside a specific context, i.e., with specific data, specific instantiation time, and so on. According to this observation, in the EC-based formalization of *Declare* fluents have the form $state(i(ID, Params), State)$, where ID is the identifier of the constraint, $Params$ is a list of parameters characterizing a specific instance of the constraints, and $State$ is the current state of the instance, i.e., one among *sat*, *viol* and *pend* (to respectively model that the constraint instance is satisfied, violated or pending). In our example, the *response* constraint over a *Cargo ship* will be identified by *cr*, and the params characterizing each instance will be the identifier of the vessel (needed to properly correlate events) and the creation time (needed to properly check the metric temporal constraints).

EC axioms are given over event types, which are then subject, during the execution, to unification with each occurring concrete event. Event types have the form $exec(Name, Who, Data)$, where $Name$ is the name of the event, Who identifies the entity that originated the event, and $Data$ is a list of further data. The *response* over a *Cargo ship* is associated to the *moored* and (*Under way using*) *engine* events, which can be represented by the two event types $exec(moored, V_{id}, [V_{type}])$ and $exec(engine, V_{id}, [V_{type}])$. It is instantiated every time a *moored* event happens for a cargo vessel; the instance is put in a pending state, waiting for the occurrence of a corresponding *engine* event:

$$initiates(exec(moored, V_{id}, [cargo]), status(i(cr, [V_{id}, T]), pend), T)$$

A state transition from the pending to the satisfied state happens for an instance, if the following conditions hold: (1) the instance is currently pending; (2) an *engine* event occurs for a *Cargo ship*; (3) the event has the same vessel identifier of the instance; (4) the timestamp of the event is after the creation time of the instance, but before the actual deadline (which corresponds to the creation time plus 2880 minutes). Such state transition is modeled by terminating the previous state and initiating the new one, if all conditions are satisfied:

$$terminates(exec(engine, V_{id}, [cargo]), status(i(cr, [V_{id}, T_c]), pend), T) :- holds\_at(status(i(cr, [V_{id}, T_c]), pend), T), T > T_c, T \leq T_c + 2880.$$

$$initiates(exec(engine, V_{id}, [cargo]), status(i(cr, [V_{id}, T_c]), sat), T) :- holds\_at(status(i(cr, [V_{id}, T_c]), pend), T), T > T_c, T \leq T_c + 2880.$$

Contrariwise, if a (generic) event happens at a time which is greater than the creation time of the instance plus 2880, and the constraint instance is still pending, this attests that the deadline has expired, and that a transition from the pending to the violated state must be triggered:

$$terminates(\_, status(i(cr, [V_{id}, T_c]), pend), T) :- \\ holds\_at(status(i(cr, [V_{id}, T_c]), pend), T), T > T_c + 2880.$$
$$initiates(\_, status(i(cr, [V_{id}, T_c]), viol), T) :- \\ holds\_at(status(i(cr, [V_{id}, T_c]), pend), T), T > T_c + 2880.$$

Finally, a further general rule is added to state that each pending instance becomes violated when the process instance is completed.

The visualization depicted in Fig. 5b shows the status of the various constraints for a running trace and is based on the above axioms (together with the ones modeling the other constraints in Fig. 1b).

5.2 Reasoner Implementation

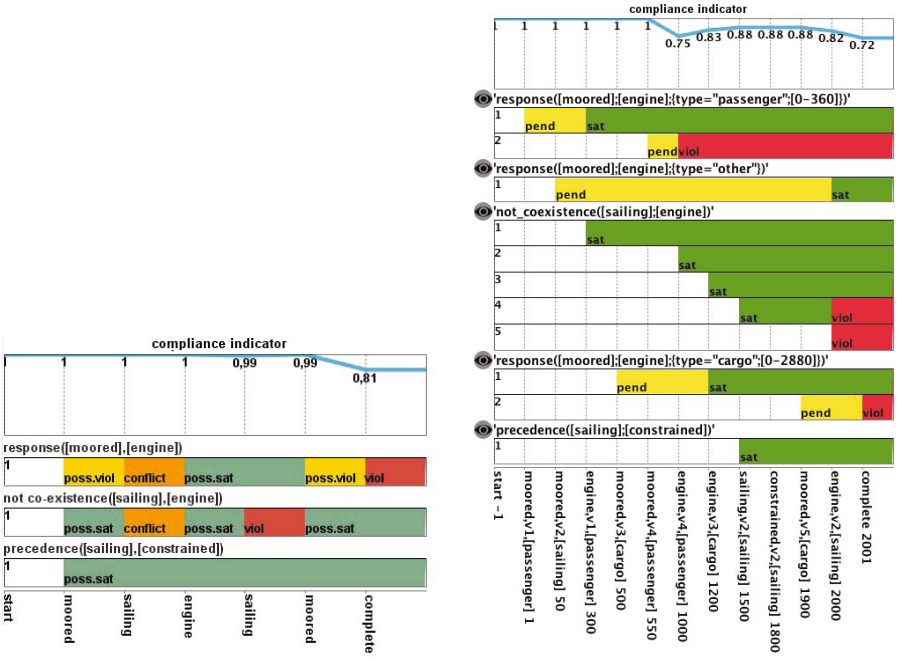
To effectively compute the MVIs characterizing the evolution of each constraint instance, Mobucon EC relies on a reactive EC reasoner and three translation components. A first translator converts the XML representation of a *Declare* reference model to a corresponding set of EC axioms. A second one converts a XES (partial) trace to a set of logic programming facts, also applying a translation of timestamps using the chosen granularity; such facts are then matched against the EC axioms that formalize the reference model. A last translator is used to convert the outcome produced by the reasoner (a set of strings) to a fluent model according to the schema of Fig. 3.

The reactive reasoner is inspired by the Cached EC (CEC) developed by Chittaro and Montanari [13]. It uses a Prolog-based axiomatization of the EC predicates following the CEC philosophy, i.e., already computed MVIs of fluents are cashed and subsequently revised and extended as new events are received.

Different underlying Prolog engines can be plugged into the tool. In particular, we experimented TuProlog (`tuprolog.alice.unibo.it/`) which is completely implemented in JAVA and thus guarantees a seamless integration inside ProM, and YAP (`yap.sourceforge.net/`), which is one of the highest-performance Prolog engine available today.

6 Case Study

In this section, we present the application of the two Mobucon providers (LTL and EC) as part of a case study conducted within the research project Poseidon (`www.esi.nl/poseidon/`) and focused on the analysis of vessel behavior in the domain of maritime safety and security. The case study has been provided by Thales, a global electronics company delivering mission-critical information systems and services for the Aerospace, Defense, and Security markets. In our experiments, we use logs collected by an on-board maritime Automatic Identification System



(a) Mobucon LTL. **(b)** Mobucon EC.

Fig. 5. Examples of monitoring results in our case study

(AIS) [14], which acts as a transponder that logs and sends events to an AIS receiver. An event represents a change in the navigational status of a vessel (e.g., *moored* or *Under way using engine*). Each event has an associated vessel ID and vessel type (e.g., *Passenger ship* or *Cargo ship*). The logs are excerpts of larger logs and correspond to a period of one week. The standard behavior of the vessels is described by domain expert using *Declare*, where constraints are used to check the compliance of the behavior of vessels as recorded in the logs.

Let us first focus on the Mobucon LTL provider. Figure 1a shows the reference model used to monitor vessels behavior. Each vessel corresponds to a process instance in the log. Figure 5a shows a graphical representation of the constraints' evolution for a specific instance. Events are displayed on the horizontal axis (for the sake of readability, a more compact notation is used). The vertical axis shows the constraints, reporting their evolution as events occur.

When event *moored* is executed the *response* constraint becomes possibly violated. Indeed, the constraint is waiting for the occurrence of another event (execution of (*Under way using*) *engine*) to become satisfied. After *moored*, (*Under way*) *sailing* is executed, leading to a conflict caused by the interplay of the *not coexistence* and the *response* constraints. The conflict is due to the fact that the first constraint forbids whereas the other constraint requires the presence of event *engine*. Note that, after a conflict or a (local) violation, constraints can

become non-violated. In fact, Mobucon LTL implements a recovery strategy where the violating events are ignored (after having been reported). In this case, for instance, when *sailing* occurs, the conflict is raised but the event is, in fact, ignored. The next event is *engine* and *response* (that was possibly violated before the conflict) becomes possibly satisfied. After that, when event *sailing* occurs, *not coexistence* becomes permanently violated because *engine* and *sailing* cannot coexist in the same trace (note that also in this case the violating event is ignored after that the violation has been reported). The next event is *moored* and *response* becomes possibly violated. When the case completes, the *response* constraint becomes violated because it is not possible to satisfy it anymore.

Finally, note the trend of the compliance indicator in Fig. 5a. The indicator decreases in correspondence of each (local) violation. This example also shows clearly that a violation of the *response* constraint influences the indicator more than a violation of the *not coexistence* constraint.

Let us now consider the Mobucon EC provider, which employs the reference model shown in Fig. 1b. In order to show the potentiality of the approach, we consider in this case the unique events stream generated by the AIS receiver; correlation between events referring the same vessel is under the responsibility of the framework itself, using the formalization discussed in Sec. 5. Figure 5b shows a graphical representation of the constraints' evolution. Events (with attached data and timestamps) are displayed on the horizontal axis. The vertical axis shows the constraints and their instances, reporting their evolution as time flows.

Every time event *moored* occurs, a new instance of the *response* constraint (for the specific vessel type) is created. At first, the state of the instance is pending because it is waiting for the occurrence of an (*Under way using*) *engine* event referring to the same vessel ID, and within the deadline specific for the corresponding vessel type. Event *engine* occurs for *Passenger ship v1* less than 6 hours after *moored*. For *v4* this takes more than 6 hours, thus resulting in a violation. Similar to the example used for the Mobucon LTL provider, also in this case, the occurrence of *sailing* for *Sailing boat v2* generates a conflict between the instance of the *response* constraint and the instance of the *not coexistence* constraint corresponding to this vessel. They can never become both satisfied, the first requiring and the other forbidding the presence of event *engine* for this vessel. However, unlike the LTL-based provider, the Mobucon EC provider does not point out any problem when the conflict arises. Only when, as the last event of the trace, *engine* occurs for *v2*, the instance of the *not coexistence* constraint for vessel *v2* becomes violated. This example shows that, on the one hand, the Mobucon EC provider is able to monitor constraints augmented with data conditions and metric temporal constraints. On the other hand, the Mobucon LTL provider supports the early detection of violations originating from a conflict among two or more constraints.

As explained in Sect. 3.3, the compliance indicator is computed differently in both providers. For both providers the indicator decreases after each violation. However, in EC-based provider, the compliance indicator increases when new satisfied instances are created.

Table 1. Comparison between the Mobucon LTL and EC providers (I = implemented, I* = partially implemented, + = supported by the formal framework, – = not supported by the formal framework)

	LTL	EC		LTL	EC
1. single constraints monitoring	I	I	5. recovery and compensation	+	+
2. non-local violations	I*	–	6. metric temporal aspects	–	I
3. continuous support	I	I	7. data and data-aware conditions	–	I*
4. diagnostics	I	–	8. non-atomic activities	–	+

7 Discussion and Conclusion

This paper presents a new Operational decision Support (OS) framework for monitoring business constraints. The framework implementation exploits the functionalities provided by the OS service in ProM. Mobucon comes with a general flexible architecture able to accommodate multiple reasoning engines. In this paper, we demonstrate two such engines, one based on (finite-trace) Linear Temporal Logic (LTL) and automata, and the other on the Event Calculus (EC) and a Prolog-based reactive reasoner.

In the literature, most of the proposed approaches for compliance verification either work on static models at design time [15,16] or on off-line a-posteriori conformance checking [17] using only historical data. The majority of approaches for *online* business process monitoring focus on measuring numerical attributes, such as Key Performance Indicators (KPIs). For example, in [18], a framework is introduced for modeling and monitoring of KPIs in Semantic Business Process Management. In particular, the authors integrate the KPI management into a semantic business process lifecycle, creating an ontology that is used by business analysts to define KPIs based on ontology concepts. In [19], an integrated framework is presented for run-time monitoring and analysis of the performance of WS-BPEL processes. In particular, this framework allows for dependency analysis and machine learning with the ultimate goal of discovering the main factors influencing process performance (KPI adherence).

An exception to this trend is the work by Ly et al. on semantic constraints in business processes [20]. This work is more related the one presented here. Both approaches recognize the importance of runtime compliance verification of processes with rules and constraints. However, while Ly et al. aims to describe a comprehensive framework for compliance of semantic constraints over the whole process lifecycle, here we have proposed concrete ways for attacking this problem during the execution of processes.

Table 1 provides a comparison of our two OS providers for monitoring business constraints (LTL-based and EC based). Analysis of this table provides some interesting insights. First of all, both approaches are able to manage the monitoring of individual business constraints. Non-local violations refer to the situation in which no single constraint is currently violated, but there is a conflicting set of constraints. Whereas the LTL-based approach can discover non-local violations thanks to the construction of the global automaton, the EC-based approach does

not support this. Note that the detection of non-local violations is currently only partially supported by the Mobucon LTL provider: the non-local violations is detected, but the minimal conflicting set is not yet computed efficiently. We are currently working on extending the colored-automata based approach to more efficiently identify minimal sets of conflicting constraints [21]. Both approaches support continuous support, i.e., the monitoring framework is able to provide support even after a violation takes place. While the Mobucon EC provider is only able to detect that a violation has taken place, Mobucon LTL also provides diagnostics about which events were expected (not) to occur. Although recovery and compensation mechanisms have not yet been included in our implementation, both approaches can support them [9,22].

The last three rows in Tab. 1 refer to the extension of the *Declare* language. Metric temporal aspects have been already incorporated into the Mobucon EC provider [8]. Metric temporal logics and timed-automata will be investigated to improve the LTL-based approach in this direction. Data and data-aware conditions are not-expressible in LTL, while the EC-based tool is being extended to accommodate them. Its ability to support data is attested by the formalization example shown in Sec. 5 and Fig. 5b. Similarly, EC is also able to support non-atomic activities.

Finally, let us briefly comment on the performance of the two approaches. For the Mobucon LTL provider, a recent investigation has revealed that very efficient algorithms can be devised for building local and global automata [11]. Once the automata are constructed, runtime monitoring can be supported in an efficient manner. The state of an instance can be monitored in constant time, independent of the number of constraints and their complexity. According to [11], the time to construct an automaton is 5-10 seconds for random models with 30-50 constraints. For models larger than this, automata can no longer routinely be constructed due to lack of memory, even on machines with 4-8 GiB RAM. For the Mobucon EC provider, some complexity results are inherited from the seminal investigation by Chittaro and Montanari [13]. An initial investigation of the performance of this approach (with YAP Prolog as underlying reasoner) can be found in [8]. Differently from the LTL-based approach, whose most resource-consuming task is the generation of the automaton, which is done before the execution, the EC-based approach triggers a reasoning phase every time a new event is acquired. Despite this, our investigation shows that, for randomly generated models and traces, the reasoner takes an average time of 300ms to process the 1000th acquired event with a model containing 100 constraints.

References

1. van der Aalst, W.M.P., Pesic, M., Song, M.: Beyond Process Mining: From the Past to Present and Future. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 38–52. Springer, Heidelberg (2010)
2. Verbeek, E., Buijs, J., van Dongen, B., van der Aalst, W.: Prom 6: The process mining toolkit. In: Demo at BPM 2010 (2010)

3. Westergaard, M., Maggi, F.M.: Modeling and Verification of a Protocol for Operational Support Using Coloured Petri Nets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 169–188. Springer, Heidelberg (2011)
4. Pesic, M., Schonenberg, H., van der Aalst, W.: DECLARE: Full Support for Loosely-Structured Processes. In: EDOC 2007, pp. 287–300 (2007)
5. Pesic, M., van der Aalst, W.M.P.: A Declarative Approach for Flexible Business Processes Management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
6. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative Specification and Verification of Service Choreographies. ACM Transactions on the Web 4(1) (2010)
7. Montali, M.: Specification and Verification of Declarative Open Interaction Models. LNBIP, vol. 56. Springer, Heidelberg (2010)
8. Montali, M., Maggi, F., Chesani, F., Mello, P., van der Aalst, W.: Monitoring Business Constraints with the Event Calculus. Technical Report DEIS-LIA-002-11, University of Bologna (Italy) (2011), LIA Series no. 97, http://www.lia.deis.unibo.it/Research/TechReport/LIA-002-11.pdf
9. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 132–147. Springer, Heidelberg (2011)
10. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: ASE 2001, pp. 412–416 (2001)
11. Westergaard, M.: Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 83–98. Springer, Heidelberg (2011)
12. Shanahan, M.: The Event Calculus Explained. In: Artificial Intelligence Today: Recent Trends and Developments, pp. 409–430 (1999)
13. Chittaro, L., Montanari, A.: Efficient Temporal Reasoning in the Cached Event Calculus. Computational Intelligence 12, 359–382 (1996)
14. International Telecommunications Union: Technical characteristics for a universal shipborne Automatic Identification System using time division multiple access in the VHF maritime mobile band. Recommendation ITU-R M.1371-1 (2001)
15. Governatori, G., Milosevic, Z., Sadiq, S.W.: Compliance Checking Between Business Processes and Business Contracts. In: EDOC 2006, pp. 221–232 (2006)
16. Awad, A., Decker, G., Weske, M.: Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 326–341. Springer, Heidelberg (2008)
17. van der Aalst, W.M.P., de Beer, H.T., van Dongen, B.F.: Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In: Meersman, R., Tari, Z. (eds.) CoopIS/DOA/ODBASE 2005. LNCS, vol. 3760, pp. 130–147. Springer, Heidelberg (2005)
18. Wetzstein, B., Ma, Z., Leymann, F.: Towards measuring key performance indicators of semantic business processes. In: BIS 2008, pp. 227–238 (2008)
19. Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Dustdar, S., Leymann, F.: Monitoring and analyzing influential factors of business process performance. In: EDOC 2009, pp. 141–150 (2009)

20. Ly, L.T., Göser, K., Rinderle-Ma, S., Dadam, P.: Compliance of Semantic Constraints - A Requirements Analysis for Process Management Systems. In: GRCIS 2008 (2008)
21. Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.P.: Runtime Verification of LTL-Based Declarative Process Models. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 131–146. Springer, Heidelberg (2012)
22. Chesani, F., Mello, P., Montali, M., Torroni, P.: Verification of Choreographies During Execution Using the Reactive Event Calculus. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 55–72. Springer, Heidelberg (2009)

Intermodeling, Queries, and Kleisli Categories

Zinovy Diskin[1,2], Tom Maibaum[1], and Krzysztof Czarnecki[2]

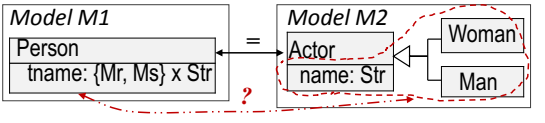
[1] Software Quality Research Lab,
McMaster University, Canada
[2] Generative Software Development Lab,
University of Waterloo, Canada
{zdiskin,kczarnec}@gsd.uwaterloo.ca, tom@maibaum.org

Abstract. Specification and maintenance of relationships between models are vital for MDE. We show that a wide class of such relationships can be specified in a compact and precise manner, if intermodel mappings are allowed to link derived model elements computed by corresponding queries. Composition of such mappings is not straightforward and requires specialized algebraic machinery. We present a formal framework, in which such machinery can be defined generically for a wide class of metamodel definitions. This enables algebraic specification of practical intermodeling scenarios, e.g., model merge.

1 Introduction

Model-driven engineering (MDE) is a prominent approach to software development, in which models of the domain and the software system are primary assets of the development process. Normally models are inter-related, perhaps in a very complex way, and to keep them consistent and use them coherently, relationships between models must be accurately specified and maintained. As noted in [1], "development of well-founded techniques and tools for the creation and maintenance of intermodel relations is at the core of MDE."

A major problem for intermodel specifications is that different models may structure the same information differently. The inset figure shows an example: model (class diagram) M1 considers Persons and their names with titles (attribute 'tname'), whereas M2 considers Actors and uses subclassing rather than titles. Suppose that classes Person in model M1 and Actor in M2 refer to the same class of entities but name them differently. We may encode this knowledge by linking the two classes with an "equality" link. In contrast, specifying "sameness" of tnames and subclassing is not straightforward and seems to be a difficult problem.



In the literature, such indirect relationships are usually specified by *correspondence rules* [2] or *expressions* [3] attached to the respective links (think of

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 163–177, 2012.

expressions replacing the question mark above). When such-annotated links are composed, it is not clear how to compose the rules; hence, it is difficult to manage scenarios that involve composition of intermodel mappings. The importance and difficulty of the mapping composition problem is well recognized in the database literature [3]; we think it will also become increasingly important in software engineering with the advancement and maturation of MDE methods.

The main goal of the paper is to demonstrate that the mapping composition problem can be solved by applying standard methods of categorical algebra, namely, the *Kleisli construction*, but applied in a non-standard way. In more detail, we present a specification framework, in which indirect links are replaced by direct links between derived rather than basic model elements. Here "derived" means that the element is computed by some operation over basic elements. We call such operations *queries*, in analogy with databases; the reader may think of some predefined query language that determines a class of legal operations and the respective derived elements. We will call links and mappings involving queries *q-links* and *q-mappings*.

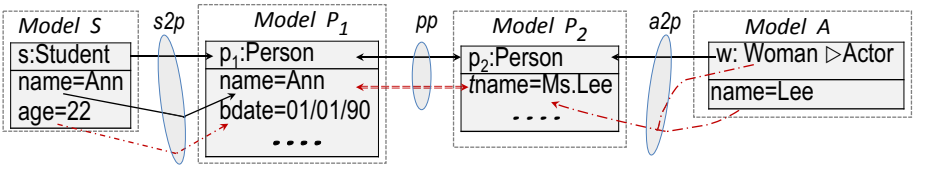
As q-mappings are sequentially composable, the universe of models and q-mappings between them can be seen as a category (in precise terms, the Kleisli category of the monad modeling the query language). Hence, intermodeling scenarios become amenable to algebraic treatment developed in category theory. We consider connection to categorical machinery to be fruitful not only theoretically, but also practically as a source of useful design patterns. In particular, we will show that q-mappings are instrumental for specifying and guiding model merge.

The paper is structured as follows. Sections 2 and 3 introduce our running example and show how q-links and q-mappings work for the problem of model merge. Section 4 explains the main points of the formalization: models' conformance to metamodels, retyping, the query mechanism and q-mappings. Section 5 briefly describes related work and Section 6 concludes.

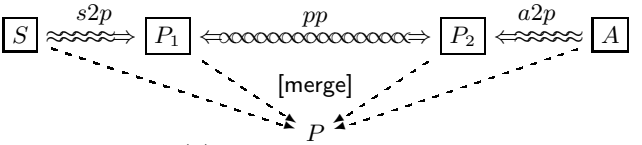
2 Running Example

To illustrate the issues we need to address, let us consider a simple example of model integration in Fig. 1. Subfigure (a) presents four object models. The expression o:Name declares an object o of class Name; the lower compartment shows o's attribute values, and ellipses in models P_1, P_2 refer to other attributes not shown. In model A, class Woman extends class Actor. When we refer to an element e (an object or an attribute) of model X, we write $e@X$. Arrows between models denote intermodel relationships explained below.

Suppose that models P_1 and P_2 are developed by two different teams charged with specifying different aspects of the same domain—different attributes of the same person in our case. The bidirectional arrow between objects $p_1@P_1$ and $p_2@P_2$ means that these objects are different representations of the same person. Model P_1 gives the first name; P_2 provides the last name and the title of the person ('tname'). We thus have a complex relationship between the attributes,



(a) four models linked informally



(b) schema of the system

Fig. 1. Running example: four models and their relationships, informally

shown by a dashed link (brown with a color display): both attributes talk about names but are complementary. Together, the two links form an informal mapping *pp* between the models.

We also assume that model P_1 is supplied with a secondary model S, representing a specific view of P_1 to be used and maintained locally at its own site (in the database jargon, S is a materialized view of P_1). Mapping $s2p$, consisting of three links, defines the view informally. Two solid-line links declare "sameness" of the respective elements. The dash-dotted link shows relatedness of the two attributes but says nothing more. Similarly, mapping $a2p$ is assumed to define model A as a view to model P_2: the solid link declares "sameness" of the two objects, and the dash-dotted link shows relatedness of their attributes and types. Mappings $s2p$, pp and $a2p$ bind all models together, so that a virtual integrated (or merged) model, say P, should say that Ms. Ann Lee is a 22 year old student and female actor born on Jan 1, 1990. Diagram Fig. 1(b) presents the merge informally: horizontal fancy arrows denote intermodel mappings, and dashed inclined arrows show mappings that embed the models into the merge.

Building model management tools capable of performing integration like above for industrial models (normally containing thousands of elements) requires clear and precise specifications of intermodel relationships. Hence, we need a framework in which intermodel mappings could be specified formally; then, operations on models and model mappings could be described in precise algebraic terms. For example, merging would appear as an instance of a formal operation that takes a diagram of models and mappings and produces an integrated model together with embeddings as shown in Fig. 1(b). We want such descriptions to be generic and applicable to a wide class of scenarios over different metamodels. Category theory does provide a suitable methodological framework (cf. [4,5,6]), e.g., homogeneous merge can be defined as the colimit of the corresponding diagram [7,8], and heterogeneity can be treated as shown in [9]. However, the basic prerequisite for applying categorical methods is that mappings and their

composition must be precisely defined. It is not straightforward even in our simple example, and we will briefly review the problems to be resolved.

Thinking in terms of elements, a mapping should be a set of links between models' elements as shown by ovals in Fig. 1(a). We can consider a link formally as a pair of elements, and it works for those links in Fig. 1(a), which are shown with solid lines. Semantically, such a link means that two elements represent the same entity in the real world. However, we cannot declare attributes 'age' in model S (we write 'age'@S) and 'bdate'@P_1 to be "the same" because, although related, they are different. Even more complex is the relationship between attribute 'tname' in base model P_2 and the view model A: it involves attributes and types (the Woman-Actor subclassing) and is shown informally by a two-to-one dash-dotted link. Finally, the dashed link between elements 'name'@P_1 and 'tname'@P_2 encodes a great deal of semantic information described above.

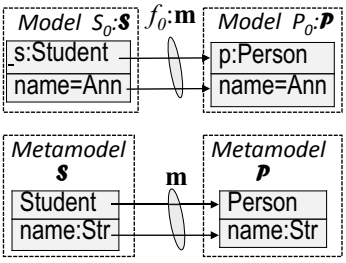
As stated in the Introduction, managing indirect links via their annotation by correspondence rules or expressions leads to difficult problems in mapping composition. In contrast, the Kleisli construction developed in categorical algebra provides a clear and concise specification framework, in which indirect relationships are modeled by q-mappings; the latter are associatively composable and constitute a category. The next section explains the basic points of the approach.

3 Intermodeling and Kleisli Mappings

We consider our running example and incrementally introduce main features of our specification framework.

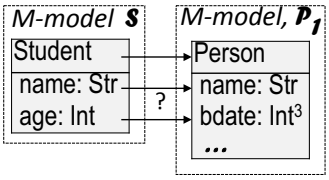
3.1 From Informal to Formal Mappings

Type Discipline. Before matching models, we need to match their metamodels. Suppose that we need to match models S_0 and P_0 over corresponding metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}$, resp. (see the inset figure on the right), linking objects s@S_0 and p@P_0 as being "the same". These objects have different types ('Student' and 'Person', resp.), however, and, with a strict type discipline, they cannot be matched. Indeed, the two objects can only be "equated" if we know that their types actually refer to the same, or, at least, overlapping, classes of real world objects. For simplicity, we assume that classes Student@$\boldsymbol{\mathcal{S}}$ and Person@$\boldsymbol{\mathcal{P}}$ refer to the same class of real world entities but are named differently; and their attributes 'name' also mean the same. To make this knowledge explicit, we match the metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}$ via mapping $\boldsymbol{m}$ as shown in the inset figure. After the metamodels are matched, we can match type-safely objects s



and p, and their attributes as well. The notation f_0:$\boldsymbol{m}$ means that each link in mapping f_0 is typed by a corresponding link in mapping $\boldsymbol{m}$. Below we will often omit metamodel postfixes next to models and model mappings if they are clear from the context.

Indirect Linking, Queries and Q-mappings. As argued above, to specify relationships between models S and P_1 in Fig. 1, we first need to relate their metamodels (the inset figure on the right). We cannot "equate" attributes 'age' and 'bdate', however. The cornerstone of our approach to intermodeling is to specify indirect relationships by *direct* links to derived elements computed with suitable queries. For example, attribute 'age' can be derived from 'bdate' with an obvious query $Q1$:



$$/\text{age} = Q1(\text{bdate}) = 2012 - \text{bdate.byear},$$

Our notation follows UML by prefixing the names of derived elements by slash; $Q1$ is the name of the query; 2012 − bdate.byear is its definition; and 'byear' denotes the year-field of the bdate-records. Now the relation between metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}_1$ is specified by three directed links, i.e., pairs, (Student, Person), (name, name) and (age, /age) as shown in the bottom of Fig. 2(a) (basic elements are shaded; the derived attribute '/age' is blank). The three links form a *direct* mapping $\boldsymbol{m_1}\colon \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{P}}_1^+$, where $\boldsymbol{\mathcal{P}}_1^+$ denotes metamodel $\boldsymbol{\mathcal{P}}_1$ augmented with derived attribute /age. Since mapping $\boldsymbol{m_1}$ is total, it indeed defines metamodel $\boldsymbol{\mathcal{S}}$ as a view of $\boldsymbol{\mathcal{P}}_1$. Query $Q1$ can be executed for any model over metamodel $\boldsymbol{\mathcal{P}}_1$, in particular, P_1 (Fig. 2(a) top), which results in augmenting model P_1 with the corresponding derived element; we denote the augmented model by P_1^+. Now model S can be directly mapped to model P_1^+ as shown in Fig. 2(a), and each link in mapping f_1 is typed by a corresponding link in mapping $\boldsymbol{m_1}$.

The same idea works for specifying mapping $a2p$ in Fig. 1. The only difference is that now derived elements are computed by a more complex query (with two *select-from-where* clauses, 'title=Ms' and 'title=Mr') as shown in Fig. 2(b): mapping $\boldsymbol{m_2}$ provides a view definition, which is executed for model P_2 and results in view model A and traceability mapping f_2. Thus, we formalize arrows $s2p$, $a2p$ in Fig. 1 as *q-mappings*, that is, mappings into models and metamodels augmented with derived elements. Ordinary mappings can be seen as degenerate q-mappings that do not use derived elements.

Links-with-New-Data via Spans. In Section 2, relationships between models P_1 and P_2 in Fig. 1 were explained informally. Fig. 3 gives a more precise description. We first introduce a new metamodel $\boldsymbol{\mathcal{P}}_{12}$ (the shaded part of metamodel $\boldsymbol{\mathcal{P}}_{12}^+$), which specifies new concepts assumed by the semantics. Then we relate these new concepts to the original ones via mappings $\boldsymbol{r}_1, \boldsymbol{r}_2$; the latter one uses derived elements. Queries $Q4_{1,2}$ are projection operations, and query Q_3 is the pairing operation. In particular, mapping $\boldsymbol{r}_2$ says that attribute 'fname'@$\boldsymbol{\mathcal{P}}_{12}^+$ does not match any attribute in model $\boldsymbol{\mathcal{P}}_2^+$, 'lname'@$\boldsymbol{\mathcal{P}}_{12}^+$ is the

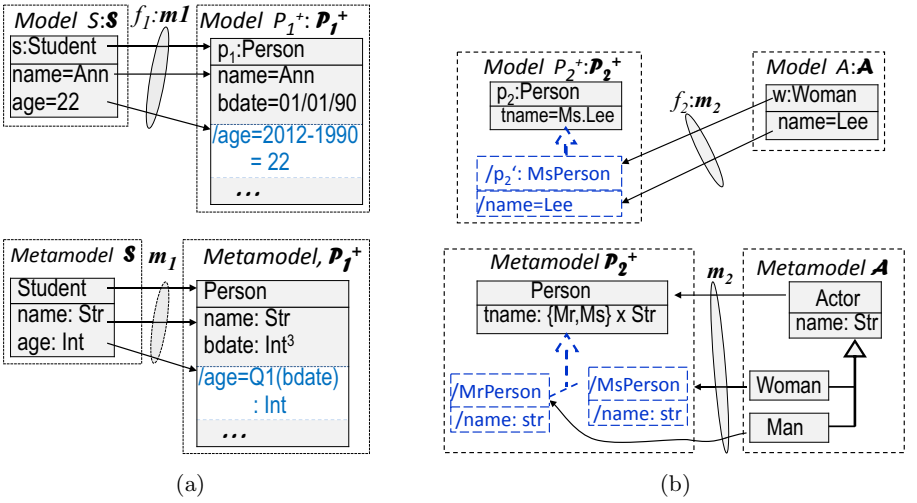


Fig. 2. Indirect matching via queries and direct mappings

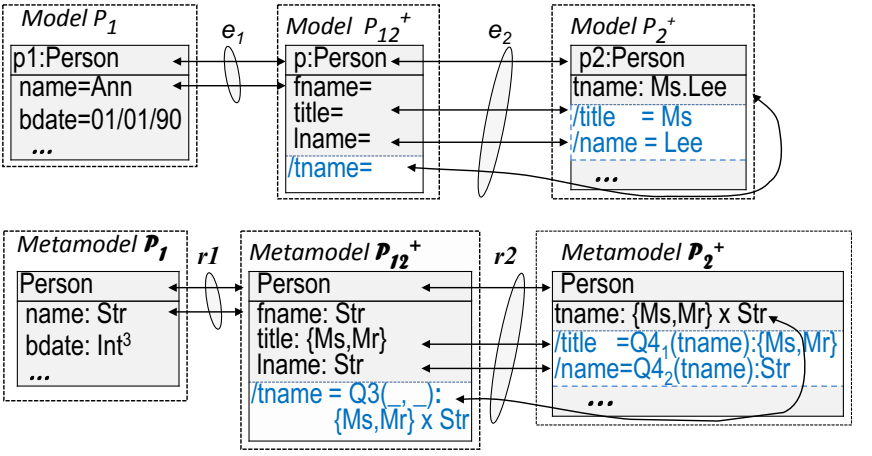


Fig. 3. Matching via spans and queries

same as '/name'@$\boldsymbol{\mathcal{P}}_2^+$ (i.e., the second component of 'tname'), and 'tname'@$\boldsymbol{\mathcal{P}}_2^+$ "equals" the pair of attributes (title, lname) in $\boldsymbol{\mathcal{P}}_{12}^+$.

On the level of models, we introduce a new model P_{12} to declare sameness of objects $p_1@P_1$ and $p_2@P_2$, and to relate their attribute slots. The new attribute slots are kept empty—they will be filled-in with the corresponding local values during the merge.

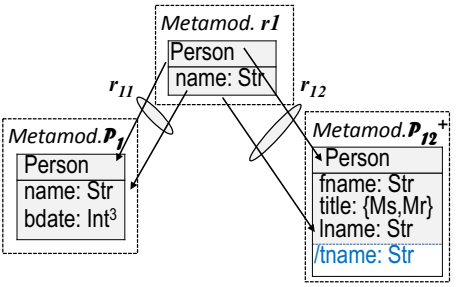
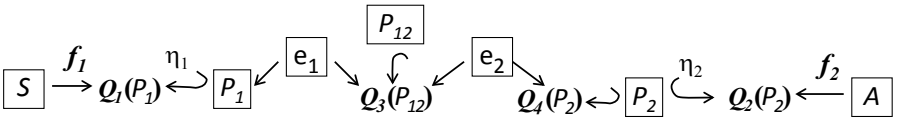


Fig. 4. Partial mappings via spans

It is well-known that algebra of totally defined functions is much simpler than that of partially defined ones. Neither of the mappings $\boldsymbol{r}_k$, e_k $(k = 1, 2)$ is total (recall that $\boldsymbol{\mathcal{P}}_2$ and P_2 may contain other attributes not shown in our diagrams). To replace these partial mappings with total ones, we apply a standard categorical construction called a *span*, as shown in Fig. 4 for mapping $\boldsymbol{r}_1$. We reify $\boldsymbol{r}_1$ as a new model $\boldsymbol{r}1$ equipped with two total projection mappings $\boldsymbol{r}_{11}$, $\boldsymbol{r}_{12}$.

Thus, we have specified all our data via models and functional q-mappings as shown in the diagram below; arrows with hooked tails denote inclusions of models into their augmentations with derived elements computed with queries Qi.



3.2 Model Merging: A Sample Multi-mapping Scenario

We want to integrate data specified by the diagram above. We focus first on merging models P_1, P_2 and P_{12} without data loss and duplication. The type discipline prescribes merging their metamodels first. To merge metamodels $\boldsymbol{\mathcal{P}}_1^+$, $\boldsymbol{\mathcal{P}}_2^+$, and $\boldsymbol{\mathcal{P}}_{12}^+$ (see Fig. 3), we take their disjoint union (no loss), and then glue together elements related by mappings $\boldsymbol{r}_{1,2}$ (to avoid duplication). The result is shown in Fig. 5(a). There is a redundancy in the merge since attribute 'tname' and pair (title, lname) are mutually derivable. We need to choose either of them as a basic structure, then the other will be derived (see Fig. 5(b1,b2)) and could be omitted from the model. We call this process *normalization*. Thus, there are two normalized merged metamodels. Amongst the three metamodels to be merged, we favor metamodel $\boldsymbol{\mathcal{P}}_{12}$ in which attribute 'tname' is considered derived from 'title' and 'lname', and hence choose metamodel $\boldsymbol{\mathcal{P}}_{n1}^+$ as the merge result (below we omit the subindex).

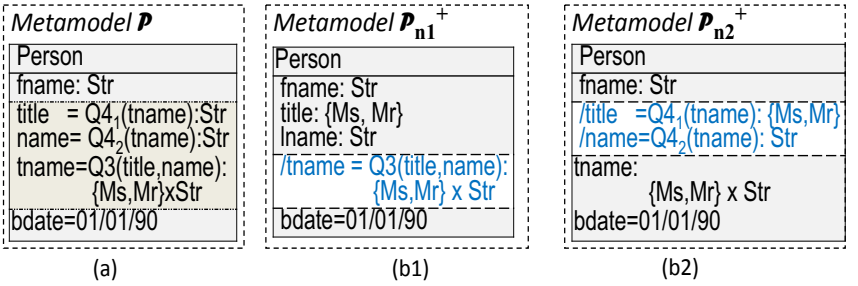


Fig. 5. Normalizing the merge

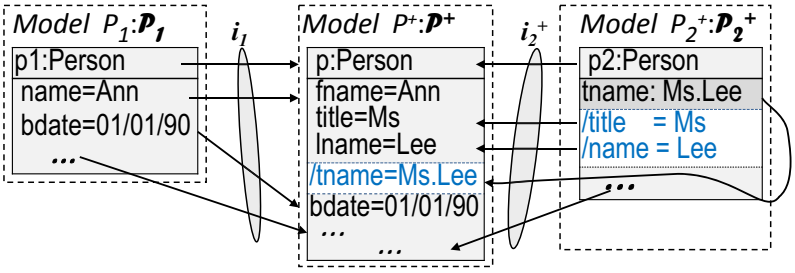


Fig. 6. Result of the merge modulo match in Fig. 3

Now take the disjoint union of models P_1^+, P_2^+, P_{12}^+ (Fig. 3), and glue together elements linked by mappings $e_{1,2}$. Note that we merge attribute slots rather than values; naming conflicts are resolved in favor of names used in metamodel $\mathcal{P}_{12}^+$. The merged model is in Fig. 6. Note how important is the interplay between basic-derived elements in mapping e_2 in Fig. 3: without these links, the merge would contain redundancies. All three component models are embedded into the merge by injective mappings $i_{1,2,3}$ (mapping i_3 is not shown).

Merge and Integration, Abstractly. The hexagon area in Fig. 7 presents the merge described above, now in an abstract way. Nodes in the diagram denote models; arrows are functional mappings, and hooked-tail arrows are inclusions. Computed mappings are shown with dashed arrows (blue if colored), and computed model P^+ is not framed.

Building model P^+ does not complete integration, however. Our system of models also has two view models, S and A, and to complete integration, we need to show how views S and A are mapped into the merge P. For this goal, we need to translate queries Q_1 and Q_2 to, resp., models P_1 and P_2 from their original models to the merge model P^+ using mappings i_1, i_2. We achieve the translation by replacing each element $x@P_k$ occurring in the expression defining query Q_k $(k = 1, 2)$ by the respective element $i_k(x)@P^+$. Then we execute the queries and augment model P^+ with the respective derived elements, as shown by inclusion mappings $\eta_k^\sharp$ $(k = 1, 2)$ within the lane (a-b) in the figure: we add to model P^+ derived attribute /age (on the left) and two derived subclasses,

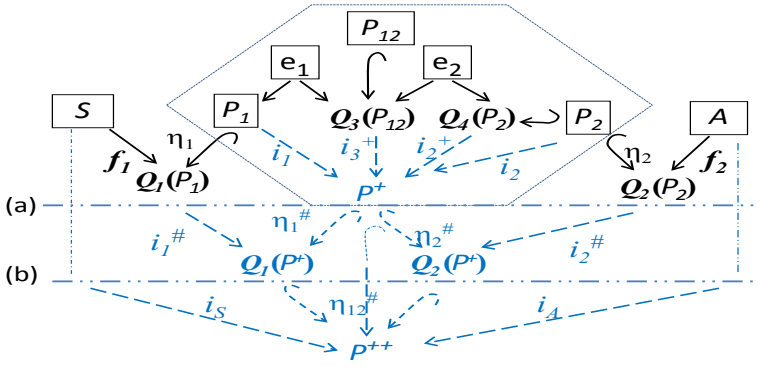


Fig. 7. The merge example, abstractly

/MrPerson and /MsPerson (on the right). Since model P^+ is embedded into its augmentations $Q_k(P^+)$ $(k = 1, 2)$, and queries Q_k preserve data embedding (are *monotonic* in database jargon), the result of executing Q_k against model P_k can be embedded into the result of executing Q_k against P^+. So, we have mappings $i_k^\sharp$ making squares $[P_k\ P^+\ Q_k(P^+)\ Q_k(P_k)]$ $(k = 1, 2)$ commutative.

Finally, we merge queries Q_1 and Q_2 to model P^+ into query Q_{12}, whose execution adds to model P^+ both derived attribute /age and the derived subclasses. We denote the resulting model by P^{++} and η_{12}: $P^+ \hookrightarrow P^{++}$ is the corresponding inclusion (see the lower diamond in Fig. 7). Now we can complete integration by building mappings i_S: $S \to P^{++}$ and i_A: $A \to P^{++}$ by sequential composition of the respective components. These mappings say that Ms. Ann Lee is a student and an actor—information that neither P^+ nor P^{++} provide.

3.3 The Kleisli Construction

The diagram in Fig. 7 is precise but looks too detailed in comparison with the informal diagram Fig. 1(b). We want to design a more compact yet still precise notation for this diagram.

Note that the diagram uses frequently the following mapping pattern

$$X \xrightarrow{f} Q(Y) \xleftarrow{\eta} Y,$$

where X, Y are, resp., the source and the target models; $Q(Y)$ is augmentation of Y with elements computed by a query Q to Y; and η is the corresponding inclusion. The key idea of the Kleisli construction developed in category theory is to view this pattern as an arrow K: $X \Rightarrow Y$ comprising two components: a query Q_K to the target Y and a functional mapping f_K: $X \to Q_K(Y)$ into the corresponding augmentation of the target. Thus, the query becomes a part of the mapping rather than of model Y, and we come to the notion of q-mapping mentioned above. We will often denote q-mappings by double-body arrows to recall that they encode both a query and a functional mapping. By a typical

abuse of notation, a q-mapping and its second component (the functional mapping) will be often denoted by the same letter; we write, say, $f\colon X \Rightarrow Y$ and $f\colon X \to Q(Y)$ using letter f for both. With this notation, the input data for integration (framed nodes and solid arrows in diagram Fig. 7) are encoded by the following diagram

$$\boxed{S} \xRightarrow{f1} \boxed{P_1} \xleftarrow{e_1} \boxed{P_{12}} \xrightarrow{e_2} \boxed{P_2} \xLeftarrow{f_2} \boxed{A}$$

where spans e_1, e_2 from Fig. 7 are encoded by arrows with bullets in the middle. Note a nice similarity between this and our original diagram Fig. 1(b)(its upper row of arrows); however, in contrast to the latter, the arrows in the diagram above have the precise meaning of q-mappings.

Finally, we want to formalize the integration procedure as an instance of the colimit operation: as well-known, the latter is a quite general pattern for "putting things together" [4]; see also [7,10,8] for concrete examples related to MDE. To realize the merge-as-colimit idea, we need to organize the universe of models and q-mappings into a category, that is, define identity q-mappings and composition of q-mappings. The former task is easy: given a model X, its identity q-mapping $\mathbb{1}_X\colon X \Rightarrow X$ comprises the empty query $Q_\varnothing$, so that $Q_\varnothing(X) = X$, and the mapping $1_X\colon X \to Q_\varnothing(X)$, which is the identity mapping of X to itself.

Composition of q-mappings is, however, non-trivial. Given two composable q-mappings $f\colon X \Rightarrow Y$ and $g\colon Y \Rightarrow Z$, defining their composition $f;g\colon X \Rightarrow Z$ is not straightforward, as shown by the diagram in Fig. 8 (ignore the two dashed arrows and their target for a moment): indeed, after unraveling, mappings f and g are simply not composable. To manage the problem, we need to apply query Q_f to model $Q_g(Z)$ and correspondingly extend mapping g as shown in the diagram. Composition of two queries is again a query, and thus pair $(f;g^{\#}, Q_f \circ Q_g)$ determines a new q-mapping from X to Z.

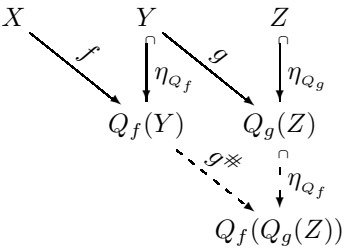


Fig. 8. Q-mapping composition

The passage from g to $g^{\#}$—the *Kleisli extension operation*—is crucial for the construction. (Note that we have used this operation in Fig. 7 too). On the level of metamodels and query definitions (syntax only), Kleisli extension is simple and amounts to term substitution. However, queries are executed for models, and an accurate formal definition of the Kleisli extension needs non-trivial work to be done. We outline the main points in the next section.

4 A Sketch of the Formal Framework

Due to space limitations, we describe very briefly the main points of the formal framework. All the details, including basic mathematical definitions we use, can be found in the accompanying technical report [11] (the TR).

4.1 Model Translation, Traceability and Fibrations

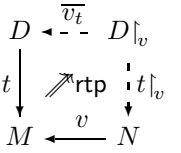
The Carrier Structure. We fix a category $\mathbb{G}$ with pullbacks, whose objects are to be thought of as (directed) graphs, or many-sorted (colored) graphs, or attributed graphs [12]. The key point is that they are definable by a metametamodel itself being a graph with, perhaps, a set of *equational* constraints. In precise categorical terms, we require $\mathbb{G}$ to be a presheaf topos [13], and hence a $\mathbb{G}$-object can be thought of as a system of sets and functions between them (e.g., a graph consists of two sets, Nd and Arr, and two functions from Arr to Nd—think of the source and the target of an arrow). It allows us to talk about elements of $\mathbb{G}$-objects, and ensures that $\mathbb{G}$ has limits, colimits, and other good properties. We will call $\mathbb{G}$-objects *'graphs'* (and as a rule skip the quotes), and write $e \in G$ to say that e is an element of graph G.

For a graph M thought of as a metamodel, an *M-model* is a pair $A = (D_A, t_A)$ with D_A a graph and $t_A\colon D_A \to M$ a mapping (arrow in category $\mathbb{G}$) to be thought of as *typing*. In a heterogeneous environment with models over different metamodels, we may say that a model A is merely an arrow $t_A\colon D_A \to M_A$ in $\mathbb{G}$, whose target M_A is called *the metamodel* of A (or the *type graph*, and the source D_A is the *data carrier* (the *data graph*). In our examples, a typing mapping for OIDs was set by colons: writing p:Person for a model A means that $p \in D_A$, Person $\in M_A$ and $t_A(p) =$ Person. For attributes, our notation covers even more, e.g., writing 'name=Ann' (nested in class Person) refers to some arrow $x\colon y \to$ Ann in graph D_A, which is mapped by t_A to arrow *value*: name $\to$ *String* in graph M_A, but names of elements x, y are not essential for us. Details can be found in [10, Sect.3].

A *model mapping* $f\colon A \to B$ is a pair of $\mathbb{G}$-mappings, $f_{\text{meta}}\colon M_A \to M_B$ and $f_{\text{data}}\colon D_A \to D_B$, commuting with typing: $f_{\text{data}}; t_B = t_A; f_{\text{meta}}$. Below we will also write f_M for f_{meta} and f_D for f_{data}. Thus, a model mapping is a commutative diagram; we usually draw typing mappings vertically and mappings f_M, f_D horizontally. We assume the latter to be monic (or injective) in $\mathbb{G}$ like in all our examples. This defines category ***Mod*** of models and model mappings.

As each model A is assigned with its metamodel M_A, and each model mapping $f\colon A \to B$ with its metamodel component $f_M\colon M_A \to M_B$, we have a projection mapping $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$, where we write ***MMod*** for either entire category $\mathbb{G}$ or for its special subcategory of 'graphs' that can serve as metamodels (e.g., all finite 'graphs'). It is easy to see that $\boldsymbol{p}$ preserves mapping composition and identities, and hence is a functor.

To take into account constraints, we need to consider metamodels as pairs $M = (G_M, C_M)$ with G_M a carrier graph and C_M a set of constraints. Then not any typing $t_A\colon D_A \to G_M$ is a model: a legal t_A must also satisfy all constraints in C_M. Correspondingly, a legal mapping $f\colon M \to N$ must be a 'graph' mapping $G_M \to G_N$ compatible with constraints in a certain sense (see [10] or [8] for details). We do not formalize constraints in this paper, but in our abstract definitions below, objects of category ***MMod*** may be understood as pairs $M = (G_M, C_M)$ as above, and ***MMod***-arrows as legal metamodel mappings.



Retyping. Any metamodel mapping $v\colon M \leftarrow N$ generates retyping of models over M into models over N as shown by the diagram on the right. If an element $e \in N$ is mapped to $v(e) \in M$, then any element in 'graph' D typed by $v(e)$, is retyped by e. Graph $D\restriction_v$ consists of such retyped elements of D, and mapping $\overline{v_t}$ traces their origin. Overall, we have an operation that takes two arrows, v and t, and produces two arrows, $\overline{v_t}$ and $t\restriction_v$, together making a commutative square as shown above.

Formally, elements of $D\restriction_v$ can be identified with pairs $(e, d) \in N \times D$ such that $v(e) = t(d)$, and mappings $t\restriction_v$ and $\overline{v_t}$ are the respective projections. The operation just described is well-known in category theory by the name *pullback* (PB) : typing arrow $t\restriction_v\colon D\restriction_v \to N$ is obtained by *pulling back* arrow t along arrow v. If we want to emphasize the vertical dimension of the operation, we will say that traceability arrow $\overline{v_t}$ is obtained by *lifting* arrow v along t.

Abstract Formulation via Fibrations. Retyping can be specified as a special property of functor $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$. That is: for an arrow $v\colon M \leftarrow N$ in $\boldsymbol{MMod}$, and an object A over M (i.e., such that $\boldsymbol{p}(A) = M$), there is an arrow $\overline{v_A}\colon A \leftarrow A\restriction_v$ over v (i.e., a commutative diagram as above), which is maximal in a certain sense amongst all arrows (commutative squares) over v. Such an arrow is called the *(weak)* $\boldsymbol{p}$*-Cartesian lifting* of arrow v, and is defined up to canonical isomorphism. Functor $\boldsymbol{p}$ with a chosen Cartesian lifting for any arrow v, which is compatible with arrow composition, is called a *split fibration* (see [14, Exercise 1.1.6]). Thus, existence of model retyping can be abstractly described by saying that we have a split fibration $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$.

We will call such a fibration an *(abstract) metamodeling framework*.

4.2 Query Mechanism via Monads and Fibrations

Background. A *monad* (in the Kleisli form) over a category $\boldsymbol{C}$ is a triple $(\mathsf{Q}, \eta, \_^{\#})$ with $\mathsf{Q}\colon \boldsymbol{C}_0 \to \boldsymbol{C}_0$ a function on $\boldsymbol{C}$-objects, η an operation that assigns to any object $X \in \boldsymbol{C}_0$ a $\boldsymbol{C}$-arrow $\eta_X\colon X \to \mathsf{Q}(X)$, and $\_^{\#}$ an operation that assigns to any $\boldsymbol{C}$-arrow $f\colon X \to \mathsf{Q}(Y)$ its *Kleisli extension* $f^{\#}\colon \mathsf{Q}(X) \to \mathsf{Q}(Y)$ such that $\eta_X; f^{\#} = f$. Two additional laws hold: $\eta_X^{\#} = 1_{\mathsf{Q}(X)}$ for all X, and $f^{\#}; g^{\#} = (f; g^{\#})^{\#}$ for all $f\colon X \to \mathsf{Q}(Y)$, $g\colon Y \to \mathsf{Q}(Z)$. In our context, if $\boldsymbol{C}$-objects are models and a monad over $\boldsymbol{C}$ is given by a query language, object $\mathsf{Q}(X)$ is to be understood as model X augmented with all derived elements computable by all possible queries. In other words, $\mathsf{Q}(X)$ is the object of queries against model X. We will identify a monad by its first component.

Any monad Q generates its *Kleisli* category $\boldsymbol{C}_{\mathsf{Q}}$. It has the same objects as $\boldsymbol{C}$, but a $\boldsymbol{C}_{\mathsf{Q}}$-arrow $f\colon X \Rightarrow Y$ is a $\boldsymbol{C}$-arrow $f\colon X \to \mathsf{Q}(Y)$. Thus, Kleilsi arrows are a special "all-queries-together" version of our q-mappings. As we have seen in Sect. 3.3, Fig. 8, composition of $\boldsymbol{C}_{\mathsf{Q}}$-arrows, say, $f\colon X \Rightarrow Y$ and $g\colon Y \Rightarrow Z$ is not immediate since f's target and g's source do not match after unraveling their definitions. The problem is resolved with the Kleisli extension operation and, moreover, the laws ensure that $\boldsymbol{C}$-objects and $\boldsymbol{C}_{\mathsf{Q}}$-arrows form a category.

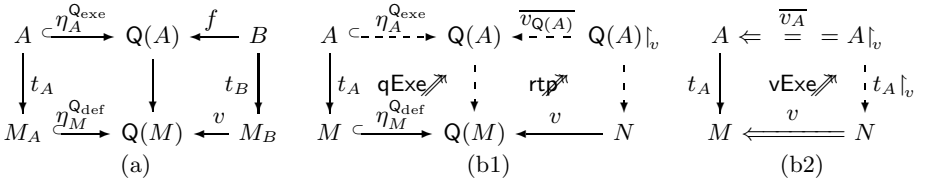


Fig. 9. Q-mappings (a) and view mechanism (b1,b2)

Lemma 1 ([15]). *If category* $\boldsymbol{C}$ *has colimits of all diagrams from a certain class* $\mathcal{D}$*, then the Kleisli category* $\boldsymbol{C}_{\mathsf{Q}}$ *has* $\mathcal{D}$*-colimits as well.*

Query Monads and Their Kleisli Categories. In the TR, we carefully motivate the following definition:

Definition 1 (main) A *monotonic query language* over an abstract metamodeling framework $\boldsymbol{p}$: $\boldsymbol{Mod} \to \boldsymbol{MMod}$ is a pair of monads $(\mathsf{Q}, \mathsf{Q}_{\mathrm{def}})$ over categories $\boldsymbol{Mod}$ and $\boldsymbol{MMod}$, resp., such that $\boldsymbol{p}$ is a monad morphism, and monad Q is $\boldsymbol{p}$-Cartesian, i.e., is compatible with the Cartesian structure of functor $\boldsymbol{p}$.

In the context of this definition, the Kleisli construction has an immediate practical interpretation. Arrows in the Kleisli category $\boldsymbol{Mod}_{\mathsf{Q}}$ are shown in Fig. 9(a). They are, in fact, our q-mappings, and we will also denote category $\boldsymbol{Mod}_{\mathsf{Q}}$ by $\boldsymbol{qMap}_{\mathsf{Q}}$(we thus switch attention from the objects of the category to its arrows). It immediately allows us to state (based on Lemma 1) that if $\mathcal{D}$-shaped configurations of models related by ordinary (not q-) model mappings are mergeable, then $\mathcal{D}$-shaped configurations of models and q-mappings are mergeable as well. For example, merge in our running example can be specified as the colimit of the diagram of Kleilsi mappings on p.10.

Metamodel-level components of q-mappings between models are arrows in $\boldsymbol{MMod}_{\mathsf{Q}_{\mathrm{def}}}$, and they are nothing but view definitions: they map elements of the source metamodel to queries against the target one Fig. 9(a). Hence, we may denote $\boldsymbol{MMod}_{\mathsf{Q}_{\mathrm{def}}}$ by $\boldsymbol{viewDef}_{\mathsf{Q}_{\mathrm{def}}}$. View definitions can be executed as shown in Fig. 9(b1): first the query is executed, and then the result is retyped along the mapping v (dashed arrows denote derived mappings).

The resulting operation of *view execution* is specified in Fig. 9(b2), where double arrows denote Kleisli mappings. Properties of the view execution mechanism are specified by Theorem 1 proved in the TR.

Theorem 1. *Let* $(\mathsf{Q}, \mathsf{Q}_{\mathrm{def}})$ *be a monotonic query language over an abstract metamodeling framework* $\boldsymbol{p}$: $\boldsymbol{Mod} \to \boldsymbol{MMod}$*. It gives rise to a split fibration* $\boldsymbol{p}_{\mathsf{Q}}$: $\boldsymbol{qMap}_{\mathsf{Q}} \to \boldsymbol{viewDef}_{\mathsf{Q}_{\mathrm{def}}}$ *between the corresponding Kleisli categories.*

Theorem 1 says that implementing view computation via querying followed by retyping is compositional. More precisely, views implemented via querying followed by retyping can be composed sequentially, and execution of the resulting composite view amounts to sequential composition of executions of its component views. Such compositionality is an evident requirement for any reasonable implementation of views, and views implemented according to our framework satisfy this requirement.

5 Related Work

Modeling inductively generated syntactic structures (term and formula algebras) by monads and Kleisli categories is well known, e.g., [16,17]. Semantic structures (algebras) then appear as Eilenberg-Moore algebras of the monad. In our approach, carriers of algebraic operations stay within the Kleilsi category. It only works for monotonic query languages, but the latter form a large, practically interesting class. (E.g, it is known that Select-Project-Join queries are monotonic.) We are not aware of a similar treatment of query languages in the literature.

Our notion of metamodeling framework is close to *specification frames* in institution theory [18]. Indeed, inverting the projection functor gives us a functor $\boldsymbol{p}_{\mathbf{Q}}^{-1}$: $\boldsymbol{viewDef}^{op}_{\mathbf{Q}_{\mathrm{def}}} \to \mathbf{Cat}$, which may be interpreted in institutional terms as mapping theories into their categories of models, and theory mappings into translation functors. The picture still lacks constraints, but adding them is not too difficult and can be found in [19]. Conversely, there are attempts to add query facilities to institutions via so called *parchments* [20]. Semantics in these attempts is modeled in a far more complex way than in our approach.

In several papers, Guerra et al. developed a systematic approach to intermodeling based on TGG (Triple Graph Grammars), see [1] for references. The query mechanism is somehow encoded in TGG-production rules, but precise relationships between this and our approach remain to be elucidated.

Our paper [9] heavily uses view definitions and views in the context of defining consistency for heterogeneous multimodels, and is actually based on constructs similar to our metamodeling framework. However, the examples therein go one step "down" in the MOF-metamodeling hierarchy in comparison with our examples here, and formalization is not provided. The combination of those structures with structures in our paper makes a two-level metamodeling framework (a fibration over a fibration); studying this structure is left for future work.

6 Conclusion

The central notion of the paper is that of a q-mapping, which maps elements in the source model to queries applied to the target model. We have shown that q-mappings provide a concise and clear specification framework for intermodeling scenarios, in particular, model merge. Composition of q-mappings is not straightforward: it requires free term substitution on the level of query definition (syntax), and actual operation composition on the level of query execution (semantics). To manage the problem, we model both syntax and semantics of a monotonic query language by a Cartesian monad over the fibration of models over their metamodels. Then q-mappings become Kleilsi mappings of the monad, and can be composed. In this way the universe of models and q-mappings gives rise to a category (the Kleisli category of the monad), providing manageable algebraic foundations for specifying intermodeling scenarios.

Acknowledgement. We are grateful for anonymous referees for valuable comments. Financial support was provided with the NECSIS project funded by Automotive Partnership Canada.

References

1. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-modelling: From Theory to Practice. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 376–391. Springer, Heidelberg (2010)
2. Romero, J., Jaen, J., Vallecillo, A.: Realizing correspondences in multi-viewpoint specifications. In: EDOC, pp. 163–172. IEEE Computer Society (2009)
3. Bernstein, P.: Applying model management to classical metadata problems. In: Proc. CIDR 2003, pp. 209–220 (2003)
4. Goguen, J.: A categorical manifesto. Mathematical Structures in Computer Science 1(1), 49–67 (1991)
5. Fiadeiro, J.: Categories for Software Engineering. Springer, Heidelberg (2004)
6. Batory, D.S., Azanza, M., Saraiva, J.: The Objects and Arrows of Computational Design. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 1–20. Springer, Heidelberg (2008)
7. Sabetzadeh, M., Easterbrook, S.M.: View merging in the presence of incompleteness and inconsistency. Requir. Eng. 11(3), 174–193 (2006)
8. Rossini, A., Rutle, A., Lamo, Y., Wolter, U.: A formalisation of the copy-modify-merge approach to version control in mde. J. Log. Algebr. Program. 79(7), 636–658 (2010)
9. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 165–179. Springer, Heidelberg (2011)
10. Diskin, Z.: Model Synchronization: Mappings, Tiles, and Categories. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 92–165. Springer, Heidelberg (2011)
11. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and Kleisli categories. Technical Report GSDLab-TR 2011-10-01, University of Waterloo (2011), http://gsd.uwaterloo.ca/QMapTR
12. Ehrig, H., Ehrig, K., Prange, U., Taenzer, G.: Fundamentals of Algebraic Graph Transformation (2006)
13. Barr, M., Wells, C.: Category theory for computing science. PrenticeHall (1995)
14. Jacobs, B.: Categorical logic and type theory. Elsevier Science Publishers (1999)
15. Manes, E.: Algebraic Theories. Springer, Heidelberg (1976)
16. Jüllig, R., Srinivas, Y.V., Liu, J.: Specware: An Advanced Evironment for the Formal Development of Complex Software Systems. In: Nivat, M., Wirsing, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 551–554. Springer, Heidelberg (1996)
17. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
18. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of ACM 39(1), 95–146 (1992)
19. Diskin, Z.: Towards generic formal semantics for consistency of heterogeneous multimodels. Technical Report GSDLAB 2011-02-01, University of Waterloo (2011)
20. Goguen, J., Burstall, R.: A Study in the Foundations of Programming Methodology: Specifications, Institutions, Charters and Parchments. In: Poigné, A., Pitt, D.H., Rydeheard, D.E., Abramsky, S. (eds.) Category Theory and Computer Programming. LNCS, vol. 240, pp. 313–333. Springer, Heidelberg (1986)

Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars

Frank Hermann[1,2,⋆], Hartmut Ehrig[1], Claudia Ermel[1], and Fernando Orejas[3]

[1] Institut für Softwaretechnik und Theoretische Informatik, TU Berlin, Germany
{frank.hermann,hartmut.ehrig,claudia.ermel}@tu-berlin.de
[2] Interdisciplinary Center for Security, Reliability and Trust, Université du Luxembourg
[3] Departament de Llenguatges i Sistemes Informàtics,
Universitat Politècnica de Catalunya, Barcelona, Spain
orejas@lsi.upc.edu

Abstract. Triple graph grammars (TGGs) have been used successfully to analyse correctness of bidirectional model transformations. Recently, also a corresponding formal approach to model synchronization has been presented, where updates on a given domain (either source or target) can be correctly (forward or backward) propagated to the other model. However, a corresponding formal approach of *concurrent* model synchronization, where a source and a target modification have to be synchronized simultaneously, has not yet been presented and analysed. This paper closes this gap taking into account that the given and propagated source or target model modifications are in conflict with each other. Our conflict resolution strategy is semi-automatic, where a formal resolution strategy – known from previous work – can be combined with a user-specific strategy.

As first result, we show *correctness* of concurrent model synchronization, that is, each result of our nondeterministic concurrent update leads to a consistent correspondence between source and target models, where consistency is defined by the TGG. As second result, we show *compatibility* of concurrent with basic model synchronization: concurrent model synchronization can realize both forward and backward propagation. The results are illustrated by a running example on updating organizational models.

Keywords: model synchronization, conflict resolution, model versioning, correctness, bidirectional model transformation, triple graph grammars.

1 Introduction

Bidirectional model transformations form a key concept for model generation and synchronization within model driven engineering (MDE, see [22]). Triple graph grammars (TGGs) have been successfully applied in several case studies for bidirectional model transformation, model integration and synchronization [20,25,14] and for the implementation of QVT [15]. Based on the work of Schürr et al. [24,25], we developed a

⋆ Supported by the National Research Fund, Luxembourg (AM2a).

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 178–193, 2012.

formal theory of TGGs [9,16], which allows handling correctness, completeness, termination and functional behaviour of model transformations. Inspired by existing synchronization tools [14] and frameworks [4], we proposed an approach for basic model synchronization in [17], showing its correctness. In that paper we studied the problem of how updates on a given domain can be correctly propagated to another model.

The aim of this paper is to provide, on this basis, also a correct TGG framework for *concurrent* model synchronization, where concurrent model updates in different domains have to be merged to a consistent solution. In this case, we have the additional problem of detecting and solving conflicts between given updates. Such conflicts may be hard to detect, since they may be caused by concurrent updates on apparently unrelated elements of the given models. Furthermore, there may be apparently contradictory updates on related elements of the given domains which may not be real conflicts.

The main idea and results of our approach are the following:

1. Model synchronization is performed by propagating the changes from one model of one domain to a corresponding model in another domain using forward and backward propagation operations. The propagated changes are compared with the given local update. Possible conflicts are resolved in a semi-automated way.
2. The operations are realized by model transformations based on TGGs [17] and tentative merge constructions solving conflicts [11]. The specified TGG also defines consistency of source and target models.
3. In general, the operation of model synchronization is nondeterministic, since there may be several conflict resolutions. The different possible solutions can be visualized to the modelers, who then decide which modifications to accept or discard.
4. The main result shows that the concurrent TGG synchronization framework is correct and compatible with the basic synchronization framework, where only single updates are considered at the same time.

Based on TGGs we present the general concurrent model synchronization framework in Sec. 2, the basic model framework in Sec. 3, and conflict resolution in Sec. 4. In Sec. 5 we combine these operations with additional auxiliary ones and present the construction of the concurrent synchronization operation, for which we show its correctness and its compatibility with the basic synchronization case in Sec. 6. All constructions and results are motivated and explained by a small case study. Finally, Secs. 7 and 8 discuss related work, conclusions and future work. Full proofs and technical details on efficiency issues and the case study are presented in a technical report [10].

2 Concurrent Model Synchronization Framework

Concurrent model synchronization aims to provide a consistent merging solution for a pair of concurrent updates that are performed on two interrelated models. This section provides a formal specification of the concurrent synchronization problem and the corresponding notion of correctness. At first, we motivate the general problem with a compact example.[1]

[1] More complex case studies are also tractable by our approach, e.g. relating class diagrams to data base models [9].

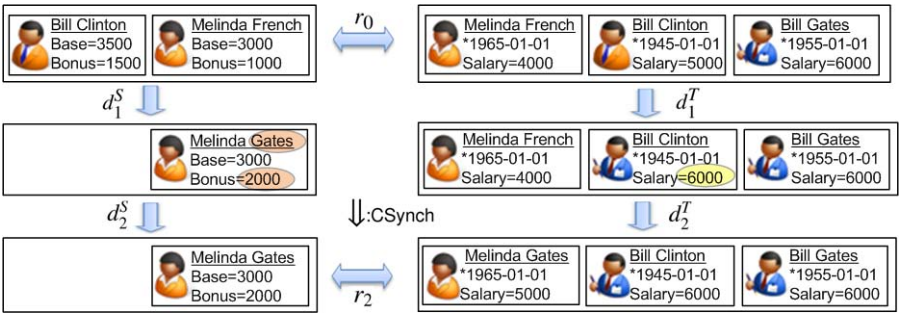


Fig. 1. Concurrent model synchronization: compact example

Example 2.1 (Concurrent model synchronization problem). Fig. 1 shows two models in correspondence that cover different aspects about employees of a company. The source model contains information about employees of the marketing department only, but shows more detailed salary information. Two model updates have to be synchronized concurrently: on the source side (model update d_1^S), Bill Clinton's node is deleted and Melinda Gates' family name changes due to her marriage; moreover, being married, her bonus is raised from 1000 to 2000. On the target side (model update d_1^T), Bill Clinton is switching from the marketing to the technical department (in the visualization in Fig. 1 this is indicated by a different role icon for Bill Clinton). His department change is combined with a salary raise from 5000 to 6000. After performing updates d_2^S and d_2^T, a "consistently integrated model" (see below) is derived that reflects as many changes as possible from the original updates in both domains and resolves inconsistencies, e.g. by computing the new Salary of Melinda Gates in the target domain as sum of the updated source attributes Base and Bonus. Note that Bill Clinton is not deleted in the target domain by the concurrent synchronization because in this case, the changes required by d_1^T could not have been realized. This conflict can be considered an apparent one. If a person leaves the marketing department, but not the company, its node should remain in the target model. Thus, a concurrent model synchronization technique has to include an adequate conflict resolution strategy.

A general way of specifying consistency between interrelated models of a source and a target domain is to provide a consistency relation that defines the consistent pairs (M^S, M^T) of source and target models. Triple graph grammars (TGGs) are a formal approach for the definition of a language of consistently integrated models [24,9]. TGGs have been applied successfully for bidirectional model transformations [25,16] and basic model synchronization [14,17], where no concurrent model updates occur.

In the framework of TGGs, an integrated model is represented by a triple graph consisting of three graphs G^S, G^C, and G^T, called source, correspondence, and target graphs, respectively, together with two mappings (graph morphisms) $s_G : G^C \to G^S$ and $t_G : G^C \to G^T$. Further concepts like attribution and inheritance can be used according to [9,8]. The two mappings in G specify a *correspondence* $r : G^S \leftrightarrow G^T$, which relates the elements of G^S with their corresponding elements of G^T and vice versa. However, it is usually sufficient to have explicit correspondences between nodes only. For simplicity, we use double arrows ($\leftrightarrow$) as an equivalent shorter notation for triple graphs, whenever the explicit correspondence graph can be omitted.

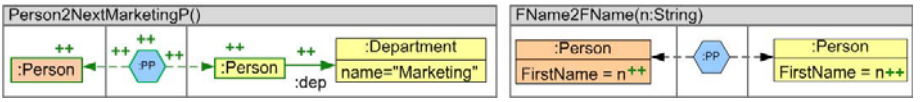


Fig. 2. Two triple rules of the TGG

Triple graphs are related by triple graph morphisms $m : G \to H$ consisting of three graph morphisms that preserve the associated correspondences (i.e., the diagrams on the right commute).

$$\begin{array}{ccccc} G = (G^S & \xleftarrow{s_G} & G^C & \xrightarrow{t_G} & G^T) \\ m\downarrow & m^S\downarrow & & m^C\downarrow & m^T\downarrow \\ H = (H^S & \xleftarrow[s_H]{} & H^C & \xrightarrow[t_H]{} & H^T) \end{array}$$

Our triple graphs are typed. This means that a type triple graph TG is given (playing the role of a metamodel) and, moreover, every triple graph G is typed by a triple graph morphism $type_G : G \to TG$. It is required that morphisms between typed triple graphs preserve the typing. For $TG = (TG^S \leftarrow TG^C \to TG^T)$, we use $VL(TG)$, $VL(TG^S)$, and $VL(TG^T)$ to denote the classes of all graphs typed over TG, TG^S, and TG^T, respectively.

A triple rule $tr = (tr^S, tr^C, tr^T)$ is an inclusion of triple graphs, represented $L \hookrightarrow R$. Notice that one or more of the rule components tr^S, tr^C, and tr^T may be empty, i.e. some elements in one domain may have no correspondence to elements in the other domain. In the example, this is the case for employees of the technical department within the target model. A triple rule is applied to a triple graph G by matching L to some subtriple graph of G via a match morphism $m : L \to G$. The result of this application is the triple graph H, where L is replaced by R in G. Technically, the result of the transformation is defined by a pushout diagram, as depicted above. This triple graph transformation (TGT) step is denoted by $G \overset{tr,m}{\Longrightarrow} H$. Moreover, triple rules can be extended by negative application conditions (NACs) for restricting their application to specific matches [16].

$$\begin{array}{ccc} L & \xhookrightarrow{tr} & R \\ m\downarrow & (PO) & \downarrow n \\ G & \xhookrightarrow[t]{} & H \end{array}$$

Example 2.2 (Triple Rules). Fig. 2 shows two triple rules of our running example using short notation, i.e., left- and right-hand side of a rule are depicted in one triple graph and the elements to be created have the label "++". Rule Person2NextMarketingP requires an existing marketing department. It creates a new person in the target component together with its corresponding person in the source component and the explicit correspondence structure. (The TGG contains a further rule (not depicted) for initially creating the marketing department.) Rule FName2FName extends two corresponding persons by their first names. There are further rules for handling the remaining attributes. In particular, the rule for attribute birth is the empty rule on the source component.

A triple graph grammar $TGG = (TG, S, TR)$ consists of a triple type graph TG, a triple start graph S and a set TR of triple rules, and generates the triple graph language $VL(TGG) \subseteq VL(TG)$. A TGG is, simultaneously, the specification of the classes of consistent source and target languages $VL_S = \{G^S \mid (G^S \leftarrow G^C \to G^T) \in VL(TGG)\}$ and $VL_T = \{G^T \mid (G^S \leftarrow G^C \to G^T) \in VL(TGG)\}$ and also of the class $C = VL(TGG) \subseteq VL(TG) = Rel$ of consistent correspondences which define the consistently integrated models. The possible model updates Δ_S and Δ_T are given by the sets of all graph modifications for the source and target domains. In our context, a model update $d : G \to G'$ is specified as a *graph modification* $d = (G \overset{i_1}{\leftarrow} I \overset{i_2}{\to} G')$. The relating

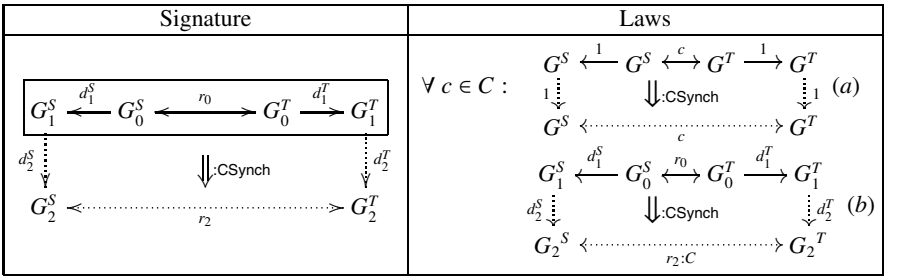


Fig. 3. Signature and laws for correct concurrent synchronization frameworks

morphisms $i_1 : I \hookrightarrow G$ and $i_2 : I \hookrightarrow G'$ are inclusions and specify which elements are deleted from G (all the elements in $G \setminus I$) and which elements are added by d (all the elements in $G' \setminus I$). While graph modifications can also be seen as triple graphs, it is conceptually important to distinguish between correspondences and updates δ.

The concurrent synchronization problem is visualized in Fig. 3, where we use solid lines for the inputs and dashed lines for the outputs. Given an integrated model $G_0 = (G_0^S \leftrightarrow G_0^T)$ and two model updates $d_1^S = (G_0^S \to G_1^S)$ and $d_1^T = (G_0^T \to G_1^T)$, the required result consists of updates $d_2^S = (G_1^S \to G_2^S)$ and $d_2^T = (G_1^T \to G_2^T)$ and a consistently integrated model $G_2 = (G_2^S \leftrightarrow G_2^T)$. The solution for this problem is a concurrent synchronization operation CSynch, which is left total but in general non-deterministic, which we indicate by a wiggly arrow "$\rightsquigarrow$" in Thm. 2.3 below. The set of inputs is given by $(Rel \otimes \Delta_S \otimes \Delta_T) = \{(r, d^S, d^T) \in Rel \times \Delta_S \times \Delta_T \mid r\colon G_0^S \leftrightarrow G_0^T, d^S : G_0^S \to {G_2}^S, d^T : G_0^T \to {G_2}^T\}$, i.e., r coincides with d^S on G_0^S and with d^T on G_0^T.

Definition 2.3 (Concurrent Synchronization Problem and Framework). *Given TGG, the* concurrent synchronization problem *is to construct a left total and nondeterministic operation* CSynch $: (Rel \otimes \Delta_S \otimes \Delta_T) \rightsquigarrow (Rel \times \Delta_S \times \Delta_T)$ *leading to the signature diagram in Fig. 3, called concurrent synchronization tile with concurrent synchronization operation* CSynch*. Given a pair* $(prem, sol) \in$ CSynch *the triple* $prem = (r_0, d_1^S, d_1^T) \in Rel \otimes \Delta_S \otimes \Delta_T$ *is called premise and* $sol = (r_2, d_2^S, d_2^T) \in Rel \times \Delta_S \times \Delta_T$ *is called a solution of the synchronization problem, written* $sol \in$ CSynch$(prem)$*. The operation* CSynch *is called* correct *with respect to consistency relation C, if laws* (a) *and* (b) *in Fig. 3 are satisfied for all solutions. Given a concurrent synchronization operation* CSynch*, the* concurrent synchronization framework *CSynch is given by* $CSynch = (TGG,$ CSynch$)$*. It is called* correct*, if operation* CSynch *is correct.*

Correctness of a concurrent synchronization operation CSynch ensures that any resulting integrated model $G_2 = (G_2^S \leftrightarrow G_2^T)$ is consistent (law (b)) and, the synchronization of an unchanged and already consistently integrated model always yields the identity of the input as output (law (a)).

3 Basic Model Synchronization Framework

We now briefly describe the basic synchronization problem and its solution [17], which is the basis for the solution for the concurrent synchronization problem in Sec. 5.

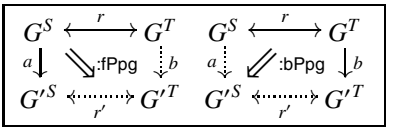


Fig. 4. Propagation operations

Given an integrated model $G^S \leftrightarrow G^T$ and an update on one domain, either G^S or G^T, the basic synchronization problem is to propagate the given changes to the other domain. This problem has been studied at a formal level by several authors (see, for instance, [12,19,26,3,28,18,5,6,17]). Many of these approaches [12,19,26,28] are state-based, meaning that they consider that the synchronization operations take as parameter the states of the models before and after the modification and yields new states of models. However, in [3,5] it is shown that state-based approaches are not adequate in general for solving the problem. Instead a number of other approaches (see, for instance, [3,18,6,17]) are δ-based, meaning that the synchronization operations take modifications as parameters and returns modifications as results. In particular, in [17], we describe a framework based on TGGs, where we include specific procedures for forward and backward propagation of modifications, proving its correctness in terms of the satisfaction of a number of laws. These results can be seen as an instantiation, in terms of TGGs, of the abstract algebraic approach presented in [6].

To be precise, according to [17], a basic synchronization framework must provide suitable left total and deterministic forward and backward propagation operations fPpg and bPpg solving this problem for any input (see Fig. 4). The input for fPpg is an integrated model $G^S \leftrightarrow G^T$ together with a source model update (graph modification) $a : G^S \to G'^S$, and the output is a target update $b : G^T \to G'^T$ together with a consistently integrated model $G'^S \leftrightarrow G'^T$. The operation bPpg behaves symmetrically to fPpg. It takes as input $G^S \leftrightarrow G^T$ and a target modification $b : G^T \to G'^T$ and it returns a source update $a : G^S \to G'^S$ together with a consistently integrated model $G'^S \leftrightarrow G'^T$. Note that determinism of these operations means that their results are uniquely determined. Note also that we require that the resulting model after a propagation operation must be consistent according to the given TGG.

We may notice that in a common tool environment, the inputs for these operations are either available directly or can be obtained. For example, the graph modification of a model update can be derived via standard difference computation.

The propagation operations are considered *correct* in [17], if they satisfy the four laws depicted in Fig. 5. Law $(a1)$ means that if the given update is the identity and the given correspondence is consistent, then fPpg changes nothing. Law $(a2)$ means that fPpg always produces consistent correspondences from consistent updated source models G'^S, where the given correspondence $r\colon G^S \leftrightarrow G^T$ is not required to be consistent. Laws $(b1)$ and $(b2)$ are the dual versions concerning bPpg.

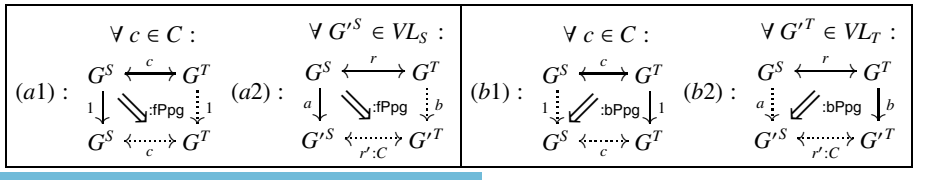


Fig. 5. Laws for correct basic synchronization frameworks

In [17], we also present specific propagation operations: Given $G^S \leftrightarrow G^T$ and the modification $a : G^S \rightarrow G'^S$, the forward propagation operation consists of three steps. In the first step, we compute an integrated model $G'^S \leftrightarrow G^T$ by deleting from the correspondence graph all the elements that were related to the elements deleted by the modification a. In the second step, we compute the largest consistently integrated model $G_0^S \leftrightarrow G_0^T$ that is included in $G'^S \leftrightarrow G^T$. Note that we do not build this model from scratch, but mark the corresponding elements in $G'^S \leftrightarrow G^T$. Moreover, we delete from G^T all the unmarked elements. Finally, using the TGG, we build the missing part of the target model that corresponds to $G'^S \setminus G_0^S$ yielding the consistently integrated model $G'^S \leftrightarrow G'^T$. Backward propagation works dually.

Remark 3.1 (Correctness of Derived Basic TGG Synchronization Framework). Correctness of the derived propagation operations fPpg, bPpg is ensured if the given TGG is equipped with *deterministic sets of operational rules* [17]. This essentially means that the forward and backward translation rules ensure functional behaviour for consistent inputs. For the technical details and automated analysis of this property using the tool AGG [27] we refer to [17], where we have shown this property for the TGG of our example and discussed the required conditions of a TGG in more detail. Note that the concurrent synchronization procedure in Sec. 5 only requires correctness of the given propagation operations and does not rely on the specific definition in [17].

4 Semi-automated Conflict Detection and Resolution

We now review the main constructions and results for conflict resolution in one domain according to [11]. Note that we apply conflict resolution either to two conflicting target model updates (one of them induced by a forward propagation operation fPpg) or to two conflicting source model updates (one of them induced by backward propagation). Hence, we here consider updates over *standard graphs* and not over triple graphs.

Two graph modifications $(G \leftarrow D_i \rightarrow H_i)$, $(i = 1, 2)$ are called *conflict-free* if they do not interfere with each other, i.e., if one modification does not delete a graph element the other one needs to perform its changes. Conflict-free graph modifications can be merged to one graph modification $(G \leftarrow D \rightarrow H)$ that realizes both original graph modifications simultaneously.

If two graph modifications are not conflict-free, then at least one conflict occurs which can be of the following kinds: (1) *delete-delete conflict*: both modifications delete the same graph element, or (2) *delete-insert conflict*: m_1 deletes a node which shall be source or target of a new edge inserted by m_2 (or vice versa). Of course, several of such conflicts may occur simultaneously. In [11], we propose a *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.[2] We call this construction *tentative* merge because usually the modeler is asked to finish the conflict resolution manually, e.g. by opting for deletion instead of insertion of certain conflicting elements. The resolution strategy to prioritize insertion over deletion preserves all model elements that are

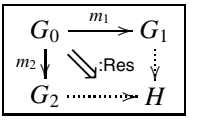
[2] Note that the conflict-free case is a special case of the tentative merge construction.

parts of conflicts and allows to highlight these elements to the user to support manual conflict resolution. We summarize the main effects of the conflict resolution strategy by Thm. 4.1 below (see also Thm. 3 in [11] for the construction).

Fact 4.1 (Conflict Resolution by Tentative Merge Construction). *Given two conflicting graph modifications* $m_i = G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$ *(i.e., they are not conflict-free). The tentative merge construction yields the merged graph modification* $m = (G \leftarrow \overline{D} \rightarrow H)$ *and resolves conflicts as follows:*

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 *(and vice versa for* (m_2, m_1) *being in delete-insert conflict).*

Note that attributed nodes which shall be deleted on the one hand and change their values on the other hand would cause delete/insert-conflicts and therefore, would not be deleted by the tentative merge construction. Attributes which are differently changed by both modifications would lead (tentatively) to attributes with two values which would cause conflicts to be solved by the modeller, since an attribute is not allowed to have more than one value at a particular time. Throughout the paper, we depict conflict resolution based on the tentative merge construction and manual modifications as shown to the right, where m_1 and m_2 are conflicting graph modifications, and H is their merge after conflict resolution. The dashed lines correspond to derived graph modifications $(G_1 \leftarrow D_3 \rightarrow H)$ and $(G_2 \leftarrow D_4 \rightarrow H)$ with interfaces D_3 and D_4.



Example 4.2 (Conflict resolution by tentative merge construction). Consider the conflict resolution square 3:Res in the upper right part of Fig. 8. The first modification $d^T_{1,F}$ deletes the node for Bill Clinton and updates the attribute values for Surname and Salary of Melinda French. The second modification d^T_1 relinks Bill Clinton's node from the marketing department to the technical department and updates his Salary attribute. The result of the tentative merge construction keeps the Bill Clinton node, due to the policy that nodes that are needed as source or target for newly inserted edges or attributes will be preserved. Technically, the attribute values are not preserved automatically. This means that the tentative merge construction only yields the structure node of "Bill Clinton" (and the updated attribute), and the modeller should confirm that the remaining attribute values should be preserved (this is necessary for the attribute values for FirstName, LastName and Birth of the "Bill Clinton" node).

Variant: As a slight variant to the above example, let us consider the case that modification d^T_1 also modifies Melinda's surname from "French" to "Smith". Since the same attribute is updated differently by both modifications, we now have two tentative attribute values for this attribute (we would indicate this by <Gates|French> as attribute value for Melinda's Surname attribute). This can be solved by the modeller, as well, who should select one attribute value.

5 Concurrent Model Synchronization with Conflict Resolution

The merge construction described in Sec. 4 cannot be applied directly to detect and solve conflicts in concurrent model synchronization. The problem here is that source and target updates occur in different graphs and not the same one. To solve this problem we use forward and backward propagation operations (Sec. 3) allowing us to see the effects of each source or target update on the other domain, so that we can apply the merge construction. In addition, we use two further operations CCS and CCT to reduce a given domain model to a maximal consistent submodel according to the TGG.

Given a source update $d_1^S : G_0^S \to G_1^S$, the consistency creating operation CCS (left part of Fig. 6) computes a maximal consistent subgraph $G_{1,C}^S \in VL_S$ of the given source model G_1^S. The resulting update from G_0^S to G_1^S is derived by update composition $d_{1,C}^S \circ d_1^S$. The dual operation CCT (right part of Fig. 6) works analogously on the target component.

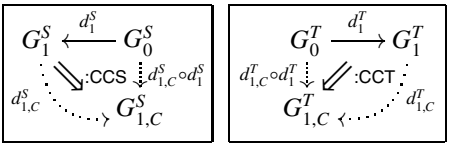


Fig. 6. Consistency creating operations

Remark 5.1 (Execution of Consistency Creating Operation CCS*).* Given a source model G_1^S, the consistency creating operation CCS is executed by computing terminated forward sequences $(H_0 \overset{tr_F^*}{\Longrightarrow} H_n)$ with $H_0 = (G_1^S \leftarrow \varnothing \rightarrow \varnothing)$. If the sets of operational rules of the TGG are deterministic (see Thm. 3.1), then backtracking is not necessary. If G_1^S is already consistent, then $G_{1,C}^S = G_1^S$, which can be checked via operation CCS. Otherwise, operation CCS is creating a maximal consistent subgraph $G_{1,C}^S$ of G_1^S. $G_{1,C}^S$ is maximal in the sense that there is no larger consistent submodel H^S of G_1^S, i.e. with $G_{1,C}^S \subseteq H^S \subseteq G_1^S$ and $H^S \in VL_S$. From the practical point of view, operation CCS is performed using forward translation rules [16], which mark in each step the elements of a given source model that have been translated so far. This construction is well defined due to the equivalence with the corresponding triple sequence $(\varnothing \overset{tr^*}{\Longrightarrow} H_n)$ via the triple rules TR of the TGG (see App. B in [10]).

The concurrent model synchronization operation CSynch derived from the given TGG is executed in five steps. Moreover, it combines operations fSynch and bSynch depending on the order in which the steps are performed. The used propagation operations fPpg, bPpg are required to be correct and we can take the derived propagation operations according to [17]. The steps of operation fSynch are depicted in Fig. 7 and Thm. 5.2 describes the steps for both operations.

Construction 5.2 (Operation fSynch **and** CSynch**).** *In the first step (operation* CCS*), a maximal consistent subgraph $G_{1,C}^S \in VL_S$ of G_1^S is computed (see Thm. 5.1). In step 2, the update $d_{1,CC}^S$ is forward propagated to the target domain via operation* fPpg*. This leads to the pair $(r_{1,F}, d_{1,F}^T)$ and thus, to the pair $(d_{1,F}^T, d_1^T)$ of target updates, which may show conflicts. Step 3 applies the conflict resolution operation* Res *including optional manual modifications (see Sec. 4). In order to ensure consistency of the resulting target model $G_{2,FC}^T$ we apply the consistency creating operation* CCT *(see Thm. 5.1) for the*

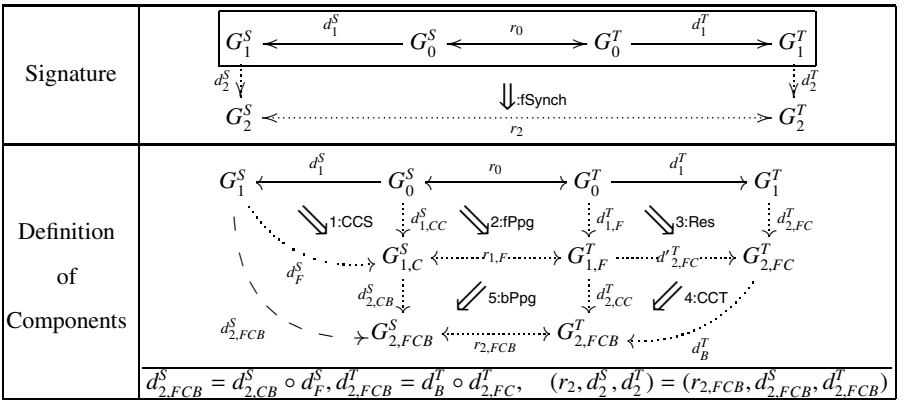


Fig. 7. Concurrent model synchronization with conflict resolution (forward case: fSynch)

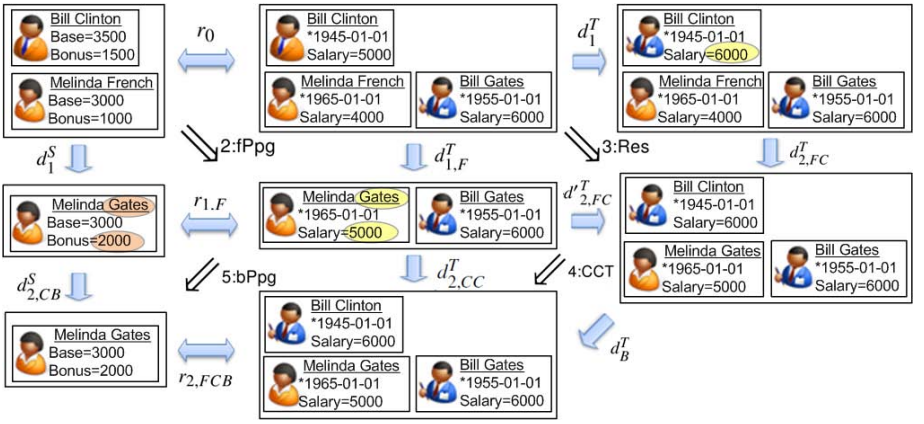


Fig. 8. Concurrent model synchronization with conflict resolution applied to organizational model

target domain and derive target model $G^T_{2,FCB} \in VL_T$ *in step 4. Finally, the derived target update* $d^T_{2,CC}$ *is backward propagated to the source domain via operation* bPpg *leading to the source model* $G^S_{2,FCB}$ *and source update* $d^S_{2,CB}$*. Altogether, we have constructed a nondeterministic solution* (r_2, d^S_2, d^T_2) *of operation* fSynch *for the premise* (r_0, d^S_1, d^T_1) *with* $(r_2, d^S_2, d^T_2) = (r_{2,FCB}, d^S_{2,FCB}, d^T_{2,FCB})$ *(see Fig. 7). The concurrent synchronization operation* bSynch *is executed analogously via the dual constructions. Starting with* CCT *in step 1, it continues via* bPpg *in step 2,* Res *in step 3,* CCS *in step 4, and finishes with* fPpg *in step 5. The non-deterministic operation* CSynch = (fSynch ∪ bSynch) *is obtained by joining the two concurrent synchronizations operations* fSynch bSynch.

Example 5.3 (Concurrent Model Synchronization with Conflict Resolution). The steps in Fig. 8 specify the execution of the concurrent synchronization in Thm. 2.1. Since the given model G^S_0 is consistent, step 1 (1:CCS) can be omitted, i.e. $G^S_{1,C} = G^S_1$ and $d^S_{1,CC} = d^S_1$. Step 2:fPpg propagates the source update to the target domain: Melinda Gates'

attributes are updated and the node representing Bill Clinton is deleted. The resolution 3:Res resolves the conflict between the target model update d_1^T and the propagated source model update on the target side $d_{1,F}^T$ (see Thm. 4.2). We assume that the modeler selected the old attribute value for Bill Clinton's birthday. Step 4:CCT does not change anything, since the model is consistent already. Finally, all elements that were introduced during the conflict resolution and concern the source domain are propagated to the source model via (5:bPpg). This concerns only the Bill Clinton node, which now is assigned to the technical department. According to the TGG, such persons are not reflected in the source model, such that the backward propagation does not change anything in the source model. The result of the concurrent model synchronization with conflict resolution is $r_{2,FCB}$, where as many as possible of both proposed update changes have been kept and insertion got priority over deletion.

Variant: Let us consider the case that both modifications d_1^T $d_{1,F}^T$ insert additionally an edge of type married between the nodes of Melinda French and Bill Gates. The conflict resolution operation 3:Res would yield two married edges between the two nodes. But the subsequent consistency creating operation 4:CCT would detect that this is an inconsistent state and would delete one of the two married edges.

Remark 5.4 (Execution and Termination of Concurrent Model Synchronization). Note that the efficiency of the execution of the concurrent synchronization operations can be significantly improved by reusing parts of previously computed transformation sequences as described in App. B in [10]. In [17], we provided sufficient static conditions that ensure termination for the propagation operations and they can be applied similarly for the consistency creating operations. Update cycles cannot occur, because the second propagation step does not lead to a new conflict.

Note that operation CSynch is nondeterministic for several reasons: the choice between fSynch and bSynch, the reduction of domain models to maximal consistent sub graphs, and the semi automated conflict resolution strategy.

Definition 5.5 (Derived Concurrent TGG Synchronization Framework). *Let* fPpg *and* bPpg *be correct basic synchronization operations for a triple graph grammar TGG and let operation* CSynch *be derived from* fPpg *and* bPpg *according to Thm. 5.2. Then, the* derived concurrent TGG synchronization framework *is given by CSynch* = (*TGG*, CSynch).

6 Correctness and Compatibility

Our main results show correctness of the derived concurrent TGG synchronization framework (Thm. 5.5) and its compatibility with the derived basic TGG synchronization framework (Sec. 3). For the proofs and technical details see App. A and B in [10]. Correctness of a concurrent model synchronization framework requires that the nondeterministic synchronization operation CSynch ensures laws (*a*) and (*b*) in Thm. 2.3. In other words, CSynch guarantees consistency of the resulting integrated model and, moreover, the synchronization of an unchanged and already consistently integrated model always yields the identity of the input as output (law (*a*)).

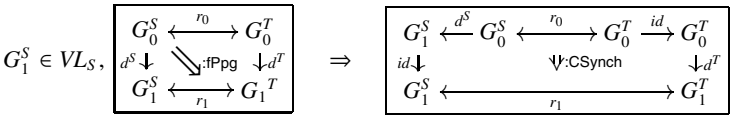


Fig. 9. Compatibility with synchronization of single updates (forward case)

According to Thm. 6.2 below, correctness of given forward and backward propagation operations ensures correctness of the concurrent model synchronization framework.

Example 6.1 (Correctness and Compatibility). In [17], we presented a suitable realization of a correct propagation operations derived from the given TGG (see Thm. 3.1). This allows us to apply the following main results Thm. 6.2 and 6.4 to our case study used as running example in Sec. 2–6.

Theorem 6.2 (Correctness of Concurrent Model Synchronization). *Let* fPpg *and* bPpg *be correct basic synchronization operations for a triple graph grammar TGG. Then, the derived concurrent TGG synchronization framework CSynch* = (*TGG*, CSynch) *(see Thm. 5.5) is correct (see Thm. 2.3).*

The second main result (Thm. 6.4 below) shows that the concurrent TGG synchronization framework is compatible with the basic synchronization framework. This means that the propagation operations (fPpg, bPpg) (see Sec. 3) provide the same result as the concurrent synchronization operation CSynch, if one update of one domain is the identity. Fig. 9 visualizes the case for the forward propagation operation fPpg. Given a forward propagation (depicted left) with solution (r_1, d^T), then a specific solution of the corresponding concurrent synchronization problem (depicted right) is given by $sol = (r_1, id, d^T)$, i.e. the resulting integrated model and the resulting updates are the same. Due to the symmetric definition of TGGs, we can show the same result concerning the backward propagation operation leading to the general result of compatibility in Thm. 6.4.

Definition 6.3 (Compatibility of Concurrent with Basic Model Synchronization). *Let* fPpg, bPpg *be basic TGG synchronization operations and let* CSynch *be a concurrent TGG synchronization operation for a given TGG. The non-deterministic synchronization operation* CSynch *is* compatible *with the propagation operations* fPpg *and* bPpg, *if the following condition holds for the forward case (see Fig. 9) and a similar one for the backward case:*

$$\forall\, (d^S, r_0) \in \Delta_S \otimes Rel,\ \text{with } (d^S : G_0^S \to G_1^S) \wedge (G_1^S \in VL_S):$$
$$(id, \text{fPpg}(d^S, r_0)) \in \text{CSynch}(d^S, r_0, id)$$

Theorem 6.4 (Compatibility of Concurrent with Basic Model Synchronization). *Let* fPpg *and* bPpg *be correct basic synchronization operations for a given TGG and let operation* CSynch *be derived from* fPpg *and* bPpg *according to Thm. 5.2. Then, the* derived concurrent TGG synchronization operation CSynch *is compatible with propagation operations* fPpg, bPpg.

7 Related Work

Triple graph grammars have been successfully applied in several case studies for bidirectional model transformation, model integration and synchronization [20,25,14] and for the implementation of QVT [15]. Several formal results are available concerning correctness, completeness, termination, functional behavior [16,13] and optimization wrt. the efficiency of their execution [16,21]. The presented approach to concurrent model synchronization is based on these results and concerns model synchronization of concurrent updates including the resolution of possible merging conflicts.

Egyed et. al [7] discuss challenges and opportunities for change propagation in multiple view systems based on model transformations concerning consistency (correctness and completeness), partiality, and the need for bidirectional change propagation and user interaction. Our presented approach based on TGGs reflects these issues. In particular, TGGs automatically ensure consistency for those consistency constraints that can be specified with a triple rule. This means that the effort for consistency checking with respect to domain language constraints is substantially reduced.

Stevens developed an abstract state-based view on symmetric model synchronization based on the concept of constraint maintainers [26], and Diskin described a more general delta-based view within the *tile algebra* framework [4,6]. These tile operations inspired the constructions for the basic synchronization operations [17], which are used for the constructions in the present paper. Concurrent updates are a central challenge in multi domain modeling as discussed in [28], where the general idea of combining propagation operations with conflict resolution is used as well. However, the paper does not focus on concrete propagation and resolution operations and requires that model updates are computed as model differences. The latter can lead to unintended results by hiding the insertion of new model elements that are similar to deleted ones.

Merging of model modifications usually means that non-conflicting parts are merged automatically, while conflicts have to be resolved manually. A survey on model versioning approaches and on (semi-automatic) conflict resolution strategies is given in [1]. A category-theoretical approach formalizing model versioning is given in [23]. Similar to our approach, modifications are considered as spans of morphisms to describe a partial mapping of models, and merging of model changes is based on pushout constructions. In contrast to [23], we consider an automatic conflict resolution strategy according to [11] that is formally defined.

8 Conclusion and Future Work

This paper combines two main concepts and results recently studied in the literature. On the one hand, basic model synchronization based on triple graph grammars (TGGs) has been studied in [17], where source model modifications can be updated to target model modifications and vice versa. On the other hand, a formal resolution strategy for conflicting model modifications has been presented in [11]. The main new contribution of this paper is the formal concept of concurrent model synchronization together with a correct procedure to implement it, where source and target modifications have to be

synchronized simultaneously, which includes conflict resolution of different source or target modifications. The main results concerning correctness and compatibility of basic and concurrent model synchronization are based on the formalization of bidirectional model transformations in the framework of TGGs [24,9,16] and the results in [17,11].

In future work, we plan to develop extended characterizations of the correctness and maximality criteria of a concurrent synchronization procedure. In this paper, correctness is defined explicitly in terms of the two laws formulated in Sec. 3 and, implicitly, in terms of the properties of compatibility with basic model synchronization proven in Thm. 6.4. We think that this can be strengthened by relating correctness of a synchronization procedure with the total or partial *realization* of the given source and target updates, for a suitable notion of realization. At a different level, we also believe that studying in detail, both from theoretical and practical viewpoints, the combination of fSynch and bSynch operations, discussed in Sec. 5, should also be a relevant matter. Finally, we also consider the possibility of taking a quite different approach for defining concurrent synchronization. In the current paper, our solution is based on implementing synchronization in terms of conflict resolution and the operations of forward and backward propagation. A completely different approach would be to obtain synchronization by the application of transformation rules, derived from the given TGG, that simultaneously implement changes associated to the source and target modifications. In particular, it would be interesting to know if the two approaches would be equally powerful, and which of them could give rise to a better implementation, on which we are working on the basis of the EMF transformation tool Henshin [2].

References

1. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. IJWIS 5(3), 271–304 (2009)
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010)
3. Barbosa, D.M.J., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: Proc. Int. Conf. on Functional Programming (ICFP 2010), pp. 193–204. ACM (2010)
4. Diskin, Z.: Model Synchronization: Mappings, Tiles, and Categories. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 92–165. Springer, Heidelberg (2011)
5. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. Journal of Object Technology 10, 6:1–6:25 (2011)
6. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 304–318. Springer, Heidelberg (2011)
7. Egyed, A., Demuth, A., Ghabi, A., Lopez-Herrejon, R., Mäder, P., Nöhrer, A., Reder, A.: Fine-Tuning Model Transformation: Change Propagation in Context of Consistency, Completeness, and Human Guidance. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 1–14. Springer, Heidelberg (2011)

8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science. Springer, Heidelberg (2006)
9. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
10. Ehrig, H., Ermel, C., Hermann, F., Orejas, F.: Concurrent model synchronization with conflict resolution based on triple graph grammars - extended version. Tech. Rep. TR 2011-14, TU Berlin, Fak. IV (2011)
11. Ehrig, H., Ermel, C., Taentzer, G.: A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 202–216. Springer, Heidelberg (2011)
12. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
13. Giese, H., Hildebrandt, S., Lambers, L.: Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. Tech. Rep. 37, Hasso Plattner Institute at the University of Potsdam (2010)
14. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8, 21–43 (2009)
15. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. Software and Systems Modeling (SoSyM) 9(1), 21–46 (2010)
16. Hermann, F., Ehrig, H., Golas, U., Orejas, F.: Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In: Proc. Int. Workshop on Model Driven Interoperability (MDI 2010), pp. 22–31. ACM (2010)
17. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y.: Correctness of Model Synchronization Based on Triple Graph Grammars. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 668–682. Springer, Heidelberg (2011)
18. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), pp. 371–384. ACM (2011)
19. Hu, Z., Mu, S.C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. Higher-Order and Symbolic Computation 21(1-2), 89–118 (2008)
20. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Tech. Rep. TR-ri-07-284, Dept. of Comp. Science, Univ. Paderborn, Germany (2007)
21. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 141–174. Springer, Heidelberg (2010)
22. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.0 formal/08-04-03, http://www.omg.org/spec/QVT/1.0/
23. 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)
24. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)

25. Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars Research Challenges, New Contributions, Open Problems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)
26. Stevens, P.: Bidirectional model transformations in qvt: semantic issues and open questions. Software and System Modeling 9(1), 7–20 (2010)
27. TFS-Group, TU Berlin: AGG (2011), http://tfs.cs.tu-berlin.de/agg
28. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Synchronizing concurrent model updates based on bidirectional transformation. Software and Systems Modeling, 1–16 (2011)

Recursive Checkonly QVT-R Transformations with General *when* and *where* Clauses via the Modal Mu Calculus

Julian Bradfield and Perdita Stevens

School of Informatics
University of Edinburgh

Abstract. In earlier work we gave a game-based semantics for checkonly QVT-R transformations. We restricted *when* and *where* clauses to be conjunctions of relation invocations only, and like the OMG standard, we did not consider cases in which a relation might (directly or indirectly) invoke itself recursively. In this paper we show how to interpret checkonly QVT-R – or any future model transformation language structured similarly – in the modal mu calculus and use its well-understood model-checking game to lift these restrictions. The interpretation via fixpoints gives a principled argument for assigning semantics to recursive transformations. We demonstrate that a particular class of recursive transformations must be ruled out due to monotonicity considerations. We demonstrate and justify a corresponding extension to the rules of the QVT-R game.

1 Introduction

QVT-R is the OMG standard *bidirectional* model transformation language [6]. It is bidirectional in the sense that, rather than simply permitting one model to be built from others, it permits changes to be propagated in any direction, something which seems to be essential in much real-world model-driven development. The same transformation can be read as specifying the circumstances under which no changes are required (checkonly mode) or as specifying exactly how one model should be modified so as to restore consistency that has been lost (enforce mode). This paper concerns checkonly mode, a thorough understanding of which is prerequisite to understanding enforce mode, because of the requirement (hippocraticness) that running a transformation in enforce mode should not modify models which are already consistent.

QVT-R has several interesting features. In particular, the fundamental way in which a QVT-R transformation is structured, using a collection of so-called *relations* connected by *when* and *where* clauses is attractive as it appears to enable the transformation to be understood by the developer in a modular way. This transformation structuring mechanism might reasonably be applied in future bidirectional model transformation languages, so it is of interest even if QVT-R itself is not ultimately successful.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 194–208, 2012.

In earlier work [7] the second author provided a game-theoretical semantics for its use in "checkonly" mode, that is, as a logic for defining predicates on pairs of models. Given a QVT-R checkonly problem instance (a transformation, together with a tuple of models to check in a given direction), we defined a formal game between two players, Verifier and Refuter, such that Verifier had a winning strategy for the game if and only if the transformation should return *true* on the given tuple of models in the stated direction. We justified the correctness of the semantics defined in this way, by referring both to [6] and to the behaviour of the most faithful QVT-R tool, ModelMorf. In that work, we did not define which player would win an infinite play of the game. Instead, we placed a restriction on the permitted transformations such that all plays of the games in our semantics would be finite; we justified this by pointing out that the OMG semantics [6] implied nothing about what the semantics in the infinite play cases should be, but we remarked that it should be possible to do better "by intriguing analogy with the modal mu calculus". Intuitively the analogy is that the interplay of *when* and *where* clauses mixes induction with coinduction; the essential character of the mu calculus is that it does the same. In this paper, we make the analogy concrete; this allows us to give semantics to many recursive QVT-R transformations, and allows us to explain why considerations of monotonicity force other recursive transformations to remain forbidden. We also use mu calculus theory to prove that extra levels of nesting of *when* and *where* clauses provide genuine extra expressivity.

When defining the semantics of QVT-R via a translation to the mu calculus, it is natural also to permit more general *when* and *where* clauses than previous work has done. The translation is an aid to clear thought, only: having made it, we extend our earlier QVT-R game so that all the transformations we can translate can also be given semantics directly by this easy-to-understand game.

Both recursion and complex clauses are useful in practice, especially where metamodels contain loops of associations; indeed, both are used in the example in [6], even though it does not give semantics of recursion.

Related work Our earlier paper [7] discusses the field of previous work on semantics for checkonly QVT-R in full. As discussed there, very few authors have interested themselves in QVT-R *as a bidirectional language.* The majority approach is to study QVT-R transformations in enforce mode only, and furthermore with the restriction that the transformation function does not take a version of the target model, only source models. The target model produced depends only on the source model and the transformation. Recursive relations typically give rise to recursion (possibly with non-termination) in the target formalism, but this does not contribute to understanding recursion in checkonly QVT-R.

More relevantly, in [3] the authors aim to generate invariants in OCL, not in order to give a formal semantics for QVT-R but to support auxiliary analysis to increase confidence in a transformation's correctness. The paper includes an example of a complex recursive QVT-R relation (in Fig 6(a), relation `ChClass-Table` is given a *where* clause `Attribute-Column(c1,t) and ChClass-Table(c1,t)`). Unfortunately, as discussed in [7], key details of the

invariant generation are elided. Looking at the example, it appears that a recursive QVT-R relation will lead to a recursive OCL constraint. The problem is thereby moved into the OCL domain, where it is still problematic: [4] in fact forbids infinite recursion. [3] does not discuss this issue, and in particular, does not specify which QVT-R transformations can be translated without producing OCL whose meaning on the relevant models is undefined.

None of the existing QVT-R tools have documented behaviour on recursive checkonly QVT-R.

2 Background

2.1 QVT-R

A transformation T is defined over a finite set of (usually two) *metamodels* (types for the input models) and, when executed in checkonly mode, can be thought of as a function from tuples of models, each conforming to the appropriate metamodel, to booleans. In any execution there is a *direction*, that is, a distinguished model which is being checked. The argument models are also known as *domains* and we will be discussing transformation execution in the direction of the kth domain. That is, the kth argument model is being checked for consistency with the others. See [7] for further discussion; here we assume some familiarity with QVT-R.

Let us discuss preliminary matters of variables, values, typing, bindings and expressions. In QVT-R these matters are prescribed, building on the MOF metamodelling discipline and OCL. The available types are the metaclasses from any of the metamodels, together with a set of base types (defined in OCL) such as booleans, strings and integers, and collections. Values are instances of these. The expression language is an extension of OCL over the metamodels. QVT-R is a typed language, with some type inference expected.

Our work will focus on the structural aspects of the transformation and will turn out to be independent of QVT-R's particular choices in these matters. We assume given sets Var of typed variables, Val of values and $Expr$ of typed expressions over variables. We write $fv(e)$ for the set of free variables in $e \in Expr$. $Constraint$ is the subset of $Expr$ consisting of expressions of type Boolean. A (partial) set of *bindings* B for a set $V \subseteq Var$ of variables will be a (partial) function $B : V \rightharpoonup Val$ satisfying the typing discipline. We write $B' \succeq B$ when $dom(B') \supseteq dom(B)$ and B' and B agree on $dom(B)$. We assume given an evaluation partial function $eval : Expr \times Binding \rightharpoonup Val$ defined on any (e, b) where $fv(e) \subseteq dom(b)$. Like [6] we will assume all transformations we consider are statically well-typed.

A transformation T is structured as a finite set of *relations* $R_1 \ldots R_n$, one or more of which are designated as *top* relations. We will use the term relation since it is that used in QVT-R, but readers should note that a QVT-R relation is not (just) a mathematical relation. Instead, a relation consists of: a unique name; for each domain a typed *domain variable* and a *pattern*; and optional *when* and *where* clauses (to be discussed shortly). We write $rel(T)$ for the set of names of

relations in T and $top(T) \subseteq rel(T)$ for the names of relations designated top. A pattern is a set of typed variables together with a constraint ("domain-local constraint") over these variables and the domain variable. A variable may occur in more than one pattern, provided that its type is the same in all.

The set of all variables used (in QVT-R declarations can be implicit) in a relation R will be denoted $vars(R)$. The subset of $vars(R)$ mentioned in the *when* clause of R is denoted $whenvars(R)$. The subset mentioned in the domains other than the kth domain is denoted $nonkvars(R)$. The set containing the domain variables is denoted $domainvars(R)$. These subsets of $vars(R)$ may overlap.

For purposes of this paper a *when* or *where* clause may contain a boolean combination of *relation invocations* and boolean constraints (from *Constraint*). Each relation invocation consists of the name of a relation together with an ordered list of argument expressions. Evaluating these expressions yields values for the domain variables of the invoked relation. The BNF (non-minimal, as it will be convenient to have all of *and*, *or* and *not*) for *where* clauses is:

$$
\begin{array}{lcll}
where(R) & := & S(e_1, \ldots e_n) & \text{where } S \in rel(T),\ e_i \in Expr \text{ and } fv(e_i) \subseteq vars(R) \\
 & | & where(R) \text{ and } where(R) \ | \ where(R) \text{ or } where(R) & \\
 & | & \text{not } where(R) \ | \ (where(R)) & \\
 & | & \phi & \text{such that } \phi \in Constraint \text{ and } fv(\phi) \subseteq vars(R)
\end{array}
$$

and the BNF for *when* is the same, substituting *when* for *where*, and *whenvars* for *vars*. The use of *whenvars* in the definition of $when(R)$ does not constrain what can be written; $v \in vars(R)$ is in $whenvars(R)$ precisely if it is used in the *when* clause. QVT-R itself uses semi-colon (in some contexts, and comma in others) for "and", but this seems unnecessarily confusing when we also want to allow other boolean connectives.

Figure 1 reproduces the moves from the game theoretic semantics of QVT-R checkonly. We refer the reader to [7] for full discussion and examples. The game G_k is played in the direction of domain k; that is, model k is being checked with respect to the other model(s).

Apart from the distinguished Initial position, positions in the game are all of the form (P, R, B, i) where: P is a player (Verifier or Refuter), indicating which player is to move from the position; R is the name of a relation from the transformation, the one in which play is currently taking place; B is a set of bindings whose domain will be specified; and i is either 1 or 2, tracking whether only one or both players have moved in the current relation. Play proceeds by the player whose turn it is to move choosing a legal move. If no legal move is available to this player, play ends and the other player wins ("you win if your opponent can't go"). The transformation returns *true* if Verifier has a winning strategy, that is, she can win however Refuter plays.

Informally, each play begins by Refuter picking a top relation to challenge and bindings for variables from the domains other than the kth and for any variables that occur in the *when* clause (Row 1). Verifier may respond by finding matching bindings from model k (Row 2) or she may counter-challenge a *when* invocation (Row 3), effectively claiming that Refuter's request for her to find

matching bindings is unreasonable because this top relation is not required to hold at his chosen bindings. If she opts to provide matching bindings, Refuter will attempt to challenge a *where* invocation (Row 4). Thus play proceeds through the transformation until one player cannot move; e.g., if Verifier successfully provides matching bindings and there is no *where* clause, it is Refuter's turn but he has no legal move, so Verifier wins the play.

Position	Next position	Notes
Initial	$(\mathsf{Verif.}, R, B, 1)$	$R \in top(T)$; $dom(B) = nonkvars(R) \cup whenvars(R)$. B is required to satisfy domain-local constraints on all domains other than k.
$(P, R, B, 1)$	$(\overline{P}, R, B', 2)$	$B' \succeq B$ and $dom(B') = vars(R)$. B' is required to satisfy domain-local constraints on all domains.
$(P, R, B, 1)$	$(\overline{P}, S, C, 1)$	$S(e_1 \dots e_n)$ is any relation invocation from the *when* clause of R; $\forall v_i \in domainvars(S).C : v_i \mapsto eval(e_i, B)$; $dom(C) = domainvars(S) \cup nonkvars(S) \cup whenvars(S)$. C is required to satisfy domain-local constraints on all domains other than k.
$(P, R, B, 2)$	$(\overline{P}, S, D, 1)$	$S(e_1 \dots e_n)$ is any relation invocation from the *where* clause of R; $\forall v_i \in domainvars(S).D : v_i \mapsto eval(e_i, B)$; $dom(D) = domainvars(S) \cup nonkvars(S) \cup whenvars(S)$. D is required to satisfy domain-local constraints on all domains other than k.

Fig. 1. Summary of the legal positions and moves of the game G_k over T: note that the first element of the Position says who picks the next move, and that we write $\overline{P}$ for the player other than P, i.e. $\overline{\mathsf{Refuter}} = \mathsf{Verifier}$ and vice versa. Recall that bindings are always required to be well-typed.

2.2 Modal Mu Calculus

The modal mu calculus [5] is a long-established and well-understood logic for specifying properties of systems, expressed as labelled transition systems. Besides the usual boolean connectives, it provides

- modal operators: $[a]\,\phi$ is true of a state s if whenever $s \xrightarrow{a} t$, ϕ is true of state t, while $\langle a \rangle \phi$ is true of a state s if there exists $s \xrightarrow{a} t$ such that ϕ is true of state t
- greatest and least fixpoints $\nu Z.\phi(Z)$ and $\mu Z.\phi(Z)$, which are formally co-inductive and inductive definitions, but which are best understood as allowing the specification of looping behaviour – infinite loops for greatest fixpoints, and finite (but unbounded) loops for least fixpoints. The combination of both fixpoints with the modal operators allows the expression of complex behaviours such as fairness.

Its semantics is most easily explained as a game between two players, Verifier and Refuter. A position, in the game to establish whether $(i, A, S, \longrightarrow)$ satisfies ϕ, is (ψ, s) where ψ is a subformula of ϕ and $s \in S$. The initial position is (ϕ, i). The top connective of ψ determines which player moves; Verifier moves if it is $\vee$ (she chooses a disjunct), $\langle a \rangle$ (she chooses an a-transition) or a maximal fixpoint or its variable (she unwinds the definition). Dually, Refuter moves otherwise. A player wins if it is their opponent's turn and the opponent has no legal move, e.g. Refuter wins if the position is $(\langle a \rangle \psi, s)$ and there is no a-transition out of state s. In an infinite play, the winner is the owner of the outermost variable unwound infinitely often (i.e. Verifier if that is a maximal fixpoint variable, otherwise Refuter).

One may think of the difference between ν and μ in terms of defaulting to true or false. In a (formal) sense, a μ formula is one where every positive claim has to be demonstrated; whereas a ν formula holds unless there is a demonstrated reason why not. See [1] for further explanation and background.

3 Connecting QVT-R and Modal Mu Calculus

We will translate a QVT-R checkonly transformation instance into a modal mu calculus model-checking instance. That is, given a QVT-R transformation T, a tuple of models $(m_1, \ldots m_n)$ and a direction k, we shall build a mu calculus formula $tr(T)$ and an LTS $lts(T, m_1, \ldots m_n, k)$ such that $(m_1, \ldots m_n)$ is consistent in the direction of the kth domain according to T iff $lts(T, m_1, \ldots m_n, k)$ satisfies $tr(T)$. Note that the LTS depends on the transformation as well as the models; this is because we choose to encode as much as possible in the LTS, leaving only the essential recursive structure to be encoded in the mu calculus formula. In particular, the LTS will capture the features of the model tuple that matter, ignoring the features that are irrelevant to this particular transformation.

Having defined our translation, we prove that this result holds for the restricted class of transformations covered by the QVT-R game. This validates the translation on the set of problem instances where a formal semantics already existed, which makes it prima facie reasonable to use the translation as the semantics of QVT-R on the full domain where it makes sense (which, as we shall see, includes many but not all transformations with recursive *when* and *where* clauses). We then propose an extension to the QVT-R game, such that the game semantics and the mu calculus translation semantics coincide everywhere. We then discuss the implications of doing so; what semantics does it assign to transformations with complex *when* and *where* clauses and/or recursive *when*/*where* structure? We will point out one decision point where two choices are possible, giving different semantics to the transformation language.

3.1 The Transition System

Apart from a distinguished initial node, nodes of the LTS we construct each consist of a pair (R, B) where $R \in rel(T)$ and $B : vars(R) \rightharpoonup Val$ is a set of

(well-typed, as always) bindings. In order to be able to handle cases where the same relation may be invoked more than once in the *when* or *where* clause of another relation, we begin by labelling each relation invocation in the static transformation text with a natural number, so that an invocation $R(e_1, \ldots, e_n)$ is replaced by $R^i(e_1, \ldots, e_n)$ for an i unique within the transformation; invoking the relation at invocation i will be modelled by a transition labelled invoke_i. Figure 2 defines the LTS formally. Note that the direction parameter k affects the meaning of *nonkvars*.

3.2 The Mu Calculus Formula

Mu calculus model checking is generally done on a version of the syntax that does not include negation. The reason is that, if negation is permitted in the language, the negation can be pushed inwards until it meets the fixpoint variables using the duality rules such as $\neg [a] \phi \equiv \langle a \rangle \neg \phi$. A formula in the mu calculus with negation is only semantically meaningful if doing this process results in all negations vanishing (using the rule $\neg\neg X \equiv X$); otherwise, the fixpoints are undefined. (Technically, it is possible for a particular formula with non-vanishing negations to be semantically meaningful, but this cannot in general be determined from the syntax.)

As mentioned in Section 2.2, the semantics of a standard mu calculus formula can be defined using a two-player model-checking game. If negation is left in the language, it corresponds to the players swapping roles, just as happens in the QVT-R game on a *when* invocation. Rather than define a version of the mu calculus game involving such player swapping, we will translate a QVT-R transformation into a mu calculus formula without negation. Our translation function will carry a boolean argument to indicate whether roles have been swapped an odd (*false*) or even (*true*) number of times.

The mu calculus formula does not represent the domain variables, the patterns or the arguments to the relation invocations, so we ignore these in our translation process: all that information is represented in the transition system, already described, and the invoke_i transitions and modalities will connect the LTS and formula appropriately. Figure 2 defines the translation process formally.

Note that *tr2* is used to translate *when* and *where* clauses, building an environment that maps relations to mu variables in the process. Relation invocations are translated using the environment if the relation has been seen before, and otherwise, using a new fixpoint.

It is easy to check that for any environment E and relation R

Lemma 1.

$$tr2_E(R, false) = \neg tr2_E(R, true)$$

□

3.3 Correctness of the Translation w.r.t. the Original QVT-R Game

Let $M_k(T, m_1, \ldots, m_n)$ be the model-checking game played on $tr(T)$ and $lts(T, m_1, \ldots, m_n, k)$. We need to establish that, if we start with a QVT-R

Input: Transformation T defined over metamodels M_i, models $m_i : M_i$, direction k.
Output: Labelled transition system $lts(T, m_i, k) = (Initial, A, S, \longrightarrow)$
Nodes:
$S = \{Initial\} \cup \{(R, B) : R \in rel(T), B : vars(R) \rightharpoonup Val\}$
Labels:
$A = \{\text{challenge}, \text{response}, \text{ext1}, \text{ext2}\} \cup \{\text{invoke}_i : i \in \mathbb{N}\}$
Transitions:

$\text{Initial} \xrightarrow{\text{challenge}} (R, B)$ if $R \in top(T)$ and $dom(B) = whenvars(R) \cup nonkvars(R)$

$(R, B) \xrightarrow{\text{response}} (R, B')$ if $dom(B) = whenvars(R) \cup nonkvars(R)$ and $B' \succeq B$ and $dom(B') = vars(R)$

$(R, B) \xrightarrow{\text{ext1}} (R, B')$ if $dom(B) = domainvars(R)$ and $B' \succeq B$ and $dom(B') = domainvars(R) \cup whenvars(R) \cup nonkvars(R)$

$(R, B) \xrightarrow{\text{ext2}} (R, B')$ if $dom(B) = domainvars(R) \cup whenvars(R) \cup nonkvars(R)$ and $B' \succeq B$ and $dom(B') = vars(R)$

$(R, B) \xrightarrow{\text{invoke}_j} (S, B')$ if S is invoked at the invocation labelled j in the *where* clause of R with arguments e_i, $dom(B) = vars(R)$ and $dom(B') = domainvars(S)$ with $\forall i \in domainvars(S).B' : v_i \mapsto eval(e_i, B)$

$(R, B) \xrightarrow{\text{invoke}_j} (S, B')$ if S is invoked at the invocation labelled j in the *when* clause of R, with arguments e_i, $dom(B) \supseteq whenvars(R)$ and $dom(B') = domainvars(S)$ with $\forall i \in domainvars(S).B' : v_i \mapsto eval(e_i, B)$

LTS definition

Input: Transformation T. **Output:** $tr(T)$ given by:

$$
\begin{array}{lll}
tr(T) & = \bigwedge_{R_i \in top(T)} tr1(R_i) & \\
tr1(R_i) & = [\text{challenge}]\,(\langle \text{response} \rangle (tr2_{\emptyset}(where(R_i), true) \vee & \\
 & \quad tr2_{\emptyset}(when(R_i), false)) & \\
tr2_E(\phi, true) & = \phi & \\
tr2_E(\phi, false) & = \neg\phi & \\
tr2_E(e \text{ and } e', true) & = tr2_E(e, true) \wedge tr2_E(e', true) & \\
tr2_E(e \text{ and } e', false) & = tr2_E(e, false) \vee tr2_E(e', false) & \\
tr2_E(e \text{ or } e', true) & = tr2_E(e, true) \vee tr2_E(e', true) & \\
tr2_E(e \text{ or } e', false) & = tr2_E(e, false) \wedge tr2_E(e', false) & \\
tr2_E(\text{not } e, b) & = tr2_E(e, \neg b) & \\
tr2_E(R^i(e_1 \ldots e_n), true) & = \langle \text{invoke}_i \rangle E[R] & \text{if } R \in \text{dom} E \\
tr2_E(R^i(e_1 \ldots e_n), true) & = \langle \text{invoke}_i \rangle \nu X.\,([\text{ext1}] & \text{otherwise} \\
 & \quad (\langle \text{ext2} \rangle tr2_{E[R \mapsto X]}(where(R), true) \vee & \\
 & \quad tr2_{E[R \mapsto X]}(when(R), false)) & \\
tr2_E(R^i(e_1 \ldots e_n), false) & = [\text{invoke}_i]\,(\neg E[R]) & \text{if } R \in \text{dom} E \\
tr2_E(R^i(e_1 \ldots e_n), false) & = [\text{invoke}_i]\,\mu X.\,(\langle \text{ext1} \rangle & \text{otherwise} \\
 & \quad ([\text{ext2}]\, tr2_{E[R \mapsto \neg X]}(where(R), false) \wedge & \\
 & \quad tr2_{E[R \mapsto \neg X]}(when(R), true)) &
\end{array}
$$

Mu calculus formula definition

Fig. 2. Definition of the translation

transformation that conforms to the constraints accepted in [7], we have indeed achieved our aim of giving equivalent semantics. Therefore let T be a transformation in which the *when–where* graph is acyclic; no relation ever invokes itself, either directly or transitively. Suppose also that all *when* and *where* clauses in T consist of conjunctions of relation invocations only. We will call such a transformation *basic*.

Notice that in this restricted case no fixpoint variable actually occurs inside the body of the corresponding μ or ν, so that (a) there is no need for the translation to retain the environment (as it will never be used) and (b) all fixpoints in the translation can be discarded. That is, we may replace $\nu X.\,\phi$ and $\mu X.\,\phi$ by ϕ (which we can be sure does not contain X free) without changing the meaning of the formula. Thus the translation *tr* yields a mu calculus formula which is equivalent to a Hennessy–Milner Logic (HML) formula in which boxes and diamonds correspond directly to challenges and responses. As required, all plays are finite, and the only winning condition is "you win if it is your opponent's turn but they have no legal move".

Theorem 1. *If T is basic, then Verifier has a winning strategy for the model-checking game M_k iff she has one on the QVT-R game G_k.*

Proof. (Sketch) The game graphs are essentially isomorphic: every position where a player of G_k has a choice corresponds to a position where the same player of M_k has a choice, these are the only choices in M_k, and the available choices correspond in turn. We only have to say "essentially" because several consecutive positions in a play of M_k (beginning with one whose formula has an "invoke" modality as the top connective) can correspond to just one position in G_k. Every position in such a sequence, except the last, has exactly one legal move from it, however, so this is unimportant. Since there are no infinite plays, every play terminates when the player whose turn it is to move has no available legal moves; the same player will win a play in G_k and the corresponding play in M_k. □

3.4 Top Relation Challenges

The translation we have given is faithful to [6,7] but readers may be wondering why we treated top relations so specially. Why is the initial challenge to a top relation so different from the invocation of a relation in a *when* or *where* clause, and why do we need two different pairs of labels in our transition system, challenge and response, and ext1 and ext2? The reason is that [6] is unequivocal that in the initial challenge to a top relation, the non-k domain variables $(\mathit{domainvars}(R) \cap \mathit{nonkvars}(R))$ are bound (chosen) at the same semantic point as the other variables in $\mathit{whenvars}(R) \cup \mathit{nonkvars}(R)$. By contrast when a relation is invoked from a *when* or *where* clause, the values of all the domain variables of the invoked relation are fixed (by the choices made for variables of the invoking relation) before values are chosen for any other non-k variables of the invoked relation. That is, in the initial challenge to a top relation, there never is a point at which the domain variables, but no others, have been bound (unless there are no others).

An alternative semantics, and one which might be considered preferable for a future language structured like QVT-R, would have Refuter challenge by picking a top relation and bindings for $domainvars(R) \cap nonkvars(R)$ only, and would then have Verifier respond by picking a binding for the kth domain variable. Then play would proceed just as though from a relation invocation with those bindings for the domain variables.

Our intuition that this might be preferable is based on the observation that a consistent pair of models would have a simpler notion of matching than in standard QVT-R. In this variant, if Verifier has a winning strategy, then given bindings for the non-k domain variables of a top relation (that is, an initial challenge by Refuter) there must be a binding for the kth domain variable (that is, a Verifier response) that matches; Verifier's choice at this initial stage must not depend on Refuter's choices of other bindings in the relation, so the matching is simpler and, perhaps, easier for a human developer to comprehend.

That this would, indeed, give different semantics for the same QVT-R transformation is demonstrated by the following relation:

```
top relation R
  domain m1 v1:V1 {}
  domain m2 v2:V2 {}
  when { S(v1,v2) }
}
```

Suppose we use a transformation with this as its only top relation, on model *m1* in which there is some model element of type *V1*, and model *m2* in which there is no model element of type *V2*, in checkonly mode in direction *m2*. In the QVT-R semantics, this will return *true*. The reason is that Refuter will be unable to pick valid bindings for $nonkvars(R) \cup whenvars(R)$ since there is no valid binding for $v2 \in whenvars(R)$ (the top level "for all valid bindings..." statement will be vacuously true). In the alternative semantics, it would return *false*, since Refuter would initially challenge with any valid binding for *v1* and Verifier would be unable to match. It would be easy to modify everything in this paper to support this alternative semantics, if desired; in fact this would simplify the translation.

4 Extending the QVT-R Game

Since not everyone will enjoy using a formal semantics of QVT-R in terms of mu calculus, we next extend the rules of the QVT-R game to match the translation. The extension to permit recursive transformations modifies only the winning conditions. To permit complex *when* and *where* clauses we need some new positions and moves.

4.1 Complex *when* and *where* Clauses

Lines 3 and 4 in Figure 1, showing the moves that involve challenging a *when* or *where* clause, are removed and replaced by the moves shown in Figure 3.

Source position	Mover	Target position	Notes
$(P, R, B, 1)$	P	$\overline{P}$ to show $when(R)$ under B	This simply indicates that player P is challenging the *when* clause of relation R, which is $when(R)$, in the presence of bindings B.
$(P, R, B, 2)$	P	$\overline{P}$ to show $where(R)$ under B	This simply indicates that player P is challenging the *where* clause of relation R, which is $where(R)$, in the presence of bindings B.
P to show Ψ_1 and Ψ_2 under B	$\overline{P}$	P to show Ψ_i under B	$i = 1, 2$: the other player chooses which conjunct P should show
P to show Ψ_1 or Ψ_2 under B	P	P to show Ψ_i under B	$i = 1, 2$: this player chooses which disjunct to show
P to show not Ψ under B	–	$\overline{P}$ to show Ψ under B	there is exactly one legal move, so it does not matter which player chooses
P to show $S(e_1 \ldots e_n)$ under B	$\overline{P}$	$(P, S, C, 1)$	$\forall v_i \in domainvars(S).C$: $v_i \mapsto eval(e_i, B)$; $dom(C) = domainvars(S) \cup nonkvars(S) \cup whenvars(S)$. C is required to satisfy domain-local constraints on all domains other than k.
P to show ϕ under B	–	–	P wins the play immediately if $eval(\phi, B) = true$ and loses the play immediately otherwise.

Fig. 3. Extensions to the moves of G_k to permit complex *when* and *where* clauses

After a player (as before) chooses to challenge a clause, we enter a sub-play, with a different form of position, to determine which relation, if any, we move to and which way round the players will be then. The positions within the subplay are of the form "P to show Ψ under B" where Ψ is a subformula of the *when* or *where* clause (recall the BNF given earlier) and B (which remains unaltered within the subplay, but is needed at the end of the subplay) is the set of bindings in force at the point where the clause was challenged. Within the subplay, as is usual in logic games, one player chooses between conjuncts, the other between disjuncts, while negation corresponds to the players swapping roles. Notice that in the simple case where *when* and *where* clauses were simply conjunctions of relation invocations, all we have done is to split up what would have been a single move according to Line 3 or 4 of Figure 1 into a sequence of moves – all by the same player who would have chosen that single move – leading eventually to the same position that was the target in the original game.

4.2 Recursive Transformations

Our translation can be applied to QVT-R transformations in which a relation does, directly or indirectly, invoke itself recursively. However, because the translation introduces negations, in certain cases it will result in an ill-formed mu calculus formula, as remarked earlier. We need a criterion that can be applied directly to the original QVT-R transformation which will ensure that the target mu calculus formula is well-formed. Fortunately this is easy.

Definition 1. *A recursion path in a QVT-R transformation is a finite sequence, whose elements may be relation names, "when", "where" or "not", such that:*

1. *the first and last elements of the sequence are the same relation name*
2. *any subsequence $R \ldots S$, where R and S are relation names and no intervening element is a relation name, corresponds to S being invoked from a when or where clause of R in the obvious way. That is, the intervening elements can only be:*
   - *"when" followed by some number $i \geq 0$ of "not"s, if S is invoked in R's when clause and the invocation is under i negations; or*
   - *"where" followed by some number $i \geq 0$ of "not"s, if S is invoked in R's where clause and the invocation is under i negations.*

Definition 2. *A QVT-R transformation is recursion-well-formed if on every recursion path the number of "not"s plus the number of "when"s is even.*

Since every not, every when, and nothing else, causes the boolean flag in the translation function to be flipped, the recursion-well-formed QVT-R transformations are precisely those that result in well-formed mu formulae.

Having decided which transformations that may lead to infinite plays to permit, we need to specify which player will win which infinite plays. In an infinite play, one or more relation names must occur infinitely often in positions of the play, that is, as the second element of a 4-tuple like those in Figure 1. Of these, let R be the one that occurs earliest in the play *not counting the positions before the first when/where invocation* (because the initial challenge to a top relation is different, as discussed in Section 3.4). Look at any 4-tuple involving R (after the first invocation). If the first element is Verifier and the last is 1, or the first element is Refuter and the last is 2 (i.e. the players are "the usual way round"), then Verifier wins this play; otherwise Refuter wins. We will get a consistent answer regardless of which position we examine, because otherwise the transformation would not have been recursion-well-formed, i.e., would have been excluded on monotonicity grounds.

Theorem 2. *The QVT-R game as modified in this section is consistent with the translation semantics.*

Proof. (Sketch) Again, the games map one-to-one onto the standard model-checking games for the mu-calculus formulae of the translation.

Remark: we could have assigned the infinite plays exactly oppositely; this would correspond to swapping μ and ν in the translation. If we did both, we would still get Theorems 1,2. This is a choice for the language designer.

5 Examples and Consequences

Fig. 4. Metamodel M and model m for examples

Consider a transformation on models conforming to the metamodel shown in Figure 4, having as its only relation the following:

```
top relation R {
  domain m1 e1:Element {}
  domain m2 e2:Element {}
  where {(e1.next is not null and e2.next is not null)
         and R(e1.next,e2.next)}
}
```

Let us play the extended game in the direction of *m2*. Refuter picks an element to bind to `e1`. Verifier must match by finding an element `e2`. Refuter will challenge the *where* clause, so the new position is "Verifier to show `(e1.next is not null and e2.next is not null) and R(e1.next,e2.next)` under B" where B records the bindings to `e1` and `e2` that the players have just made. (`(e1.next is not null and e2.next is not null)` $\in$ *Constraint*, abbreviated ϕ.) Because the top level connective of the formula in the new position is `and`, Refuter chooses a conjunct, giving new position either $p =$"Verifier to show ϕ under B" or "Verifier to show `R(e1.next,e2.next)` under B". In the first case, Verifier wins the play unless, in fact, `e1.next` or `e2.next` was null. Thus in choosing bindings for `e1` and `e2` we see that it is in Refuter's interest to choose an `e1` with no `next` if there is one – in that case he has a winning strategy – and in Verifier's interest to avoid such a choice for `e2`. In fact, Refuter can win by eventually driving play to position p (with some bindings B) iff *either* there is some Element `e` in *m1* with `e.next == null` (in which case, he may as well choose it immediately) *or* there is no loop in the `next` graph of *m2*, i.e. every element `e` eventually leads, by following `next` links, to some element `e'` with `e'.next == null`. What should happen, however, if Refuter never has the chance to drive play to a position p, because every element `e` from *m1* has non-null `next` and there is some loop in *m2* that Verifier can use to match? (Or, indeed, if he could, but does not choose to?) Refuter can repeatedly choose the "Verifier to show `R(e1.next,e2.next)` under B" position, and play will continue for ever. We consider it natural that Verifier should win such a play, and under our extended rules this is what happens; e.g. position $(\mathsf{Refuter}, R, B, 2)$ recurs.

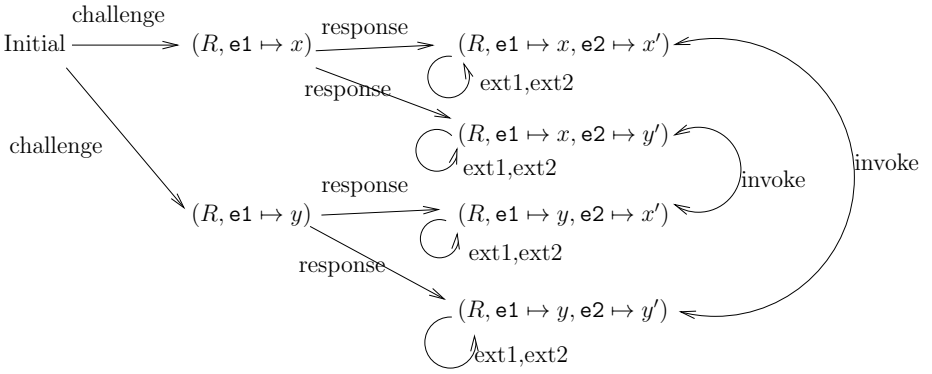


Fig. 5. Labelled transition system for example

Next we demonstrate how this example works under the translation. The translation of the transformation is

$$[\text{challenge}]\,\langle\text{response}\rangle(\phi \wedge \langle\text{invoke}\rangle\nu X.\ [\text{ext1}]\,\langle\text{ext2}\rangle(\phi \wedge X))$$

whose formal semantics corresponds closely to the above description. Specifically, if models *m1* and *m2* are both taken to be copies of m from Figure 4 (distinguished by *m2* having x', y'), the LTS is that shown in Figure 5. Any play of the model-checking game leads to one of the four right-hand LTS nodes, and then as the fixed point is repeatedly unrolled, loops between that node and the one connected to it by an invoke transition. Since our translation used a maximal fixpoint, unrolling the fixed point infinitely often is allowed and Verifier wins any play, so she has a winning strategy and our semantics says that the transformation returns *true*.

5.1 Expressiveness

In principle, a QVT-R transformation can have arbitrarily deep nesting of *when* and *where* clauses. A natural question is whether this actually adds expressivity, or whether every transformation could actually be re-expressed using at most n nestings, for some n. The corresponding question for the modal mu calculus is whether the alternation hierarchy is strict, which it is (see ([1] for details). That is, in the modal mu calculus, allowing more (semantic) nesting always does allow the expression of more properties. However, thus far we only have a translation from QVT-R to mu calculus; it could be that the image of this translation was a subset of mu calculus in which the alternation hierarchy collapsed. In fact, constructing a suitable family of examples enables us to show (see proof in Appendix of [2]):

Theorem 3. *There is no n such that every QVT-R transformation is equivalent to one with when and where clauses nested to a depth less than n.*

Clearly we inherit upper-bound complexity results also from the mu calculus; however, the complexity of mu calculus model checking is a long-open problem. It is known to be in the class NP ∩ co-NP but is not known to be in P. The problem instance size is the size of the model checking game graph; the running time of well-understood algorithms involves an exponent which depends on the alternation depth of the mu calculus formula. This is of mostly theoretical interest, however, since in practice alternation depths are typically small.

6 Conclusion

We have given a semantics to recursive checkonly QVT-R transformations with complex *when* and *where* clauses by first translating the checking problem into a modal mu calculus model checking problem, and then using this to discover a corresponding change to the rules of our earlier defined QVT-R game. Thus we end up with a semantics which is simultaneously formal and intuitive, requiring no formal training beyond the ability to follow the rules of a game. Our semantics can be instantiated with any desired metamodelling and expression languages, not just MOF and OCL.

Acknowledgements. We thank the referees for their constructive suggestions, including some that could not be implemented in this version for space reasons. The first author is partly supported by UK EPSRC grant EP/G012962/1 'Solving Parity Games and Mu-Calculi'.

References

1. Bradfield, J.C., Stirling, C.: Modal mu-calculi. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) Handbook of Modal Logic, vol. 3, pp. 721–756. Elsevier (2007)
2. Bradfield, J., Stevens, P.: Recursive checkonly QVT-R transformations with general when and where clauses via the modal mu calculus. Technical Report EDI–INF–RR–1410, University of Edinburgh, Includes Appendix (2012)
3. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2), 283–302 (2010)
4. Object Management Group. Object constraint language, version 2.0, formal/2006-05-01 (May 2006)
5. Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354 (1983)
6. OMG. MOF2.0 query/view/transformation (QVT) version 1.1. OMG document formal/2009-12-05 (2009), www.omg.org
7. Stevens, P.: A simple game-theoretic approach to checkonly QVT Relations. Journal of Software and Systems Modeling (SoSyM) (March 16, 2011), doi: 10.1007/s10270-011-0198-8

Graph Transforming Java Data*

Maarten de Mol[1], Arend Rensink[1], and James J. Hunt[2]

[1] Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, The Netherlands
{M.J.deMol,rensink}@cs.utwente.nl
[2] aicas GmbH, Karlsruhe, Germany
jjh@aicas.com

Abstract. This paper introduces an approach for adding graph transformation-based functionality to existing JAVA programs. The approach relies on a set of annotations to identify the intended graph structure, as well as on user methods to manipulate that structure, within the user's own JAVA class declarations. Other ingredients are a custom transformation language, called CHART, and a compiler from CHART to JAVA. The generated JAVA code runs against the pre-existing, annotated code.

The advantage of the approach is that it allows any JAVA program to be enhanced, non invasively, with declarative graph rules, improving clarity, conciseness and verifiability.

1 Introduction

Proponents of Graph Transformation (GT) as a modeling technique have always claimed as strong points its general applicability and its declarative nature. Many structures can naturally be regarded as graphs and their manipulation as a set of graph operations. For these reasons, GT has been advocated in particular as a vehicle for model transformation [5,14,16], a major component in the Model-Driven Engineering (MDE) paradigm. In this paper we focus on JAVA as application domain, aiming to replace JAVA code that manipulates object oriented data by declarative graph transformations.

Weak points of GT that are often quoted are its lack of efficiency and the need to transform data between the application domain and the graph domain. Though efficiency may to some degree be the price for general applicability, this does not appear to be the dominant factor. Transforming application data structures into a well defined graph format to facilitate sound transformations and then transforming the result back to a form suitable for the application is a bigger problem. These two "transfers" are themselves really model transformations in their own right, and seriously aggravate the complexity of the technique in practice, to the point of making it completely impractical for large graphs, e.g., graphs with hundreds of thousands of nodes.

One solution to the problem is to force an application to use the graph structure of the tool as a basis for its data structures. This has serious drawbacks, as the tool graph

* This work was funded by the Artemis Joint Undertaking in the CHARTER project, grant-nr. 100039. See http://charterproject.ning.com/

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 209–223, 2012.

structure may not be rich enough for the application, and existing code that may already be in place must be rewritten. This makes GT an invasive technique.

In this paper, we propose a radically different approach, aimed at JAVA, which does not share the invasive nature yet preserves the advantages of GT, including its general applicability and its declarative nature. There are three main parts of this approach.

- The graph structure (i.e., the type graph) is specified through JAVA annotations added to existing user code classes. For instance, the framework provides annotation types to specify that a given class represents a node or edge, along with edge properties such as multiplicities and ordering. As no actual code needs to be modified, we consider our method non invasive. Effectively, the JAVA annotation types constitute a type graph specification language.
- Graph manipulation, such as adding or deleting nodes or edges and updating attributes, is achieved by invoking user-provided operations. Again, these operations need to be annotated in order to express their effect in terms of the graph structure.
- Rules are written in a (textual) declarative language (called CHART), and subsequently compiled into JAVA code that runs against the aforementioned user classes, invoking the annotated methods. This obviates the need for transferring data structures to and from the graph domain. Everything is modified in place, using preexisting code.

Our approach allows components of any existing JAVA program to be replaced with declarative graph transformations, while only requiring non invasive additions to the data structures of the program. The approach was developed in the CHARTER project [4], where it is applied within three different tools that in turn make up a tool chain for the development of code for safety critical systems; see Section 4.

1.1 Related Work

There is, of course, a wealth of approaches and tools for model transformation, some of which are in fact based on graph transformation. To begin with, the OMG has published the QVT standard for model transformation [13], which is a reference point for model transformation, even though compliance with the standard is not claimed by many actual tools. A major tool effort is the ATL approach [9]; other successful tool suites are VMTS [11], VIATRA2 [17], HENSHIN [1] and FUJABA [7].

However, none of the above share the aforementioned characteristics of the CHART approach; in particular, all of them rely on their own data structures for the actual graph representation and manipulation. Although an old implementation of ATL appears to have supported the notion of a "driver" which could be tuned to a metamodelling framework and hence imaginably to our annotations, this has been abandoned in newer versions (see `http://wiki.eclipse.org/ATL/Developer_Guide#Regular_VM`).

Another transformation framework for JAVA is SITRA [12], but in contrast to the declarative rules of CHART it still requires individual rules to be written in JAVA directly.

1.2 Roadmap

In this paper we concentrate on the fundamentals of the CHART approach. In particular, in Section 2 we introduce the formal graph model used, and show how the structure

and manipulation of graphs is specified through JAVA annotations. In Section 3, we introduce the language concepts of CHART and indicate how the formal semantics of the language is defined (details can be found in [6]). Section 4 gives an overview of the use cases of the approach within the CHARTER project.

2 Graphs and Annotations

The basic idea of our approach is that the graph to be transformed is represented externally, in JAVA. In order to be able to transform this graph, its structure must be known, and operations to manipulate it must be available. This information is obtained by an automated analysis of a JAVA program. Not all necessary information can be obtained from the source code alone, however. Therefore, we have defined a language of JAVA annotations, which must be added to the code explicitly to fill the gaps.

In the following sections, we will explain how a graph structure is recognized in a JAVA program. In Section 2.1, we first provide a formal description of the graphs and type graphs that are allowed. In Sections 2.2, 2.3, 2.4 and 2.5, we describe how nodes, edges, attributes and manipulation methods are defined, respectively. In Section 2.6, we investigate the (non) invasiveness of our approach.

2.1 Graphs and Type Graphs

Our approach operates on simple graphs (nodes, binary directed edges, attributes) that are extended with basic model transformation concepts (subtyping and abstractness for nodes; multiplicity and orderedness for edges). This leads to the following formalization of type graphs:

Definition 2.1.1: *(types)*
A type (set $\mathcal{T}$) is either a node type, a set or list over a particular node type, or a basic type. The supported basic types are boolean, character, integer, real and string.

Definition 2.1.2: *(type graphs)*
A type graph is a structure $(\mathcal{T}_n, \mathcal{T}_f, src, type_f, abs, \leq_t, min, max)$, in which:

- $\mathcal{T}_n$ and $\mathcal{T}_f$ are the disjoint sets of node and field types, respectively;
- $src : \mathcal{T}_f \to \mathcal{T}_n$ associates each field type with a source node type;
- $type_f : \mathcal{T}_f \to \mathcal{T}$ determines the value type of each field type;
- $abs \subseteq \mathcal{T}_n$ is the subset of node types that are abstract;
- $\leq_t \subseteq \mathcal{T}_n \times \mathcal{T}_n$ is the subtyping relation on nodes, which must be a partial order;
- $min : \mathcal{T}_f \to \mathbb{N}$ and $max : \mathcal{T}_f \to \mathbb{N} \cup \{\mathsf{many}\}$ are the multiplicity functions.

Attributes and edges are collectively called 'fields'. If the maximum multiplicity of a field is greater than one, a single field connects a single source node to multiple targets. In that case, the targets are either stored in a set (unordered), or in a list (ordered).

A graph is straightforward instantiation of a type graph. However, we also require each graph to be rooted:

Definition 2.1.3: *(values)*
A value (set $\mathcal{V}$) is either a node, a basic value, or a set or list of nodes or values.

Definition 2.1.4: *(graphs)*
A graph (set $\mathcal{G}$) is a structure (N, r, F), in which:
- $N \subseteq \mathcal{N}$ is the set of nodes in the graph;
- $r \in N$ is the designated root node of the graph;
- $F : N \times \mathcal{T}_f \hookrightarrow \mathcal{V}$ are the field values in the graph.

Our semantics ensures that a node that gets disconnected from the root becomes invisible for further graph operations.

2.2 Definition of Node Types

A node type must be defined by annotating a *class* or *interface* with the custom @Node annotation. The name and supertypes (possible many) of the node type are determined directly by the JAVA representation. The @Node annotation has an additional argument to indicate whether the node type is abstract[1] or not.

Example 2.2.5: *(node type example)*
The following piece of code defines the abstract node type Book and the concrete node type Comic on the left, and the concrete node types Author and Picture on the right. Comic is defined to be a subtype of Book.

```
1 @Node(isAbstract = true)
2 public class Book { ...
3
4 @Node
5 public class Comic extends Book { ...
```

```
1 @Node
2 public class Author { ...
3
4 @Node
5 public class Picture { ...
```

In the JAVA code that will be produced for transformation rules, instance nodes will be represented by objects of the associated JAVA class or interface.

2.3 Definition of Edge Types

An edge type must be defined by annotating an *interface* with the custom @Edge annotation. Only the name of the edge type is determined directly by the JAVA representation. The @Edge annotation has additional arguments to define its target and multiplicity, and to indicate whether it is ordered or not.

The annotated interface does not yet define the source of the edge. Instead, it defines an abstract edge, which can be instantiated with an arbitrary source. Each node type (or more precisely, the JAVA representation of it) that implements the edge interface provides a new source for the edge type.

[1] We allow abstract classes to define concrete node types, and vice versa.

Example 2.3.6: *(edge type example)*
The following piece of code defines the edge types writtenBy, which connects a Book to its Author, and contains, which connects a Comic to its Pictures. The multiplicity indicates the number of targets that a single source may be connected to, which is exactly one for writtenBy, and arbitrarily many for contains. Also, contains is ordered.

```
1 @Edge(target = Author.class, min = 1, max = 1)
2 public interface WrittenBy { ...
3
4 @Edge(target = Picture.class, min = 0, max = Multiplicity.MANY, ordered = true)
5 public interface Contains { ...
```

Note that to make these definitions complete, Book has to implement writtenBy, and Comic has to implement contains.

In the JAVA code that will be produced for transformation rules, instance edges will not be represented on their own, but are instead assumed to be stored by their source nodes.

2.4 Definition of Attributes

An attribute type must be defined by annotating a getter *method* with the custom @AttributeGet annotation. The name, source (the node class/interface in which the method is declared) and type (the return type of the method) of the attribute are all determined directly by the JAVA representation. Our framework does not yet support multiplicity or orderedness of attributes.

Example 2.4.7: *(attribute example)*
The following piece of code defines the text attribute name for Authors on the left, and the integer attribute price for Books on the right. Note that the attribute name is obtained by removing the leading 'get' from the method name, and putting the first character in lower case.

```
1 @Node
2 public class Author {
3     ...
4     @AttributeGet
5     public String getName();
```

```
1 @Node(isAbstract = true)
2 public class Book implements writtenBy {
3     ...
4     @AttributeGet
5     public int getPrice();
```

2.5 Definition of Manipulation Methods

The JAVA code for transformation rules needs to be able to modify the graph. Instead of exposing the actual implementation, our framework defines a number of operations that may be implemented by the JAVA code. Each operation has its own custom annotation, and can only be attached to a method of a certain type and a certain behavior[2].Our framework makes the following operations available:

[2] The type is enforced statically, but the behavior is not; instead, it is currently the responsibility of the user to provide a method with the correct behavior.

- For nodes: *creation*, *visiting* all nodes of specific type.
- For all edges: *visiting* all target nodes.
- For edges (with maximum multiplicity 1): *creation*, *getting* the target node.
- For unordered edges: *addition* of a new target, *removal* of a given target, *clearing* all targets at once, *replacing* a given target with another one, checking if a given node *occurs* as a target, getting the *number* of connected targets.
- For ordered edges: *insertion* of a new target at a given index, *removal* of a given index, *getting* the target at a given index, *replacing* the target at a given index, *clearing* all targets at once, checking if a given node *occurs* as a target, getting the *number* of connected targets.
- For attributes: *getting*, *setting*.

The user is free to implement as few or as many operations as desired, but if insufficient operations are available, JAVA code cannot be produced for certain rules any more. Some operations are optimizations only, for instance computing the size of an edge is a more efficient version of increasing a counter when visiting all the targets one by one. If both are available, the efficient version will always be used.

Example 2.5.8: *(operations example)*
The following piece of code defines the manipulation methods for the contains edge type. Insertion of an element at a given index (@EdgeAdd), removal of an element at a given index (@EdgeRemove), getting an element at a given index (@EdgeGet), and visiting all elements (@EdgeVisit) are declared. Visiting makes use of the predefined class GraphVisitor, which basically wraps a method that can be applied to an edge target into an interface.

```
1  @Edge(target = Picture.class, min = 0, max = Multiplicity.MANY, ordered = true)
2  public interface Contains {
3      @EdgeAdd
4      public void insertPicture(int index, Picture picture);
5
6      @EdgeRemove
7      public void removePicture(int index);
8
9      @EdgeGet
10     public Picture getPicture(int index);
11
12     @EdgeVisit
13     public GraphVisitor.CONTINUE visit(GraphVisitor<Picture> visitor)
14     throws GraphException;
15 }
```

2.6 Invasiveness

To apply graph transformation rules on an existing JAVA program with our approach, the following modifications have to be made:

- A @Node annotation must be added to each intended node class and interface.
- For each edge type, a dedicated interface must be defined, and an @Edge annotation must be added to it.
- For each required operation, a manipulation method must be made available. This may either involve annotating an existing method with the appropriate annotation, or defining a new method and then annotating it.

These modifications only enrich existing code with meta data, and can safely be applied to any JAVA program. We therefore call our approach *non invasive*. This should not be confused with non modifying, as annotations and edge interfaces still have to be added, and additional manipulation methods need to be implemented as well.

3 Transformation Language

With our approach, we intend to make graph transformation available for JAVA programmers. For this purpose, we have defined a custom *hybrid* transformation language, called CHART. On the one hand, CHART has a textual JAVA like syntax and a sequential control structure. On the other hand, it has declarative matching and only allows graph updating by means of simultaneous assignment.

In the following sections, we will introduce CHART and briefly go into its semantics. In Section 3.1, we first present the rule based structure of CHART. In Sections 3.2, 3.3 and 3.4, we describe the main components of CHART, which are matching, updating and sequencing, respectively. In Section 3.5, we shortly describe the semantics of CHART.

3.1 Rule Structure

A CHART transformation is composed of a number of transformation rules, and can be started by invoking any one of them. Each rule has a signature, which declares its name, its input and its output. Multiple (or no) inputs and outputs are allowed, and each can be an arbitrary (typed) value (see Definition 2.1.3).

Example 3.1.1: *(rule signature)*
The following piece of code declares the rules findRich, addPicture and addPictures. Set types are denotes by trailing '{}', and list types by trailing '[]'. void denotes that a rule has no return type.

```
1 rule Author{} findRich(int price) { ...
2 rule int addPicture(Comic comic, Picture picture) { ...
3 rule void addPictures(Comic comic, Picture[] pictures) { ...
```

The body of a rule consists of a match block, an update block, a sequence block and a return block. The blocks are optional (and no more than one of each type can be used in a rule), and can only appear in the order match-update-sequence-return.

3.2 Match Blocks

A match block searches for all the possible values of a given set of match variables such that a given set of equations is satisfied. The equations are either boolean expressions, or 'foreach' statements that lift equations to all elements of a collection. A match block corresponds to the left-hand-side of a rewrite rule, represented textually.

Example 3.2.2: *(match block)*
The following match block finds all authors that have written a book with a price higher than a certain threshold. Line 3 specifies that the block searches for a set of Authors, which will be stored in the variable rich. Line 4 specifies that the equations on lines 5-6 must hold for all[3] these authors. Lines 5-6 specify that for each author there must exist a book (line 5) with a price higher than the threshold (line 6).

```
1  rule Author{} findRich(int price) {
2      Author{} rich;
3      match (rich) {
4          foreach (Author author : rich) {
5              Comic comic;
6              comic.price > price;
7          }
8      }
9      return rich;
10 }
```

A match block can either *fail*, if no valid values for the match variables can be found, or *succeed*, with a single binding for the match variables. If multiple bindings are possible, then one is chosen, and the other possibilities are thrown away. The semantics does not prescribe which binding must be returned.

3.3 Update Blocks

An update block changes the instance graph, and it is the only place where this is possible. It consists of a list of create statements, which are evaluated sequentially, and a list of update statements, which are evaluated *simultaneously*, but after the creations. Each update statement may modify a single field in the graph, and there may not be two update statements that change the same field. An update block corresponds to the differences between the right- and left-hand-side of a rewrite rule.

Example 3.3.3: *(update block)*
The following update block creates a new comic (line 7), which is the same as an existing one, but extended with one picture (line 9). The old comic is decreased in price (line 12), and the old price of the old comic is returned (lines 13 and 15). The match block ensures that the rule can only be applied to comics with a price greater than 1. Note that the right-hand-side of line 13 is evaluated in the graph before the update block, and therefore refers to the old price, instead of the new one.

[3] In our approach, a foreach over a match variable always finds the largest possible set/list only.

```
1  rule int addPicture(Comic comic, Picture picture) {
2      int old_price;
3      match () {
4          comic.price > 1;
5      }
6      update let {
7          Comic new_comic = new Comic();
8      } in {
9          new_comic.contains = comic.contains + [picture];
10         new_comic.price = comic.price;
11         new_comic.writtenBy = comic.writtenBy;
12         comic.price = comic.price − 1;
13         old_price = comic.price;
14     }
15     return old_price;
16 }
```

3.4 Sequence Blocks

A sequence block establishes flow of control, and is the only block in which other rules can be invoked. It contains sequential, JAVA like statements, such as 'if' and 'foreach' (our notation for 'for'), and custom statements for managing rule failure, such as 'try' and 'repeat' (see below).

Example 3.4.4: *(sequence block)*
The following sequence block repeatedly adds pictures to a comic by invoking addPicture (line 4). The 'pictures[2:]' that appears in line 3 is a range selection, which selects all elements starting from index 2. The try statement in line 4 is used to catch a possible failure of addPicture. Because of it, if addPicture fails, execution still continues with the next iteration of the loop.

```
1  rule void addPictures(Comic comic, Picture[] pictures) {
2      sequence {
3          foreach (Picture picture : pictures[2:]) {
4              try { addPicture(comic, picture); }
5          }
6      }
7  }
```

Because a sequence block can contain rule calls, it can also *succeed* or *fail*. If a rule call fails, one of the following things will happen:

- If the rule call is surrounded by a 'try' or 'repeat', then the failure is *caught*, and the remainder of the sequence block is executed normally.
- If the failure is not caught, and the rule in which the sequence block appears has not yet changed the graph, then the sequence block (and consequently the rule itself) *fails*. This is the same kind of failure as in the match block.

– If the failure is not caught, and the graph has already been changed, then the failure is *erroneous*, and the transformation as a whole stops with an exception. This is because our approach does not support roll-back.

3.5 Semantics

A full operational semantics of CHART is available in [6]. Below, we will restrict ourselves to a brief explanation of the top level functions of our semantics, which define the meaning of match blocks, update blocks, sequence blocks, and rule systems.

Notation 3.5.5: *(preliminaries)*
In the following, let $\mathcal{X}$ denote the set of variables, $\ell(B)$ denote the set of *lists* (i.e. ordered sequences) over B, and *Autom* denote the universe of automata.

Match blocks. A match block consists of a list of match statements, which are assumed to be defined by the *MatchStat* set. Its meaning is determined by the

$$match : \wp(\mathcal{X} \hookrightarrow \mathcal{V}) \times \ell(MatchStat) \times \mathcal{G} \to \wp(\mathcal{X} \hookrightarrow \mathcal{V})$$

function, which computes the set of *all* valid matches (=variable bindings) relative to a given input graph. An implementation only has to return one of these matches, but the semantics takes all of them into account. Later, we will enforce that all choices converge to a single output graph.

The *match* function is initialized with a single match, which is the input variable binding of the rule in which it appears. It then processes each match statement iteratively. If the statement is a match variable, then each input match is extended will all possible values of that variable. If the statement is an equation (or 'foreach'), then the input matches are filtered, and only those that satisfy the equation are kept.

Update blocks. An update block consists of a list of create statements and a list of update statements, which are assumed to be defined by the *CreateStat* and *UpdateStat* sets, respectively. Its meaning is determined by the

$$update : (\mathcal{X} \hookrightarrow \mathcal{V}) \times \ell(CreateStat) \times \ell(UpdateStat) \times \mathcal{G} \to (\mathcal{X} \hookrightarrow \mathcal{V}) \times \mathcal{G}$$

function, which modifies a variable binding and a graph. It first processes the create statements sequentially. Then the update statements are merged into one atomic update action, which is applied on the intermediate state in one go.

Sequence blocks. A sequence block consists of a list of sequence statements, which are assumed to be defined by the *SequenceStat* set. Its meaning is determined by the

$$sequence : \ell(SequenceStat) \to Autom$$

function, which builds an automaton that statically models the sequence block. It uses simplified sequence statements as alphabet, and a basic numbering for states only. The purpose of the automaton is to distinguish between success with or without changing the graph, and failure. For this purpose, it has three distinctive final states, and it models the conditions under which these states are reached. The automaton does not model any dynamic behavior, however, as its states do not include graphs or variable bindings.

Rule systems (1) A rule system consists of a set of rules, which are assumed to be defined by the *Rule* set. The meaning of a rule system is determined by the

$$\mathit{automs} : \wp(\mathit{Rule}) \times \mathit{Rule} \times \mathcal{G} \to \mathit{Autom} \times \mathit{Autom}$$

function, which computes two automata. The first is the *applier* automaton, which has tuples of graphs and variable bindings as states, and all possible applications of all available match and update blocks as (separate) transitions. The second is the *control* automaton, which is basically the combination of the sequence automata of all rules. The initial state of the applier automaton is the empty variable binding with the input graph, and the initial state of the control automaton is the initial state of the start rule.

Rule systems (2) The final purpose of a rule system is to transform an input graph into a single output graph. This is modeled by the semantic function:

$$[\![\cdot]\!] : \wp(\mathit{Rule}) \times \mathit{Rule} \times \mathcal{G} \hookrightarrow \mathcal{G}$$

First, the *product* automaton of the applier and control automata is built, which synchronizes on the rule calls and adds local state to the control automaton. In our approach, the final transformation has to terminate and be deterministic. If the product automaton is not confluent, acyclic and finite, then the transformation is therefore considered to be erroneous, and the semantic function does not yield a result. Otherwise, the semantics is given by the graph in the unique final state of the product automaton.

4 Experience and Evaluation

A major part of the effort has gone into the CHART compiler that generates the corresponding JAVA code. The compiler is called RDT, which stands for Rule Driven Transformer, and supports all features that have been described in this paper. We have used the RDT successfully on several smaller test cases, and more importantly, it is currently being used by three of our partners in the CHARTER project [4]. The tool will be made publicly available by the final project deliverable planned for April 2012.

Concretely, the following transformations have been built with the RDT:

- For testing purposes, we have implemented an interpreter for finite state automata, and an interpreter for a simplified lazy functional programming language. Some metrics are collected in Table 1.
- A collection of CHART rules have been produced by ATEGO to replace the JAVA code generator within ARTISAN STUDIO [2].
- A collection of CHART rules are being developed by AICAS for the purpose of optimisations and machine code generation in the JAMAICAVM byte code compiler [15]. This application is discussed in some more detail below.
- Part of the code simplification from JAVA to Static Single Assignment form within the KEY tool [3] is scheduled to be replaced by CHART rules.

These examples show that the technology can already be applied in practice. In all cases, the CHART rules are more concise than the JAVA code. The JAMAICAVM and KEY examples also show that the RDT can successfully be connected to an existing JAVA program. In the other examples, a custom JAVA program was built explicitly.

Table 1. metrics for the functional interpreter case

JAVA data (represent functional program)	21 classes, 11 interfaces, 1448 lines of code
added annotations	19 nodes, 11 edges, 49 methods, 3 auxiliary
CHART rules	53 rules, 1024 lines of code
produced JAVA code	53 classes, 7667 lines of code
analysis and compilation time	approx. 4,5 seconds (2GHZ, 4GB laptop)
execution time (50 primes computed with sieve)	approx. 1,5 seconds (2GHZ, 4GB laptop)

4.1 Using the RDT in JAMAICAVM

A complex application of the RDT is its application in the JAMAICAVM byte code compiler. This application demonstrates the strength of the synthesis of a strongly typed object-oriented programming language with a domain specific graph transformation language. The compiler implementation takes advantage of JAVA for implementing the basic graph operations, and uses the RDT for deciding what transforms should be applied (match clause), when a transform should be applied (sequence clause), and what change a transform should make (update clause).

The most general CHART rules are used in the optimization of the intermediate representation. These rules implement standard compiler optimizations such as:

- unnecessary node removal,
- expression simplification,
- duplicate check removal,
- common subexpression elimination,
- method inlining,
- loop inversion, and
- loop expression hoisting.

There are not that many optimizations, but each optimization takes in general several rules to implement it. Each rule has about 10 to 20 lines of RDT code. The generated code is approximately ten times as long. Hand coding might bring a factor of two improvement, but that would still be five times large than the CHART code.

There are many more CHART rules for translating the intermediate representation to the low-level representation: each intermediate instruction requires its own rule. These rules are simpler, so they tend to be shorter than the optimization ones. Still there is a similar ten fold expansion of code when these rules are compiled.

CHART rules are also used in the optimization of the low-level representation. These rules tend to be more instruction dependent. Some of the simpler intermediate optimizations are applicable on the low-level too, because they do not depend on the actual instructions and operate on an abstract representation of the graph. This works because both graphs share a common parametrized subclass structure using JAVA generics. Again, a ten fold expansion is typical.

Performance measurements have not yet been made, but there is no noticeable slowdown for the couple of optimizations that have been converted to rules. In fact, the new compiler is faster than the previous one. However, this is probably due to improvements in the graph structure. Certainly, the performance is within acceptable bounds.

In general, CHART rules are easier to reason about than JAVA code for two reasons. Firstly, the code is much shorter. Secondly, the language itself is declarative instead of imperative. The next challenge will be to provide theory and methods for reasoning about the correctness of rules written in CHART.

4.2 Evaluation

The cases reported above provide first evidence of the advantages and disadvantages of the CHART approach. We can make the following observations:

- The approach is flexible enough to be applicable in several, quite different contexts: model-to-text generation for ARTISAN, code compilation and optimisation for JAMAICAVM and text-to-text transformation in the case of KEY. The latter is done on the basis of its JAVA syntax tree structure.
- The non invasive nature of the approach enables a partial or stepwise adoption of CHART. Indeed, the fact that all data stay within the user domain was the reason to adopt CHART for KEY, where it was initially not foreseen in the project proposal.
- CHART enhances productivity by a factor of ten, measured in lines of code. This metric should be taken with a grain of salt as the generated code is less compact than hand-written code for the same purpose would have been; however, as argued in Section 4.1, even taking this into account the ratio in lines of code is a factor 5.
- The generated code been applied to very large graphs (in the order of 10^5 – 10^6 nodes) with a performance comparable to the replaced handwritten JAVA code.

All in all, we feel that these are very encouraging results.

4.3 Future Work

Although, as shown above, CHART has already proved its worth in practice, there is obviously still a lot of work that can be done to strengthen both the formal underpinnings and the practical usability.

- The formal semantics presented in this paper enables reasoning on the level of the CHART rules. As a next step, we intend to develop this into a theory that allows the CHART programmer to deduce confluence and termination of his rule system. A more ambitious goal is to be able to prove semantic preservation of model transformations in CHART (see, e.g., [8]).
- Based on the formal semantics, we plan to formally verify that the RDT actually produces correct code. Code correctness can be addressed on different levels: firstly, by requiring it to run without errors; and secondly, to actually implement the transformation specified in CHART. The second especially is a major effort, analogous to the Verified Compiler research in, e.g., [10], and will be addressed in a separate follow-up project.
- On the pragmatic side, the RDT needs further experimentation with an eye towards efficiency. This is likely to lead to improvements in performance of the generated code.
- The CHART language can be extended with additional functionality, as well as syntactic sugar for the existing features.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010), http://www.eclipse.org/modeling/emft/henshin/
2. Artisan studio (2011), http://www.atego.com/
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007), http://www.key-project.org
4. Charter: Critical and hight assurance requirements transformed through engineering rigour (2010), http://charterproject.ning.com/page/charter-project
5. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)
6. de Mol, M., Rensink, A.: Formal semantics of the CHART transformation language. CTIT technical report, University of Twente (2011), http://www.cs.utwente.nl/~rensink/papers/chart.pdf
7. The FUJABA Toolsuite (2006), http://www.fujaba.de
8. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 183–198. Springer, Heidelberg (2010)
9. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
10. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
11. Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A systematic approach to metamodeling environments and model transformation systems in VMTS. Electr. Notes Theor. Comput. Sci. 127(1), 65–75 (2005), http://www.aut.bme.hu/Portal/Vmts.aspx?lang=en
12. Akehurst, D.H., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: SiTra: Simple Transformations in Java. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 351–364. Springer, Heidelberg (2006)
13. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2011), http://www.omg.org/spec/QVT/1.1/
14. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
15. Siebert, F.: Realtime garbage collection in the JamaicaVM 3.0. In: Bollella, G. (ed.) JTRES. ACM International Conference Proceeding Series, pp. 94–103. ACM (2007), http://www.aicas.com
16. Taentzer, G.: What algebraic graph transformations can do for model transformations. ECEASST 30 (2010)
17. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming 68(3), 187–207 (2007), http://www.eclipse.org/gmt/VIATRA2/

```
1  /**
2   * ================================================================
3   * CLASS generated for RULE 'examples.comic.generated.addPicture'.
4   * ================================================================
5   */
6  public class addPicture extends RDTRule.RDTRule1<Integer> {
7      ...
8      /** Finds a single match for the rule. */
9      private boolean match() throws GraphException {
10         // check (V1_comic.price > (1));
11         if (!(V1_comic.getPrice() > 1)) {
12             return false;
13         }
14         // Report success.
15         return true;
16     }
17
18     /** Applies the update block of the rule on an earlier found match. */
19     private void update() throws GraphException {
20         // Store for postponed graph updates.
21         final List<Closure> postponed = new ArrayList<Closure>(25);
22         // Comic V4_new_comic = new Comic();
23         final Comic V4_new_comic = Comic.createComic(this.context.getSupport());
24         // V4_new_comic.contains = V1_comic.contains + [V2_picture];
25         final GraphVisitor<Picture> t1 = new GraphVisitor<Picture>() {
26             @Override
27             public CONTINUE apply(final Picture node) throws GraphException {
28                 postponed.add(new Closure() {
29                     @Override
30                     public void apply() throws GraphException {
31                         V4_new_comic.insertPicture(-1, node);
32                     }
33                 });
34                 return CONTINUE.YES;
35             }
36         };
37         ...
38     }
39     ...
40 }
```

Part of the generated code for the addPicture rule.

Language Independent Refinement Using Partial Modeling

Rick Salay, Michalis Famelis, and Marsha Chechik

Department of Computer Science, University of Toronto, Toronto, Canada
{rsalay,famelis,chechik}@cs.toronto.edu

Abstract. Models express not only information about their intended domain but also about the way in which the model is incomplete, or "partial". This partiality supports the modeling process because it permits the expression of what is known without premature decisions about what is still unknown, until later refinements can fill in this information. A key observation of this paper is that a number of partiality types can be defined in a modeling language-independent way, and we propose a formal framework for doing so. In particular, we identify four types of partiality and show how to extend a modeling language to support their expression and refinement. This systematic approach provides a basis for reasoning as well as a framework for generic tooling support. We illustrate the framework by enhancing the UML class diagram and sequence diagram languages with partiality support and using Alloy to automate reasoning tasks.

1 Introduction

Models are used for expressing two different yet complementary kinds of information. The first is about the *intended domain* for the modeling language. For example, UML class diagrams are used to express information about system structure. The second kind of information is used to express the degree of incompleteness or *partiality* about the first kind. For example, class diagrams allow the type of an attribute to be omitted at an early modeling stage even though the type will ultimately be required for implementation. Being able to express partiality within a model is essential because it permits a modeler to specify the domain information she knows without prematurely committing to information she is still uncertain about, until later refinements can add it.

The motivating observation of this work is that *many types of model partiality are actually domain independent* and thus *support for expressing partiality can be handled in a generic and systematic way in any modeling language!* Furthermore, each type of partiality has its own usage scenarios and comes with its own brand of refinement. Thus, we can define certain model refinements formally yet independently of the language type and semantics. This may be one reason why many practitioners of modeling resist the formalization of the domain semantics for a model: it is possible to do some sound refinements without it!

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 224–239, 2012.

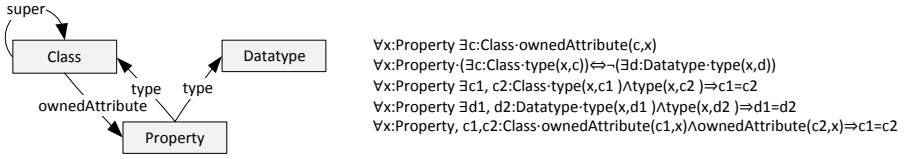


Fig. 1. A simplified UML class diagram metamodel

Current modeling languages incorporate partiality information in ad-hoc ways that do not clearly distinguish it from domain information and leave gaps in expressiveness. For example, with a state machine diagram, if the modeler uses multiple transitions on the same event out of a state, it may not be clear (e.g., to another modeler) whether she is specifying a non-deterministic state machine (domain information) or an under-specified deterministic state machine (partiality information). Benefits of explicating partiality in a language-independent manner include generic tool support for checking partiality-reducing refinements, avoiding gaps in expressiveness by providing complete coverage of partiality within a modeling language, and reusing a modeler's knowledge by applying partiality across different modeling languages consistently. Ad-hoc treatments of partiality do not allow the above benefits to be effectively realized. Our approach is to systematically add support for partiality information to any language in the form of annotations with well-defined formal semantics and a refinement relation for reducing partiality.

The use of partiality information has been studied for particular model types (e.g., behavioural models [9,13]) but our position paper [3] was the first to discuss language-independent partiality and its benefits for Model Driven Engineering. In this paper, we build on this work and provide a framework for defining different types of language-independent partiality. Specifically, this paper makes the following contributions: (1) we define the important (and novel) distinction between domain and partiality information in a modeling language; (2) we describe a formal framework for adding support for partiality and its refinement to any modeling language; (3) we use the framework to define four types of language-independent partiality that correspond to typical pragmatic modeling scenarios; (4) we implement the formalization for these using Alloy and show some preliminary results.

The rest of this paper is organized as follows: We begin with a brief introduction to the concept of partiality in Section 2 and give an informal description of four simple language-independent ways of adding partiality to a modeling language. We describe the composition of these partiality types and illustrate them through application to the UML class diagram and sequence diagram languages in Section 3. In Section 4, we describe a formalization of these types of partiality. In Section 5, we describe our tool support based on the use of Alloy [8]. After discussing related work in Section 6, we conclude the paper in Section 7 with the summary of the paper and suggestions for future work.

2 Adding Partiality to Modeling Languages

When a model contains partiality information, we call it a *partial* model. Semantically, it represents the set of different possible concrete (i.e., non-partial) models that would resolve the uncertainty represented by the partiality. In this paper, we focus on adding partiality information to existing modeling languages in a language-independent way, and thus, we must work with arbitrary metamodels. Figure 1 gives an example of a simple metamodel for class diagrams, with boxes for element types and arrows for relations. The well-formedness constraints (on the right) express the fact that every `Property` must have one `type` given by a `Class` or a `Datatype` and must be an `ownedAttribute` of one `Class`. Models consist of a set of *atoms* - i.e., the elements and relation instances of the types defined in its metamodel. In order to remain language-independent, we assume that partiality information is added as annotations to a model.

Definition 1 (Partial model). *A partial model P consists of a* base model*, denoted $bs(P)$, and a set of annotations. Let T be the metamodel of $bs(P)$. Then, $[P]$ denotes the set of T models called the* concretizations *of P.*

Partiality is used to express uncertainty about the model until it can be resolved using *partiality refinement*. Refining a partial model means removing partiality so that the set of concretizations shrinks until, ultimately, it represents a single concrete model. In general, when a partial model P' refines another one P, there is a mapping from $bs(P')$ to $bs(P)$ that expresses the relationship betwen them and thus between their concretizations. We give examples of such mappings later on in this section. In the special case that the base models are equivalent, P' refines P iff $[P'] \subseteq [P]$.

We now informally describe four possible partiality types, each adding support for a different type of uncertainty in a model: *May* partiality – about existence of its atoms; *Abs* partiality – about uniqueness of its atoms; *Var* partiality – about distinctness of its atoms; and *OW* partiality – about its completeness.

***May* Partiality.** Early in the development of a model, we may be unsure whether a particular atom should exist in the model and defer the decision until a later refinement. *May* partiality allows us to express the level of certainty we have about the presence of a particular atom in a model, by annotating it. The annotations come from the lattice $\mathcal{M} = \langle \{\text{E}, \text{M}\}, \preceq \rangle$, where the values correspond to "must exist" (E) and "may exist" (M), respectively, $\prec$ means "less certain than", and $\text{M} \prec \text{E}$.

A *May* model is refined by changing M atoms to E or eliminating them altogether. Thus, refinements result in submodels with more specific annotations. The *ground* refinements of a *May* model P are those that have no M elements and thus, correspond to its set of concretizations $[P]$. Figure 2(a) gives an example of a *May* model (`P`), a refinement (`P'`) and a concretization (`M`). The models are based on the metamodel in Figure 1. Atoms are given as *name:type* with the above annotations, and mappings between models are shown using dashed lines. Model (`P`) says "there is a class `Car` that may have a superclass `Vehicle`

and may have a `Length` attribute which may be of type `int`". The refinement (P′) and concretization (M) resolve the uncertainty.

***Abs* Partiality.** Early in the development of a model we may expect to have collections of atoms representing certain kinds of information but not know exactly what those atoms are. For example, in an early state machine diagram for a text editor, we may know that we have `InputingStates`, `ProcessingStates` and `FormattingStates`, and that `InputingStates` must transition to `ProcessingStates` and then to `FormattingStates`. Later, we refine these to sets of particular concrete states and transitions. *Abs* partiality allows a modeler to express this kind of uncertainty by letting her annotate atoms as representing a "particular", or unique, element (P) or a "set" (S). The annotations come from the lattice $\mathcal{A} = \langle \{\text{P}, \text{S}\}, \preceq \rangle$, where $\preceq$ indicates "less unique than", and $\text{S} \prec \text{P}$.

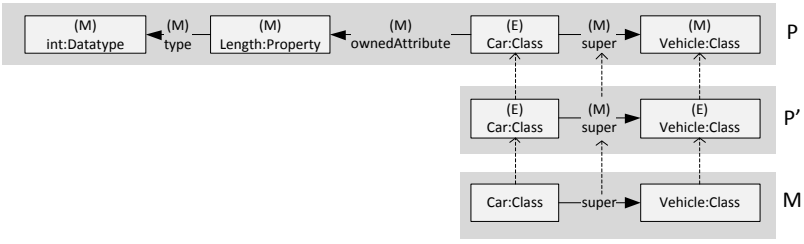
A refinement of an *Abs* model elaborates the content of "set" atoms S by replacing them with a set of S and P atoms. The ground refinements of an *Abs* model P are those that have no S elements and thus, correspond to its set of concretizations $[P]$. Figure 2(b) illustrates an *Abs* model, a refinement and concretization. Only node mappings are shown to reduce visual clutter. Model (P) says "class `Car` has a set `SizeRelated` of attributes with type `int`". The refinement (P′) refines `SizeRelated` into a particular attribute `Length` and the set `HeightRelated`.

***Var* Partiality.** Early in a modeling process, we may not be sure whether two atoms are distinct or should be the same, i.e., we may be uncertain about atom identity. For example, in constructing a class diagram, we may want to introduce an attribute that is needed, without knowing which class it will ultimately be in. To achieve well-formedness, it must be put into *some* class but we want to avoid prematurely putting it in the wrong class. To solve this problem, we could put it temporarily in a "variable" class - i.e., something that is treated like a class but, in a refinement, can be equated (merged) with other variable classes and eventually be assigned to a constant class. *Var* partiality allows a modeler to express uncertainty about distinctness of individual atoms in the model by annotating an atom to indicate whether it is a "constant" (C) or a "variable" (V). The annotations come from the lattice $\mathcal{V} = \langle \{\text{C}, \text{V}\}, \preceq \rangle$, where $\text{V} \prec \text{C}$.

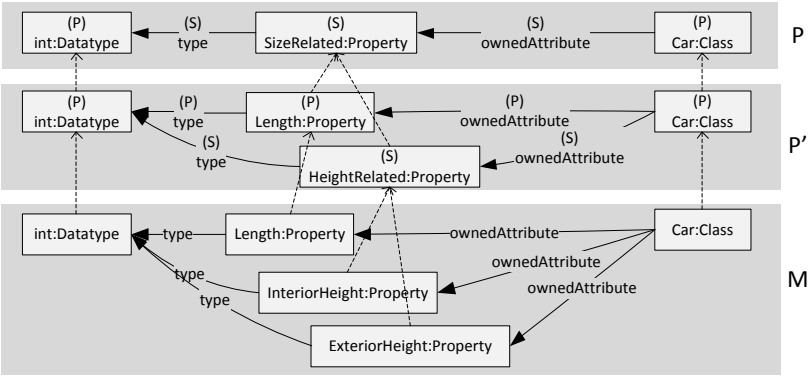
A refinement of a *Var* model involves reducing the set of variables by assigning them to constants or other variables. The ground refinements of a *Var* model P are those that have no V elements and thus, correspond to its set of concretizations $[P]$.

Figure 2(c) illustrates a *Var* model, its refinement and concretization. Model (P) says "class `Car` has superclass `Vehicle` and variable class `SomeVehicle` has attribute Length with variable type `SomeType`". Refinement (P′) resolves some uncertainty by assigning `SomeVehicle` to `Car`.

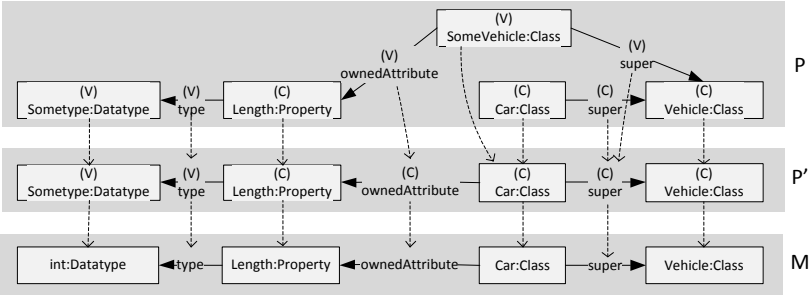
***OW* Partiality.** It is common, during model development, to make the assumption that the model is still incomplete, i.e., that other elements are yet to be added to it. This status typically changes to "complete" (if only temporarily) once some milestone, such as the release of software based on the model,



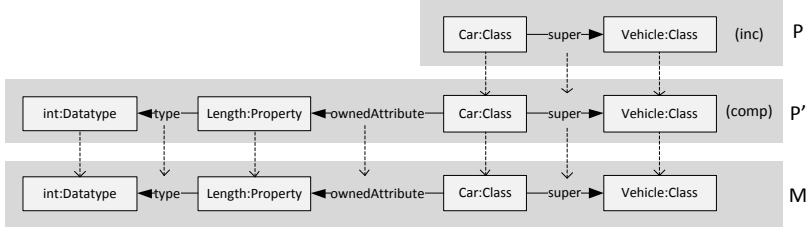
(a)



(b)



(c)



(d)

Fig. 2. Examples of different partiality types: (a) *May*; (b) *Abs*; (c) *Var*; (d) *OW*. In each example, model `M` concretizes both `P′` and `P`, and `P′` refines `P`.

is reached. In this paper, we view a model as a "database" consisting of a set of syntactic facts (e.g., "a class C_1 is a superclass of a class C_2", etc.). Thus, incompleteness corresponds to an Open World assumption on this database (the list of atoms is not closed), whereas completeness – to a Closed World. *OW* partiality allows a modeler to explicitly state whether her model is incomplete (i.e., can be extended) (INC) or not (COMP). In contrast to the other types of partiality discussed in this paper, here the annotation is at the level of the entire model rather than at the level of individual atoms. The annotations come from the lattice $\mathcal{O} = \langle \{\text{COMP}, \text{INC}\}, \preceq \rangle$, where INC $\prec$ COMP .

A refinement of an *OW* model means making it "more complete". The ground refinements of an *OW* model P, corresponding to its set of concretizations $[P]$, are its "complete" extensions. Figure 2(d) illustrates an *OW* model, refinement and concretization.

3 Combining and Applying Partiality Types

In this section, we show how to combine the four partiality types defined in Section 2 and then apply them to UML class diagrams and sequence diagrams, showing the language-independence of partiality-reducing refinements.

Combining Partiality Types. The four partiality types described above have distinctly different pragmatic uses for expressing partiality and can be combined within a single model to express more situations. We refer to the combination as the *MAVO* partiality, which allows model atoms to be annotated with *May*, *Abs* and *Var* partiality by using elements from the product lattice $\mathcal{M} \times \mathcal{A} \times \mathcal{V}$ defined as $\mathcal{MAV} = \langle \{\text{E}, \text{M}\} \times \{\text{P}, \text{S}\} \times \{\text{C}, \text{V}\}, \preceq \rangle$. For example, marking a class as $(\text{M}, \text{S}, \text{C})$ means that it represents a set of classes that may be empty, while marking it as $(\text{E}, \text{S}, \text{V})$ indicates that it is a non-empty set of classes but may become a different set of classes in a refinement. *OW* partiality is also used, but only at the model level, to indicate completeness.

MAVO refinement combines the refinement from the four types component-wise. If *MAVO* model P_1 is refined by model P_2, then there is a mapping from the atoms of P_1 to those of P_2, and the annotation in P_2 has a value that is no less than any of its corresponding atoms in P_1. Thus, the class marked $(\text{M}, \text{S}, \text{C})$ can be refined to a set of classes that have annotations such as $(\text{M}, \text{P}, \text{C})$ or $(\text{E}, \text{S}, \text{C})$ but not $(\text{M}, \text{S}, \text{V})$. Examples of applying the *MAVO* partiality are given below.

Application: *MAVO* Class Diagrams. One of the benefits of the fact that a partiality type extends the base language is that we can build on the existing concrete syntax of the languages. For example, consider the *MAVO* partial class diagram `P1` shown in the top of Figure 3. We do not show ground annotations (i.e., C for Var, P for Abs, etc.) and use the same symbols as in the abstract syntax for non-ground annotations. While there may be more intuitive ways to visualize some of these types of partiality (e.g., dashed outlines for "maybe" elements), we consider this issue to be beyond the scope of this paper.

In `P1`, the modeler uses *May* partiality to express uncertainty about whether a `TimeMachine` should be a `Vehicle` or not. *May* partiality is also used with

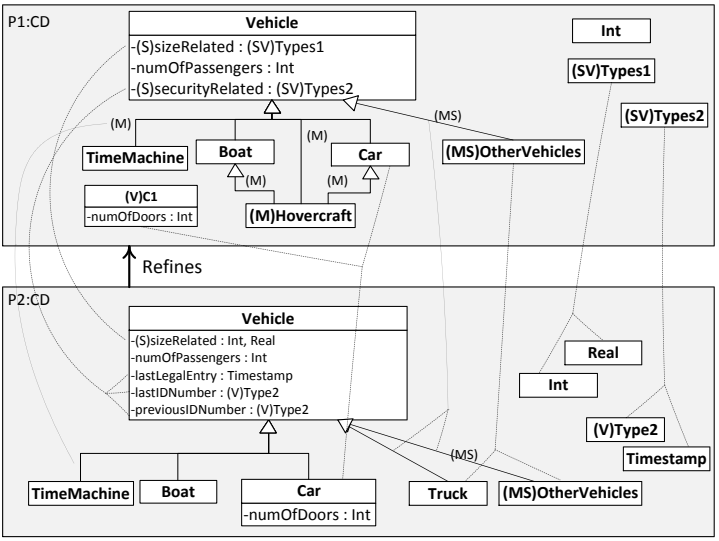


Fig. 3. Example of *MAVO* class diagrams with refinement

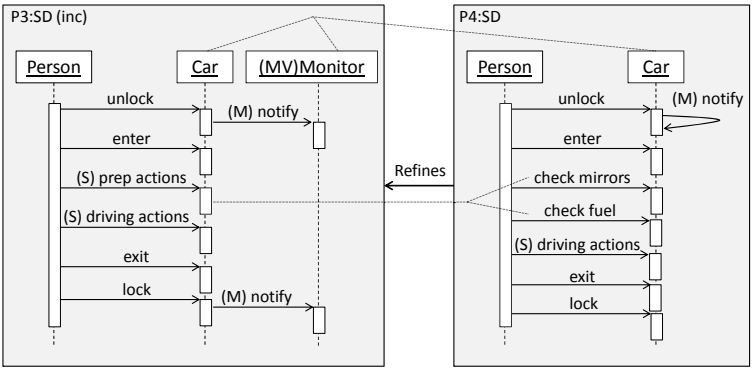


Fig. 4. Example sequence diagram with *MAVO* partiality

`Hovercraft` to express that the modeler is uncertain whether or not to include it and which class should be its superclass. *Var* partiality is used with "variable" class `C1` to introduce the attribute `numOfDoors : Integer` since the modeler is uncertain about which class it belongs in. *Abs* and *Var* partiality are used together to model sets of `Vehicle` attributes with unknown types with `sizeRelated : Types1` and `securityRelated : Types2`. Finally, *May* and *Abs* partiality are used with `OtherVehicles` and `super(OtherVehicles, Vehicle)` to indicate that the modeler thinks that there may be other, not yet known, vehicle classes.

Model P2, on the bottom of Figure 3, is a refinement of P1. Refinement mappings are shown as dashed lines and, to avoid visual clutter, we omit the identity mappings between ground atoms. In P2, the modeler refines super(TimeMachine, Vehicle) from "may exist" to "exists"; however, the decision on Hovercraft is to omit it. The refinement puts attribute numOfDoors : Integer into Car by setting C1 = Car. Also, the types of sizeRelated attributes are refined to Int or Real, and the securityRelated attributes are refined as well; however, the types of LastIDNumber and PreviousIDNumber are still unknown, although they are now known to be the same SType2. Finally, OtherVehicles is refined to expose Truck as one of these but still leaves the possibility for more Vehicle subclasses. The omitted *OW* annotation indicates that the models are "complete", and thus, new elements can only be added by refining an *Abs* set such as OtherVehicles.

Application: *MAVO* Sequence Diagrams. The left model in Figure 4, P3, shows a *MAVO* sequence diagram specifying how a Person interacts with a Car. We follow the same concrete syntactic conventions for annotations as for the class diagrams in Figure 4. While some interactions are known in P3, at this stage of the design process, it is known only that there will be a set of prepActions and drivingActions, and *Abs* partiality is used to express this. In addition, there is a possibility of there being a monitoring function for security. *May* partiality is used to indicate that this portion may be omitted in a refinement, and *Var* partiality is used to indicate that it is not yet clear which object will perform the Monitor role. Finally, P3 uses the *OW* partiality since we expect more objects to be added in a refinement.

In the model P4, on the right of Figure 4, the modeler has refined prepActions to a particular set of actions. In addition, she has assigned the Monitor role to Car itself (i.e., Monitor=Car) and retained only the first Notify message. Finally, she has decided that the model will not be extended further and it is set as "complete".

Discussion. While class diagrams and sequence diagrams are different syntactically and in their domains of applicability (i.e., structure vs. behaviour), the *MAVO* partiality provides the same capabilities for expressing and refining uncertainty in both languages. In particular, it adds the ability to treat atoms as removable (*May*), as sets (*Abs*), and as variables (*Var*), and to treat the entire model as extensible (*OW*). Furthermore, we were able to use the same concrete syntactic conventions in both languages — this is significant because modeler knowledge can be reused across languages. Note that while our examples come from UML, *MAVO* annotations are not UML-specific and can be applied to any metamodel-based language, regardless of the degree of formality of the language. The reason is that the semantics of partiality is expressed in terms of sets of models (i.e., possible concretizations) and does not depend on the native semantics of the underlying modeling language.

Most of the expressions of partiality in these examples required the added partiality mechanisms. The exceptions, which could have been expressed natively, are: (1) that types of attributes are unknown (as with the sizeRelated attributes), in class diagram P1, and (2) the choice between the Monitor and

its `Notify` messages (using an *Alt* operator, e.g., based on the STAIRS semantics [6]), in sequence diagram `P3`. This suggests that language-independent partiality types can add significant value to modeling languages.

4 Formalizing Partiality

In this section, we define an approach for formalizing the semantics of a partial model and apply it to *MAVO* partiality. Specifically, given a partial model P, we specify the set of concretizations $[P]$ using First Order Logic (FOL). Our approach has the following benefits: (1) it provides a general methodology for defining the semantics of a partial modeling language; (2) it provides a mechanism for defining refinement, even between partial models of different types; (3) it provides the basis for tool support for reasoning with partial models using off-the-shelf tools; and (4) it provides a sound way to compose partial modeling languages.

We begin by noting that a metamodel represents a set of models and can be expressed as an FOL theory.

Definition 2 (Metamodel). *A* metamodel *is a First Order Logic (FOL) theory $T = \langle \Sigma, \Phi \rangle$, where Σ is th e* signature *and Φ is a set of sentences representing the well-formedness constraints. $\Sigma = \langle \sigma, \rho \rangle$ consists of the set of sorts σ defining the element types and the set ρ of predicates defining the types of relations between elements. The models that conform to T are the finite FO Σ-structures that satisfy Φ according to the usual FO satisfaction relation. We denote the set of models with metamodel T by $Mod(T)$.*

The class diagram metamodel in Figure 1 fits this definition if we interpret boxes as sorts and edges as predicates.

Like a metamodel, a partial model also represents a set of models and thus can also be expressed as an FOL theory. Specifically, for a partial model P, we construct a theory $FO(P)$ s.t. $Mod(FO(P)) = [P]$. Furthermore, since P represents a subset of T models, we require that $Mod(FO(P)) \subseteq Mod(T)$. We guarantee this by defining $FO(P)$ to be an extension of T that adds constraints.

Let $M = bs(P)$ be the base model of a partial model P and let P_M be the *ground* partial model which has M as its base model and its sole concretization – i.e., $bs(P_M) = M$ and $[P_M] = \{M\}$. We first describe the construction of $FO(P_M)$ and then define $FO(P)$ in terms of $FO(P_M)$. To construct $FO(P_M)$, we extend T to include a unary predicate for each element in M and a binary predicate for each relation instance between elements in M. Then, we add constraints to ensure that the only first order structure that satisfies the resulting theory is M itself.

We illustrate the above construction using the class diagram `M` in Figure 2(a). Interpreting it as a partial model $P_{\mathtt{M}}$, we have:

$$FO(P_{\mathtt{M}}) = \langle \langle \sigma_{\mathtt{CD}}, \rho_{\mathtt{CD}} \cup \rho_{\mathtt{M}} \rangle, \Phi_{\mathtt{CD}} \cup \Phi_{\mathtt{M}} \rangle$$

(see Definition 2), where $\sigma_{\texttt{CD}}$, $\rho_{\texttt{CD}}$ and $\Phi_{\texttt{CD}}$ are the sorts, predicates and well-formedness constraints, respectively, for class diagrams, as described in Figure 1. $\rho_{\texttt{M}}$ and $\Phi_{\texttt{M}}$ are model M-specific predicates and constraints, defined in Figure 5. Since $FO(P_{\texttt{M}})$ extends CD, the FO structures that satisfy $FO(P_{\texttt{M}})$ are the class diagrams that satisfy the constraint set $\Phi_{\texttt{M}}$ in Figure 5. Assume N is such a class diagram. The constraint *Complete* ensures that N contains no more elements or relation instances than M. Now consider the class Car in M. *Exists* says that N contains at least one class called Car, *Unique* – that it contains no more than one class called Car, and *Distinct* – that the class called Car is different from the class called Vehicle. Similar sentences are given for class Vehicle and super instance CsuperV. The constraint *Type* ensures that CsuperV has correctly typed endpoints. These constraints ensure that $FO(P_{\texttt{M}})$ has exactly one concretization and thus $N = \texttt{M}$.

Relaxing the constraints $\Phi_{\texttt{M}}$ allows additional concretizations and represents a type of uncertainty. For example, if we are uncertain about whether M is complete, we can express this by removing the *Complete* clause from $\Phi_{\texttt{M}}$ and thereby allow concretizations to be class diagrams that extend M. Note that keeping or removing the *Complete* clause corresponds exactly to the semantics of the annotations COMP and INC in *OW* partiality, as defined in Section 2. Similarly, expressing each of *May*, *Abs* and *Var* partiality corresponds to relaxing $\Phi_{\texttt{M}}$ by removing *Exists*, *Unique* and *Distinct* clauses, respectively, for particular atoms. For example, removing the *Exists* clause $\exists x : \texttt{Class} \cdot \texttt{Car}(x)$ is equivalent to marking the class Car with M(i.e., Car may or may not exist), while removing the *Distinct* clause $\forall x : \texttt{Class} \cdot \texttt{Car}(x) \Rightarrow \neg\texttt{Vehicle}(x)$ is equivalent to marking the class *Car* with V(i.e., Car is a variable that can merge with Vehicle).

Figure 6 generalizes the construction in Figure 5 to an arbitrary ground theory $FO(P_M)$. ρ_M contains a unary predicate E for each element E in M and a binary predicate $\texttt{R}_{\texttt{ij}}$ for instance $R(E_i, E_j)$ of relation R in M. Each of the atom-specific clauses is indexed by an atom in model M to which it applies (e.g., $Exists_{\texttt{E}}$ applies to element E). For simplicity, we do not show the element types of the quantified variables.

We now formalize our earlier observation about relaxing Φ_M:

Observation 3 *Given a ground partial model P_M with $FO(P_M) = \langle\langle \sigma_T, \rho_T \cup \rho_M\rangle, \Phi_T \cup \Phi_M\rangle$ constructed as in Figure 5, any relaxation of the constraint Φ_M introduces additional concretizations into $Mod(FO(P_M))$ and represents a case of uncertainty about* M.

This observation gives us a general and sound approach for defining the semantics of a partial model. For partial model P with base model M, we define $FO(P)$ as $FO(P_M)$ with Φ_M replaced by a sentence Φ_P, where $\Phi_M \Rightarrow \Phi_P$.

Application to *MAVO*. Table 1 applies the general construction in Figure 6 to the individual *MAVO* partiality annotations by identifying which clauses to remove from Φ_M for each annotation. For example, the annotation (S)E corresponds to removing the clause $Unique_{\texttt{E}}$. *Note that nothing in the construction*

$\rho_{\mathtt{M}}$ contains the unary predicates $\mathtt{Car}(\mathtt{Class})$, $\mathtt{Vehicle}(\mathtt{Class})$ and the binary predicate $\mathtt{CsuperV}(\mathtt{Class}, \mathtt{Class})$.

$\Phi_{\mathtt{M}}$ contains the following sentences:

(Complete) $(\forall x : \mathtt{Class} \cdot \mathtt{Car}(x) \vee \mathtt{Vehicle}(x)) \wedge$
$(\forall x, y : \mathtt{Class} \cdot \mathtt{super}(x, y) \Rightarrow \mathtt{CsuperV}(x, y)) \wedge \neg\exists x \cdot \mathtt{Datatype}(x) \wedge \ldots$

`Car`:

(Exists) $\exists x : \mathtt{Class} \cdot \mathtt{Car}(x)$

(Unique) $\forall x, x' : \mathtt{Class} \cdot \mathtt{Car}(x) \wedge \mathtt{Car}(x') \Rightarrow x = x'$

(Distinct) $\forall x : \mathtt{Class} \cdot \mathtt{Car}(x) \Rightarrow \neg\mathtt{Vehicle}(x)$

similarly for `Vehicle`

`CsuperV`:

(Type) $\forall x, y : \mathtt{Class} \cdot \mathtt{CsuperV}(x, y) \Rightarrow \mathtt{Car}(x) \wedge \mathtt{Vehicle}(y)$

(Exists) $\forall x, y : \mathtt{Class} \cdot \mathtt{Car}(x) \wedge \mathtt{Vehicle}(y) \Rightarrow \mathtt{CsuperV}(x, y)$

(Unique) $\forall x, y, x', y' : \mathtt{Class} \cdot \mathtt{CsuperV}(x, y) \wedge \mathtt{CsuperV}(x', y') \Rightarrow x = x' \wedge y' = y$

Fig. 5. Example constraints for class diagram `M` in Figure 2(a)

of $FO(P_M)$ or in Table 1 is dependent on any specific features of the metamodel and hence the semantics of MAVO is language-independent.

The semantics for combined annotations is obtained by removing the clauses for each annotation – e.g., the annotation (SV)E removes the clause $Unique_{\mathsf{E}}$ and the clauses $Distinct_{\mathsf{EE}'}$ and $Distinct_{\mathsf{EE}'}$ for all elements E'.

The *MAVO* partiality types represent special cases of relaxing the ground sentence Φ_M by removing clauses but, as noted in Observation 3, any sentence weaker than Φ_M could be used to express partiality of Mas well. This suggests a natural way to enrich *MAVO* to express more complex types: augment the basic annotations with sentences that express additional constraints. We illustrate this using examples based on model `P1` in Figure 3. The statement "if `TimeMachine` is a `Vehicle`, then `Hovercraft` must be one as well" imposes a further constraint on the concretizations of `P1`. Using $FO(\mathtt{P1})$, we can express this in terms of the *Exists* constraints for individual atoms: $Exists_{\mathtt{TimeMachine}} \Rightarrow Exists_{\mathtt{Hovercraft}} \wedge Exists_{\mathtt{HsuperV}}$. Thus, propositional combinations of *Exists* sentences allow richer forms of the *May* partiality to be expressed.

Richer forms of the *Abs* partiality can be expressed by putting additional constraints on "S"-annotated atoms to further constrain the kinds of sets to which they can be concretized. For example, we can express the multiplicity

Table 1. Semantics of *MAVO* Partiality Annotations

MAVO annotation	Clause(s) to remove from Φ_M
INC	*Complete*
(M)E	$Exists_{\mathsf{E}}$
(S)E	$Unique_{\mathsf{E}}$
(V)E	$Distinct_{\mathsf{EE}'}$ and $Distinct_{\mathsf{E}'\mathsf{E}}$ for all E', $\mathsf{E}' \neq \mathsf{E}$
(M)R_{ij}	$Exists_{\mathsf{R}_{ij}}$
(S)R_{ij}	$Unique_{\mathsf{R}_{ij}}$
(V)R_{ij}	$Distinct_{\mathsf{R}_{ij}\mathsf{R}'_{kl}}$ and $Distinct_{\mathsf{R}'_{kl}\mathsf{R}_{ij}}$ for all R'_{kl}, $i \neq k, j \neq l$

Input: model M of type $T = \langle\langle \sigma_T, \rho_T\rangle, \Phi_T\rangle$
Output: $FO(P_M)$
$FO(P_M) = \langle\langle \sigma_T, \rho_T \cup \rho_M\rangle, \Phi_T \cup \Phi_M\rangle$
$\rho_M = \rho^e \cup \rho^r$, where $\rho^e = \{\mathsf{E}(\cdot) | E \text{ is an element of } M\}$
and $\rho^r = \{\mathsf{R_{ij}}(\cdot,\cdot) | \mathsf{R_{ij}} \text{ is an instance of relation } R \in \rho_T \text{ in } M\}$
Φ_M contains the following sentences:

(Complete) $(\forall x \cdot \bigvee_{\mathsf{E}\in\rho^e} \mathsf{E}(x)) \wedge (\bigwedge_{R\in\rho_T} \forall x, y \cdot R(x,y) \Rightarrow \bigvee_{\mathsf{R_{ij}}\in\rho^r} \mathsf{R_{ij}}(x,y))$

for each element E in M:

($Exists_{\mathsf{E}}$) $\exists x \cdot \mathsf{E}(x)$

($Unique_{\mathsf{E}}$) $\forall x, y \cdot \mathsf{E}(x) \wedge \mathsf{E}(y) \Rightarrow x = y$

$\bigwedge_{\mathsf{E}'\in\rho^e, \mathsf{E}'\neq\mathsf{E}}$ *($Distinct_{\mathsf{EE}'}$)* $\forall x \cdot \mathsf{E}(x) \Rightarrow \neg\mathsf{E}'(x)$

for each relation instance $\mathsf{R_{ij}}$ in M:

($Type_{\mathsf{R_{ij}}}$) $\forall x, y \cdot \mathsf{R_{ij}}(x,y) \Rightarrow \mathsf{E_i}(x) \wedge \mathsf{E_j}(y)$

($Exists_{\mathsf{R_{ij}}}$) $\forall x, y \cdot \mathsf{E_i}(x) \wedge \mathsf{E_j}(y) \Rightarrow \mathsf{R_{ij}}(x,y)$

($Unique_{\mathsf{R_{ij}}}$) $\forall x, y, x', y' \cdot \mathsf{R_{ij}}(x,y) \wedge \mathsf{R_{ij}}(x',y') \Rightarrow x = x' \wedge y = y'$

$\bigwedge_{\mathsf{R'_{kl}}\in\rho^r, i\neq k, j\neq l}$ *($Distinct_{\mathsf{R_{ij}R'_{kl}}}$)* $\forall x, y \cdot \mathsf{R_{ij}}(x,y) \Rightarrow \neg\mathsf{R'_{kl}}(x,y)$

Fig. 6. Construction of $FO(P_M)$

constraint that there can be at most two `sizeRelated` attributes by replacing the constraint $Unique_{\texttt{sizeRelated}}$ with the following weaker one:

$$\forall x, x', x'' \cdot \texttt{sizeRelated}(x) \wedge \texttt{sizeRelated}(x') \wedge \texttt{sizeRelated}(x'') \Rightarrow (x = x' \vee x = x'' \vee x' = x'')$$

Of course, this can be easily expressed in a language with sets and counting, like OCL. Similar enrichments of the *Var* and the *OW* partialities can be produced by an appropriate relaxation of the *Distinct* and *Complete* constraints, respectively. These enrichments of *MAVO* remain language-independent because they do not make reference to the metamodel-specific features.

Refinement of *MAVO* Partiality. We have defined partial model semantics in terms of relaxations to Φ_M. Below, we define refinement in terms of these as well. Specifically, assume we have relaxations $\Phi_{P'}$ and Φ_P for partial models P' and P, respectively. In the special case that their base models are equivalent, we have P' refines P iff $[P'] \subseteq [P]$ and this holds iff $\Phi_{P'} \Rightarrow \Phi_P$. However, when the base models are different, the sentences are incomparable because they are based on different signatures. The classic solution to this kind of problem (e.g., in algebraic specification) is to first translate them into the same signature and then check whether the implication holds in this common language (e.g., see [5]). In our case, we can use a refinement mapping R between the base models, such as the one in Figures 3 and 4, to define a function that translates Φ_P to a semantically equivalent sentence $\mathsf{R}(\Phi_P)$ over the signature $\Sigma_{P'}$. Then, P' refines P iff $\Phi_{P'} \Rightarrow \mathsf{R}(\Phi_P)$. We omit the details of this construction due to space limitations; however, interested readers can look at the Alloy model for Experiment 6 in Section 5 for an example of this construction.

Table 2. Results of experiments using Alloy

Exp. #	Question	Answer	Scope	Time (ms)
1	Does the ground case for P1 have a single instance?	**Yes**	7	453
2	Does the ground case for P2 have a single instance?	**Yes**	6	366
3	Is P1 extended with Q1 consistent?	**Yes**	4	63
4	Is P1 extended with Q1 and Q2 consistent?	**No**	20	64677
5a	Is P1 extended with Q1 and Q3 consistent?	**Yes**	4	64
5b	Is P1 extended with Q1 and ¬Q3 consistent?	**Yes**	5	151
6	Is P2 a correct refinement of P1?	**Yes**	10	9158

5 Tool Support and Preliminary Evaluation

In order to show the feasibility of using the formalization in Section 4 for automated reasoning, we developed an Alloy [8] implementation for *MAVO* partiality. We used a Python script to generate the Alloy encoding of the clauses (as defined in Figure 6) for the models P1 and P2, shown in Figure 3. The Alloy models are available online at `http://www.cs.toronto.edu/se-research/fase12.htm`. We then used this encoding for *property checking*. More specifically, we attempted to address questions such as "does any concretization of P have the property Q?" and "do all concretizations of P have the property Q?", where Q is expressed in FOL. The answer to the former is affirmative iff $\Phi_P \wedge Q$ is satisfiable, and to the latter iff $\Phi_P \wedge \neg Q$ is not satisfiable. We also used the tooling to check correctness of *refinement*, cast as a special case of property checking. As discussed in Section 4, P' refines P iff $\Phi_{P'} \Rightarrow \mathsf{R}(\Phi_P)$ where R translates Φ_P according to the refinement mapping. Thus, the refinement is correct iff $\Phi_{P'} \wedge \neg \mathsf{R}(\Phi_P)$ is not satisfiable.

Table 2 lists the experiments we performed, using the following properties:

Q1 : `Vehicle` has at most two direct subclasses.
Q2 : Every class, except C1 is a direct subclass of C1.
Q3 : There is no multiple inheritance.

Experiments (1) and (2) verify our assumption that the encoding described in Figure 6 admits only a single concretization. Although any pure MAVO model is consistent by construction, Experiments (3) and (4) illustrate that this is not necessarily the case when additional constraints are added. First, P1 is extended with Q1 and shown to be consistent. However, extending P1 with both Q1 and Q2 leads to an inconsistency. This happens because Q2 forces (a) C1 to be merged with `Vehicle`, and (b) `TimeMachine` to be its subclass, raising its number of direct subclasses to 3. This contradicts Q1, and therefore, P1 ∧ Q1 ∧ Q2 is inconsistent. Note that Experiment (4) takes longer than the others because showing inconsistency requires that the SAT solver enumerate all possible models within the scope bounds. In Experiment (5), we asked whether the version of P1 extended with Q1 satisfies property Q3 and found that this is the case in some

(Experiment 5a) but not all (Experiment 5b) concretizations. Finally, in Experiment (6) we verified the refinement described in Figure 3, using the mapping in the figure to construct a translation of $\Phi_{\mathtt{P1}}$, as discussed in Section 4.

Our experiments have validated the feasibility of using our formalization for reasoning tasks. In our earlier work [4], we have done a scalability study for property checking using a SAT solver for *May* partiality (with propositional extensions). The study showed that, compared to explicitly handling the set of concretizations, our approach offers significant speedups for large sets of concretizations. We intend to do similar scalability studies for all *MAVO* partialities in the future.

6 Related Work

In this section, we briefly discuss other work related to the types of partiality introduced in this paper.

A number of partial *behavioural* modeling formalisms have been studied in the context of abstraction (e.g., for verification purposes) or for capturing early design models [12]. The goal of the former is to represent property-preserving abstractions of underlying concrete models, to facilitate model-checking. For example, Modal Transition Systems (MTSs) [9] allow introduction of uncertainty about transitions on a given event, whereas Disjunctive Modal Transition Systems (DMTSs) [10] add a constraint that at least one of the possible transitions must be taken in the refinement. Concretizations of these models are Labelled Transition Systems (LTSs). MTSs and DMTSs are results of a limited application of `May` partiality. Yet, the MTS and DMTS refinement mechanism allows resulting LTS models to have an arbitrary number of states which is different from the treatment provided in this paper, where we concentrated only on "structural" partiality and thus state duplication was not applicable.

In another direction, Herrmann [7] studied the value of being able to express *vagueness* within design models. His modeling language SeeMe has notational mechanisms similar to *OW* and *May* partiality; however, there is no formal foundation for these mechanisms.

Since models are like databases capturing facts about the models' domain, work on representing incomplete databases is relevant. *Var* partiality is traditionally expressed in databases by using null values to represent missing information. In fact, our ideas in this area are inspired by the work on data exchange between databases (e.g., [2]) which explicitly uses the terminology of "variables" for nulls and "constants" for known values. An approach to the *OW* partiality is the use of the Local Closed World Assumption [1] to formally express the places where a database is complete.

Finally, our heavy reliance on the use of FOL as the means to formalize metamodels and partial models gives our work a strong algebraic specification flavor and we benefit from this connection. In particular, partial model refinement is a kind of specification refinement [11]. Although our application is different – dealing with syntactical uncertainty in models rather than program semantics – we hope to exploit this connection further in the future.

7 Conclusion and Future Work

The key observation of our work is that many types of partiality information and their corresponding types of refinement are actually language-independent and thus can be added to any modeling language in a uniform way. In this paper, we defined a formal approach for doing so in any metamodel-based language by using model annotations with well-defined semantics. This allows us to incorporate partiality across different languages in a consistent and complete way, as well as to develop language-independent tools for expressing, reasoning with, and refining partiality within a model. We then used this approach to define four types of partiality, each addressing a distinctly different pragmatic situation in which uncertainty needs to be expressed within a model. We combined all four and illustrated their language independence by showing how they can be applied to class diagrams and to sequence diagrams. Finally, we demonstrated the feasibility of tool support for our partiality extensions by describing an Alloy-based implementation of our formalism and various reasoning tasks using it.

The investigation in this paper suggests several interesting directions for further research. First, since adding support for partiality lifts modeling languages to partial modeling languages, it is natural to consider whether a similar approach could be used to lift *model transformations* to *partial model transformations*. This would allow partiality to propagate through a transformation chain during model-driven development and provide a principled way of applying transformations to models earlier in the development process, when they are incomplete or partial in other ways. Second, it would be natural to want to interleave the partiality-reducing refinements we discussed in this paper with other, language-specific, refinement mechanisms during a development process. We need to investigate how these two types of refinements interact and how they can be soundly combined. Third, since modelers often have uncertainty about entire model fragments, it is natural to ask how to extend *MAVO* annotation to this case. Applying *May* partiality to express a design alternative is straightforward – a fragment with annotation M may or may not be present; however, the use of the other *MAVO* types is less obvious and deserves further exploration. Finally, although we have suggested scenarios in which particular *MAVO* annotations would be useful, we recognize that the methodological principles for applying (and refining) partial models require a more thorough treatment. We are currently developing such a methodology.

References

1. Cortés-Calabuig, A., Denecker, M., Arieli, O.: On the Local Closed-World Assumption of Data-Sources. J. Logic Programming (2005)
2. Fagin, R., Kolaitis, P., Miller, R., Popa, L.: Data Exchange: Semantics and Query Answering. Theoretical Computer Science 336(1), 89–124 (2005)
3. Famelis, M., Ben-David, S., Chechik, M., Salay, R.: Partial Models: A Position Paper. In: Proc. of MoDeVVa 2011, pp. 1–6 (2011)

4. Famelis, M., Salay, R., Chechik, M.: Partial Models: Towards Modeling and Reasoning with Uncertainty (2011) (submitted)
5. Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. Journal of the ACM (JACM) 39(1), 95–146 (1992)
6. Haugen, O., Husa, K.E., Runde, R.K., Stolen, K.: STAIRS: Towards Formal Design with Sequence Diagrams. SoSyM 4(4), 355–357 (2005)
7. Herrmann, T.: Systems Design with the Socio-Technical Walkthrough. In: Hndbk of Research on Socio-Technical Design and Social Networking Systems, pp. 336–351 (2009)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
9. Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: Proc. of LICS 1988, pp. 203–210 (1988)
10. Larsen, P.: The Expressive Power of Implicit Specifications. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 204–216. Springer, Heidelberg (1991)
11. Sannella, D., Tarlecki, A.: Essential Concepts of Algebraic Specification and Program Development. Formal Aspects of Computing 9(3), 229–269 (1997)
12. Uchitel, S., Chechik, M.: Merging Partial Behavioural Models. In: FSE 2004, pp. 43–52 (2004)
13. Wei, O., Gurfinkel, A., Chechik, M.: On the Consistency, Expressiveness, and Precision of Partial Modeling Formalisms. J. Inf. Comput. 209(1), 20–47 (2011)

A Conceptual Framework for Adaptation*

Roberto Bruni[1], Andrea Corradini[1], Fabio Gadducci[1],
Alberto Lluch Lafuente[2], and Andrea Vandin[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] IMT Institute for Advanced Studies Lucca, Italy

Abstract. In this position paper we present a conceptual vision of adaptation, a key feature of autonomic systems. We put some stress on the role of control data and argue how some of the programming paradigms and models used for adaptive systems match with our conceptual framework.

Keywords: Adaptivity, autonomic systems, control data, MAPE-K control loop.

1 Introduction

Self-adaptive systems have been widely studied in several disciplines ranging from Biology to Economy and Sociology. They have become a hot topic in Computer Science in the last decade as a convenient solution to the problem of mastering the complexity of modern software systems, networks and architectures.

According to a widely accepted *black-box* or *behavioural* definition, a software system is called "self-adaptive" if it can modify its behaviour as a reaction to a change in its context of execution, understood in the widest possible way, including both the external environment and the internal state of the system itself. Typically the changes of behaviour are aimed at improving the degree of satisfaction of some either functional or non-functional requirements of the system, and self-adaptivity is considered a fundamental feature of *autonomic systems*, that can specialize to several other self-* properties (see e.g. [9]).

An interesting taxonomy is presented in [14], where the authors stress the highly interdisciplinary nature of the studies of such systems. Indeed, just restricting to the realm of Computer Science, active research on self-adaptive systems is carried out in Software Engineering, Artificial Intelligence, Control Theory, and Network and Distributed Computing, among others. However, as discussed in §3, only a few contributions address the foundational aspects of such systems, including their semantics and the use of formal methods for analysing them.

In this paper we propose an answer to very basic questions like **"when is a software system adaptive?"** or **"how can we identify the adaptation logic in an adaptive system?"**. We think that the limited effort placed in the investigation of the foundations of (self-)adaptive software systems might be due to the fact that it is not clear what are the characterizing features that distinguish

* Research supported by the European Integrated Project 257414 ASCENS.

J, de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 240–254, 2012.

such systems from plain ("non-adaptive") ones. In fact, almost any software system can be considered self-adaptive, according to the black-box definition recalled above, since any system of a reasonable size can *modify its behaviour* (e.g., by executing different conditional branches) as a *reaction to a change in the context of execution* (like the change of variables or parameters).

These considerations show that the above behavioural definition of adaptivity is not useful in pinpointing adaptive systems, even if it allows to discard many systems that certainly are not. We should rather take a *white-box* perspective which allows us to inspect, to some extent, the internal structure of a system: we aim to have a clear *separation of concerns* to distinguish the cases where the changes of behaviour are part of the application logic from those where they realize the adaptation logic, calling adaptive only systems capable of the latter.

Self-adaptivity is often obtained by enriching the software that implements the standard application logic with a control loop that monitors the context of execution, determines the changes to be enforced, and enacts them. Systems featuring such an architectural pattern, often called MAPE-K [8,9,10], should definitely be considered as adaptive. But as argued in [4] there are other, simpler adaptive patterns, like the Internal Feedback Loop pattern, where the control loop is not as neatly separated from the application logic as in MAPE-K, and the Reactive Adaptation pattern, where the system just reacts to events from the environment by changing its behaviour. Also systems realizing such patterns should be captured by a convincing definition of adaptivity, and their adaptation logic should be exposed and differentiated from their application logic.

Other software systems that can easily be categorized as (self-)adaptive are those implemented with programming languages explicitely designed to express adaptation features. Archetypal examples are languages belonging to the paradigm of Context Oriented Programming, where the contexts of execution are first-class citizens [15], or to that of Dynamic Aspect Oriented Programming. Nevertheless, it is not the programming language what makes a program adaptive or not: truly adaptive systems can be programmed in traditional languages, exactly like object-oriented systems can, with some effort, in traditional imperative languages.

The goal of this position paper is to present a conceptual framework for adaptation, proposing a simple structural criterion to portray adaptivity (§2). We discuss how systems developed according to mainstream methodologies are shown to be adaptive according to our definition (§3), and explain how to understand adaptivity in many computational formalisms (§4). We sketch a first formalization of our concepts (§5). Finally, we discuss future developments of these ideas (§6).

2 When is a Software Component Adaptive?

Software systems are made of one or more processes, roughly programs in execution, possibly interacting among themselves and with the environment in arbitrarily complex ways. Sometimes an adaptive behaviour of such a complex system may emerge from the interactions among its components, even if the components in isolation are not adaptive. However, we do not discuss this kind

of adaptivity here: we focus instead on the adaptivity of simple components, for which we introduce the following conceptual framework.

According to a traditional paradigm, a program governing the behaviour of a component is made of *control* and *data*: these are two conceptual ingredients that in presence of sufficient resources (like computing power, memory or sensors) determine the behaviour of the component. In order to introduce adaptivity in this framework, we require to make explicit the fact that the behaviour of a component depends on some well identified *control data*. At this level of abstraction we are not concerned with the way the behaviour of the component is influenced by the control data, nor with the structure of such data.

Now, **we define *adaptation* as the run-time modification of the control data**. From this basic definition we derive several others. **A component is *adaptable* if it has a distinguished collection of control data that can be modified at run-time**. Thus if either the control data are not identified or they cannot be modified, then the system is not adaptable. Further, **a component is *adaptive* if it is adaptable and its control data are modified at run-time**, at least in some of its executions. And **a component is *self-adaptive* if it is able to modify its own control data at run-time**.

Given the intrinsic complexity of adaptive systems, this conceptual view of adaptation might look like an oversimplification. Our goal is to show that instead it enjoys most of the properties that one would require from such a definition.

Any definition of adaptivity should face the problem that the judgement whether a system is adaptive or not is often subjective. Indeed, one can always argue that whatever change in the behaviour the system is able to manifest is part of the application logic, and thus should not be deemed as "adaptation". From our perspective, this is captured by the fact that the collection of control data of a component can be defined, at least in principle, in an arbitrary way, ranging from the empty set ("the system is not adaptable") to the collection of all the data of the program ("any data modification is an adaptation").

As a concrete example, we may ask ourselves whether the execution of a simple branching statement, like **if** `tooHeavy` **then** `askForHelp` **else** `push` can be interpreted as a form of adaptation. The answer is: it depends.

Suppose that the statement is part of the software controlling a robot, and that the boolean variable `tooHeavy` is set according to the value returned by a sensor. If `tooHeavy` is considered as a *standard program variable*, then the change of behaviour caused by a change of its value is not considered "adaptation". If `tooHeavy` is instead considered as control data, then its change triggers an adaptation. Summing up, our question can be answered only after a clear identification of the control data.

Ideally, a sensible collection of control data should be chosen to enforce a separation of concerns, allowing to distinguish neatly, if possible, the activities relevant for adaptation (those that affect the control data) from those relevant for the application logic only (that should not modify the control data). We will come back to this methodological point along §3 and §4.

Of course, any computational model or programming language can be used to implement an adaptive system, just by identifying the part of the data that governs the behaviour. Consequently, the nature of control data can vary considerably, ranging from simple configuration parameters to a complete representation of the program in execution that can be modified at run-time. This latter scenario is typical of computational models that support meta-programming or reflective features even if, at least in principle, it is possible for any Turing-complete formalism. We shall discuss in §4 how adaptivity, as defined above, can be obtained in systems implemented according to several computational formalisms. Before that, as a *proof of concept*, we discuss in the next section how several well accepted architectures of adaptive systems can be cast in our framework.

3 Architectures, Patterns and Reference Models for Adaptivity

Several contributions to the literature describe possible architectures or reference models for adaptive systems (or for *autonomic systems*, for which self-adaptivity is one of the main features). In this section we survey some of such proposals, stressing for each of them how a reasonable notion of control data can be identified. According to the MAPE-K architecture, a widely accepted reference model introduced in a seminal IBM paper [8], a self-adaptive system is made of a component implementing the application logic, equipped with a control loop that monitors the execution through suitable sensors, analyses the collected data, plans an adaptation strategy, and finally executes the adaptation of the managed component through some effectors; all the phases of the control loop access a shared knowledge repository. Adaptation according to this model naturally fits in our framework with an obvious choice for the control data: these are the data of the managed component which are either sensed by the monitor or modified by the execute phase of the control loop. Thus the control data represent the interface exposed by the managed component through which the control loop can operate, as shown in Fig. 1. The managed component is adaptable, and the system made of both the component and the control loop is self-adaptive.

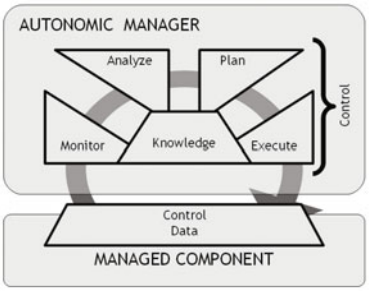


Fig. 1. Control data in MAPE-K

The construction can be iterated, as the control loop itself could be adaptable. As an example think of a component which follows a plan to perform some tasks. It can be adaptable, having a manager which devises new plans according to changes in the context or in the component's goals. In turn, this planning component might itself be adaptable, with another component that controls and adapts its planning strategy, for instance determining the new strategy on the basis of a tradeoff between optimality of the plans and computational cost. In this case, the planning component (that realizes the control loop of the base component)

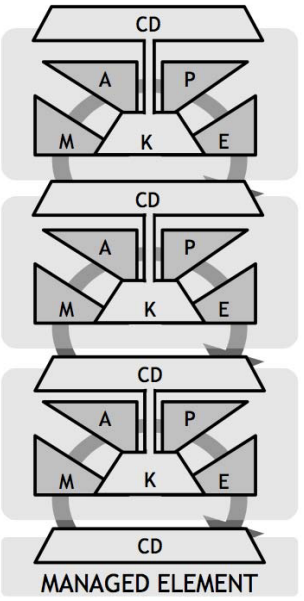


Fig. 2. Tower of adaptation

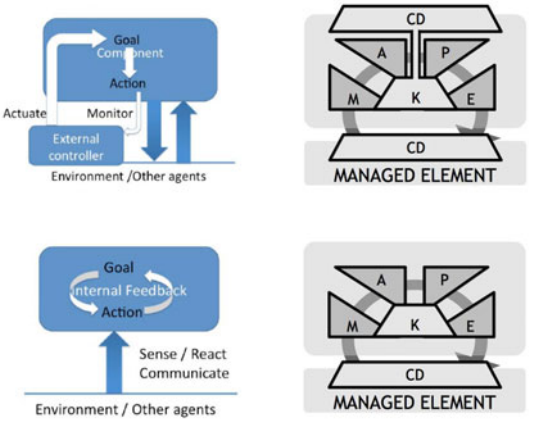


Fig. 3. External (top) and internal (bottom) patterns

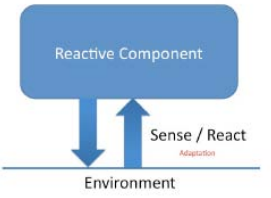


Fig. 4. Reactive pattern

exposes itself some control data (conceptually part of its knowledge), thus enabling a hierarchical composition that allows one to build towers of adaptive components (Fig. 2).

Another general reference model has been proposed in [1], where *(computational) reflection is promoted as a necessary criterion for any self-adaptive software system.* Reflection implies the presence, besides of base-level components and computations, also of meta-level subsystems and meta-computations that act on a meta-model. Meta-computations can inspect and modify the meta-model that is causally connected to the base-level system, so that changes in one are reflected in the other. The authors argue that most methodologies and frameworks proposed for the design and development of self-adaptive systems rely on some form of reflection, even if this is not always made explicit. Building on these considerations, in [18] they introduce FORMS, a formal reference model that provides basic modeling primitives and relationships among them, suitable for the design of self-adaptive systems. Such primitives allow one to make explicit the presence of reflective (meta-level) subsystems, computations and models.

The goals of [1] are not dissimilar from ours, as they try to capture the essence of *self-adaptive* systems, identifying it in computational reflection; recall anyway that with our notion of control data we aimed at capturing the essence of the sole *adaptability*. We argue that in self-adaptive systems conforming to this model it should be relatively easy to identify the relevant control data. It is pretty clear

that in reflective systems containing an explicit meta-model of the base-level system (like those conforming to the architecture-based solution proposed in [12]), such meta-model plays exactly the role of control data. Nevertheless, the FORMS modeling primitives can be instantiated and composed in a variety of ways (one for modeling MAPE-K and one for a specific application are discussed in [18]); in general in any such reflective system the control data could be identified at the boundaries between the meta-level and the base-level components.

In other frameworks for the design of adaptive systems (like [19]) the base-level system has a fixed collection of possible behaviours (or behavioural models), and adaptation consists of passing from one behaviour to another one, for example for the sake of better performance, or to ensure, in case of partial failure, the contractually agreed functionalities, even if in a degraded form. The approach proposed in [19] emphasizes the use of formal methods to validate the development of adaptive systems, for example by requiring the definition of global invariants for the whole system and of local requirements for the "local" behaviours. Specifically, it represents the local behavioural models with coloured Petri nets, and the adaptation change from one local model to another with an additional Petri net transition (labeled `adapt`). Such `adapt` transitions describe how to transform a state (a set of tokens) in the source Petri net into a state in the target model, thus providing a clean solution to the *state transfer problem* common to these approaches. In this context, a good choice of control data would be the Petri net that describes the current base-level computation, which is replaced during an adaptation change by another local model. Instead, the alternative and pretty natural choice of control data as the tokens that are consumed by the `adapt` transition would be considered poor, as it would not separate clearly the base-level from the meta-level computations.

In the architectural approach of [2], a system specification has a two-layered architecture to enforce a separation between computation and coordination. The first layer includes the basic computational components with well-identified interfaces, while the second one is made of connectors (called *coordination contracts*) that link the components appropriately in order to ensure the required system's functionalities. Adaptation in this context is obtained by *reconfiguration*, which can consist of removal/addition/replacement of both base components and connectors among them. The possible reconfigurations of a system are described declaratively with suitable rules, grouped in *coordination contexts*: such rules can be either invoked explicitly, or triggered automatically by the verification of certain conditions. In this approach, as adaptation is identified with reconfiguration, the control data consist of the whole two-layered architecture, excluding the internal state of the computational components.

More recently, a preliminary taxonomy of adaptive patterns has been proposed [4]. Two of these capture typical control loop patterns such as the *internal* and the *external* ones. Like MAPE-K, also these patterns can be cast easily in our framework (see Fig. 3): in the internal control loop pattern, the manager is a wrapper for the managed component and it is not adaptable, while in the external control loop pattern the manager is an adaptable component that is

connected with the managed component. The third adaptive pattern describes *reactive* components (see Fig. 4). Such components are capable to modify their behavior in reaction to an external event, without any control loop. In our conceptual framework, a reactive system of this kind is (self-)adaptive if we consider as control data the variables that are modified by sensing the environment.

Let us conclude by considering two of the few contributions that propose a formal semantics for adaptive systems. In [13] the author identifies suitable semantical domains aimed at capturing the essence of adaptation. The behaviour of a system is formalized in terms of a category of *specification carrying programs* (also called *contracts*), i.e. triples made of a program, a specification and a satisfaction relation among them; arrows between contracts are refinement relations. Contracts are equipped with a functorial semantics, and their adaptive version is obtained by indexing the semantics with respect to a set of *stages of adaptation*, yielding a coalgebric presentation potentially useful for further generalizations. At present it is not yet clear whether a notion of control data could fit in this abstract semantical framework or not: this is a topic of current investigation.

Finally, [3] proposes a formal definition of when a system exposes an *adaptive behaviour* with respect to a user. A system is modeled as a black-box component that can interact with the user and with the environment through streams of data. A system is assumed to be deterministic, thus if it reacts non-deterministically to the input stream provided by the user, this is interpreted as an evidence of the fact that the system adapted its behaviour after an interaction with the environment. Different kinds of adaptation are considered, depending on how much of the interaction between the environment and the system can be observed by the user. Even if formally crisp, this definition of adaptivity is based on strong assumptions (e.g. systems are deterministic, all adaptive systems are interactive) that can restrict considerably its range of applicability. For example, it would not classify as adaptive a system where a change of behaviour is triggered by an interaction with the user.

4 Adaptivity in Various Computational Paradigms

As observed in §2 and §3, the nature of control data can vary considerably depending both on the degree of adaptivity of the system and on the nature of the computational formalisms used to implement it. Examples of control data include configuration variables, rules (in rule-based programming), contexts (in context-oriented programming), interactions (in connector-centered approaches), policies (in policy-driven languages), aspects (in aspect-oriented languages), monads and effects (in functional languages), and even entire programs (in models of computation exhibiting higher-order or reflective features).

We outline some rules of thumb for the choice of control data within a few computational formalisms that are suited for implementing adaptive systems.

Context-Oriented Programming. Many programming languages have been promoted as suitable for programming adaptive systems [7]. A recent example is context-oriented programing which has been designed as a convenient

paradigm for programming autonomic systems in general [15]. The main idea of this paradigm is that the execution of a program depends on the run-time environment under which the program is running.

Many languages have been extended to adopt the context-oriented paradigm. We mention among others Lisp, Python, Ruby, Smalltalk, Scheme, Java, and Erlang. The notion of context varies from approach to approach and in general it might refer to any computationally accessible information. A typical example is the environmental data collected from sensors. In many cases the universe of all possible contexts is discretised in order to have a manageable, abstract set of fixed contexts. This is achieved, for instance, by means of functions mapping the environmental data into the set of fixed contexts. Code fragments like methods or functions can then be specialized for each possible context. Such chunks of behaviours associated with contexts are called *variations*.

The context-oriented paradigm can be used to program autonomic systems by activating or deactivating variations in reaction to context changes. The key mechanism exploited here is the dynamic dispatching of variations. When a piece of code is being executed, a sort of dispatcher examines the current context of the execution in order to decide which variation to invoke. Contexts thus act as some sort of possibly nested scopes. Indeed, very often a stack is used to store the currently active contexts, and a variation can propagate the invocation to the variation of the enclosing context.

The key idea to achieve adaptation along the lines of the MAPE-K framework is for the manager to control the context stack (for example, to modify it in correspondence with environmental changes) and for the managed component to access it in a read-only manner. Those points of the code in which the managed component queries the current context stack are called *activation hooks*.

Quite naturally, context-oriented programming falls into our framework by considering the context stack as *control data*. With this view, the only difference between the approach proposed in [15] and our ideas is that the former suggests the control data to reside within the manager, while we locate the control data in the interface of the managed component.

Declarative Programming. Logic programming and its variations are one of the most successful declarative programming paradigms. In the simplest variant, a logic program consists of a set of Horn clauses and, given a goal, a computation proceeds by applying repeatedly SLD-resolution trying to reach the empty clause in order to refuse the initial goal.

Most often logic programming interpreters support two extra-logical predicates, *assert* and *retract*, whose evaluation has the effect of adding or removing the specified Horn clause from the program in execution, respectively, causing a change in its behaviour. This is a pretty natural form of adaptation that fits perfectly in our framework by considering the same clauses of the program as control data. More precisely, this is an example of self-adaptivity, because the program itself can modify the control data.

Rule-based programming is another example of a very successful and widely adopted declarative paradigm, thanks to the solid foundations offered by rule-based machineries like term and graph rewriting. As many other programming paradigms, several rule-based approaches have been adapted or directly applied to adaptive systems (e.g. graph transformation [6]). Typical solutions include dividing the set of rules into those that correspond to ordinary computations and those that implement adaptation mechanisms, or introducing context-dependent conditions in the rule applications (which essentially corresponds to the use of standard configuration variables). The control data in such approaches are identified by the above mentioned separation of rules, or by the identification of the context-dependent conditions. Such identification is often not completely satisfactory and does not offer a neat and clear separation of concerns.

The situation is different when we consider rule-based approaches which enjoy higher-order or reflection mechanisms. A good example is *logical reflection*, a key feature of frameworks like rewriting logic. At the ground level, a rewrite theory $\mathcal{R}$ (e.g. software module) let us infer a computation step $\mathcal{R} \vdash t \to t'$ from a term (e.g. program state) t into t'. A universal theory $\mathcal{U}$ let us infer the computation at the meta-level, where theories and terms are meta-represented as terms: $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}}, \overline{t'})$. Since $\mathcal{U}$ itself is a rewrite theory, the reflection mechanism can be iterated yielding what is called the *tower of reflection*. This mechanism is efficiently supported by Maude [5] and has given rise to many interesting meta-programming applications like analysis and transformation tools.

In particular, the reflection mechanism of rewriting logic has been exploited in [11] to formalize a model for distributed object reflection, suitable for the specification of adaptive systems. Such model, suggestively called Reflective Russian Dolls (RRD), has a structure of layered configurations of objects, where each layer can control the execution of objects in the lower layer by accessing and executing the rules in their theories, possibly after modifying them, e.g., by injecting some specific adaptation logic in the wrapped components. Even at this informal level, it is pretty clear that the RRD model falls within our conceptual framework by identifying as "control data" for each layer the rules of its theory that are possibly modified by the upper layer. Note that, while the tower of reflection relies on a white-box adaptation, the russian dolls approach can deal equally well with black-box components, because wrapped configurations can be managed by message passing. The RRD model has been further exploited for modeling policy-based coordination [16] and for the design of PAGODA, a modular architecture for specifying autonomous systems [17].

Models of Concurrency. Languages and models emerged in the area of concurrency theory are natural candidates for the specification and analysis of autonomic systems. We inspect some (most) widely applied formalisms to see how the conceptual framework can help us in the identification of the adaptation logic within each model. Petri nets are without doubts the most popular model of concurrency, based on a set of repositories, called places, and a set of activities, called transitions. The state of a Petri net is called a marking, that is a

distribution of resources, called tokens, among the places of the net. A transition is an atomic action that consumes several tokens and produces fresh ones, possibly involving several repositories at once. Since the topology of the net is static, there is little margin to see a Petri net as an adaptive component: the only possibility is to identify a subset of tokens as control data. Since tokens are typed by repositories, i.e. places, the control data of a Petri net must be a subset CP of its "control" places. Tokens produced or removed from places in CP can enable or inhibit certain activities, i.e. adapt the net. The set CP can then be used to distinguish the adaptation logic from the application logic: if a transition modifies the tokens in CP, then it is part of the adaptation logic, otherwise it is part of the application logic. In particular, the transitions with self-loops on places in CP are those exploiting directly the control data in the application.

Mobile Petri nets allow the use of colored tokens carrying place names, so that the output places of a transition can depend on the data in the tokens it consumes. In this case, it is natural to include the set of places whose tokens are used as output parameters from some transition in the set of control places.

Dynamic nets allow for the creation of new subnets when certain transitions fire, so that the topology of the net can grow dynamically. Such "dynamic" transitions are natural candidates for the adaptation logic.

Classical process algebras (CCS, CSP, ACP) are certainly tailored to the modeling of reactive systems and therefore their processes easily fall within the hat of the interactive pattern of adaptation. Instead, characterizing the control data and the adaptation logic is more difficult in this setting. Since process algebras are based on message passing facilities over channels, an obvious attempt is to identify suitable adaptation channels. Processes can then be distinguished on the basis of their behavior on such channels, but in general this task is more difficult with respect to Petri nets, because processes will likely mix adaptation, interaction and computation.

The π-calculus, the join calculus and other nominal calculi, including higher-order versions (e.g. the HO π-calculus) can send and receive channels names, realizing some sort of reflexivity at the level of interaction: they have the ability to communicate transmission media. The situation is then analogous to that of dynamic nets, as new processes can be spawn in a way which is parametric with respect to the content of the received messages. If again we follow the distinction between adaptation channel names from ordinary channel names, then we inherit all the difficulties described for process algebras and possibly need sophisticated forms of type systems or flow analysis techniques to separate the adaptation logic from the application logic.

Paradigms with Reflective, Meta-level or Higher-Order Features. The same kind of adaptivity discussed for rewriting logic can be obtained in several other computational paradigms that, offering reflective, meta-level or higher-order features, allow one to represent programs as first-class citizens. In these cases adaptivity emerges, according to our definitions, if the program in execution is

represented in the control data of the system, and it is modified during execution causing changes of behaviour. Prominent examples of such formalisms, besides rewriting logic, are process calculi with higher-order or meta-level aspects (e.g. HO π-calculus, MetaKlaim), higher-order variants of Petri nets and Graph Grammars, Logic Programming, and programming languages like LISP, Java, C#, Perl and several others. Systems implemented in these paradigms can realize adaptation within themselves (self-adaptivity), but in general the program under execution can be modified also by a different entity, like an autonomic control loop written in a different language, or in the same language but running in a separate thread.

5 A Formal Model for our Framework

We propose a simple formal model inspired by our conceptual framework. Our main purpose is to provide a proof-of-concept that validates the idea of developing formal models of adaptive systems where the key features of our approach (e.g. *control data*) are first-class citizens. The model we propose is deliberately simple and based on well-known computational artifacts, namely transition systems.

Overall Setting. We recall that a *labelled transition system* (LTS) is defined as a triple $L = (Q, A, \rightarrow)$ such that Q is the set of states, A is the alphabet of action labels and $\rightarrow \subseteq Q \times A \times Q$ is the transition relation. We write $q \xrightarrow{a} q'$ when $(q, a, q') \in \rightarrow$ and we say that the system can evolve from q to q' via action a. Sometimes, a distinguished initial state q_0 is also assumed.

The first ingredient is an LTS S that describes the behaviour of a software component. It is often the case that S is not running in isolation, but within a certain environment. The second ingredient is a LTS E that models the environment and that can constrain the computation of S, e.g. by forbidding certain actions and allowing others. We exploit the following composition operator over LTSs to define the behaviour of S within E as the LTS $S||E$.

Definition 1 (Composition). *Given two LTSs* $L_1 = (Q_1, A_1, \rightarrow_1)$ *and* $L_2 = (Q_2, A_2, \rightarrow_2)$*, we let* $L_1||L_2$ *denote the labelled transition system* $(Q_1 \times Q_2, A_1 \cup A_2, \rightarrow)$*, where* $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$ *iff either of the following holds:* $q_i \xrightarrow{a}_i q_i'$ *for* $i = 1, 2$ *with* $a \in A_1 \cap A_2$*;* $q_i \xrightarrow{a}_i q_i'$ *and* $q_j' = q_j$ *for* $\{i, j\} = \{1, 2\}$ *with* $a \in A_i \setminus A_j$*.*

Note that in general it is not required that $A_1 = A_2$: the transitions are synchronised on common actions and are asynchronous otherwise.

Since adaptation is usually performed for the sake of improving a component's ability to perform some task or fulfill some goal, we provide here a very abstract but flexible notion of a component's objective in form of logical formulae. In particular, we let ψ be a formula (expressed in some suitable logic) characterizing the component's goal and we denote with the predicate $L \models \psi$ the property of the LTS L satisfying ψ. Note that it is not necessarily the case that $L \models \psi$ gives a yes/no result. For example, we may expect $L \models \psi$ to indicate how well L fits ψ, or the likelihood that L satisfies ψ. In the more general case, we can assume that $L \models \psi$ evaluates to a value in a suitable domain. We write $L \not\models \psi$ when L does not fit ψ, e.g. when the value returned is below a given threshold.

Adaptable vs non-adaptable Components. In a perfect but static world, one would engineer the software component S by ensuring that $S||E \models \psi$ and live happily afterwards (if such an S can be found). This is not realistic: the analyst has only a partial knowledge of E; S must be designed for running in different environments; the environment may change in an unpredictable manner by external causes while S is running; the goal ψ may be superseded by a more urgent goal ψ' to be accomplished. Roughly, we can expect frequent variations of E and possible, but less frequent, variations of ψ. The component is adaptable if it can cope with these changes in E and ψ by changes in its control data.

When S has no control data the component is not adaptable. The other extreme is when the whole S is the control data. Indeed an LTS can be represented and manipulated in several forms: as a list of transitions or as a transition matrix when it is finite; as a set of derivation rules when it is finitely specified.

Most appealing is the case when S is obtained as the combination of some statically fixed control FC and of some control data CD, i.e., $S = FC||CD$. Then, adaptavity is achieved by plugging-in a different control data CD' in reaction to a change in the environment from E to E' (with $S||E' \not\models \psi$ and $FC||CD'||E' \models \psi$), or to a change in the goal from ψ to ψ' (with $S||E \not\models \psi'$ and $FC||CD'||E \models \psi'$), or to a change in both.

We assume here that the managed component FC is determined statically such that it cannot be changed during execution and that each component may run under a unique manager CD at any time. However, adaptable components come equipped with a set of possible alternative managers $CD_1, ..., CD_k$ that can be determined statically or even derived dynamically during the computation.

Knowledge-Based Adaptation. Ideally, given FC, E and ψ it should be possible for the manager to select or construct the best suited control data CD_i (among the available choices) such that $FC||CD_i||E \models \psi$ and install it over FC. However, in real cases E may not be known entirely or may be so large that it is not convenient to represent it exactly. Therefore, we allow the manager to have a perfect knowledge of FC and of the goal ψ, but only a partial knowledge of E, that we denote by O and call the *observed environment*, or *context*.

The context O is derived by sensing the component's run-time environment. In general we cannot expect O and E to coincide: first, because the manager has limited sensing capabilities and second because the environment may be changed dynamically by other components after it has been sensed. Thus, O models the current perception of the environment from the viewpoint of the component.

The context O is expected to be updated frequently and to be used to adapt the component. This means that CD is chosen on the basis of FC, O and ψ, and that the manager can later discover that the real environment E differs from O in such a way that $FC||CD||E \not\models \psi$ even if $FC||CD||O \models \psi$. When this occurs, on the basis of the discovered discrepancies between E and O, a new context O' can be sensed to approximate E better than O, and O' can be used to determine some control data CD' in such a way that $FC||CD'||O' \models \psi$.

Self-adaptive Components. If the available control data strategies $CD_1, ..., CD_k$ are finitely many and statically fixed, then some precompilation can be applied that facilitates the adaptation to the changing environment, as explained below.

We assume that, given FC, ψ and any CD_i we can synthesize the weakest precondition ϕ_i on the environment such that $O \models \phi_i$ implies $FC||CD_i||O \models \psi$. Then, when the context changes from O to O', the manager can just look for some ϕ_j such that $O' \models \phi_j$ and then update the control data to CD_j.

Definition 2 (Self-adaptive Component). *A* self-adaptive component *is a tuple* $\langle FC, \mathcal{CD}, \psi, \alpha_\psi \rangle$ *where* FC *models the managed component;* $\mathcal{CD}$ *is a family of control data;* ψ *is the component's goal; and* $\alpha_\psi : \mathcal{O} \times \mathcal{CD} \to \mathcal{CD}$ *is a function that given a context* $O \in \mathcal{O}$ *and the current control data* CD *returns a control data* CD' *such that* $FC||CD'||O \models \psi$.

Enforcing the analogy of LTS based control, a possible formalization of the control manager of a self-adaptive component can be given as the composition of two LTSs: a fixed manager FM and the control data MCD defined as follows. The set of states of FM is $\mathcal{CD}$, and its transitions are labelled by context/goal pairs: for any CD, CD', O, ψ we have a transition $CD \xrightarrow{O,\psi} CD'$ iff $\alpha_\psi(O, CD) = CD'$. The LTS MCD has a single state and one looping transition labelled with the current context O and the current goal ψ. The composition $FM||MCD$ constrains the manager to ignore all transitions with labels different from O, ψ. The manager updates the control data of the managed component according to its current state. If CD' is the preferred strategy for O, ψ but CD is the current strategy, then the manager will move to CD' and then loop via $CD' \xrightarrow{O,\psi} CD'$.

Stacking Adaptive Components. Pushing our formal model further, by exploiting the control data of $\langle FC, \mathcal{CD}, \psi, \alpha \rangle$ we can add one more manager on top of the self-adaptive component, along the tower of adaptation (§3).

This second-level control manager can change the structure of MCD. For example, just by changing the label of its sole transition this (meta-)manager can model changes in the context, in the current goal, or in both.

However, one could argue that also other elements of the self-adaptive component could be considered as mutable. For example, one may want to change at run-time the adaptation strategy α_ψ that resolves the nondeterminism when there are several control data that can be successfully used to deal with the same context O, or even the set of available control data $\mathcal{CD}$, for example as the result of a learning mechanism. This can be formalized easily by exposing a larger portion of FM as control data.

Needless to say, also the above meta-manager can be designed as an adaptable component, formalizing its logic via a suitable LTS that exposes some control data to be managed by a upper level control manager, and so on.

6 Conclusion and Future Developments

We presented a conceptual framework for adaptation, where a central role is played by the explicit identification of the control data that govern the adaptive behavior of components. As a proof of concept we have discussed how systems conforming to well-accepted adaptive architectures, including IBM's MAPE-K and several adaptive patterns, fit into our framework. We have also considered several representative instances of our approach, focusing on foundational models of computation and programming paradigms, and we proposed a simple formalization of our concepts based on labelled transition systems.

We plan to exploit our conceptual framework by developing sound design principles for architectural styles and patterns in order to ensure correctness-by-design, and guidelines for the development of adaptive systems conforming to such patterns. For instance, we might think about imposing restrictions on the instances of our framework such as requiring an explicit separation of the component implementing the application logic from the component modifying the control data, in order to avoid self-adaptation within an atomic component and to guarantee separation of concerns, and an appropriate level of modularity.

We also plan to develop analysis and verification techniques for adaptive systems grounded on the central role of control data. Data- and control-flow analysis techniques could be used to separate, if possible, the adaptation logic from the application logic. This could also reveal the limits of our approach in situations where the adaptation and the application logics are too entangled.

Another current line of research aims at developing further the reflective, rule-based approach (§4). Starting from [11] we plan to use the Maude framework to develop prototype models of archetypal and newly emerging adaptive scenarios. The main idea is to exploit Maude's meta-programming facilities (based on logical reflection) and its formal toolset in order to specify, execute and analyze those prototype models. A very interesting road within this line is to equip Maude-programmed components with formal analysis capabilities like planning or model checking based on Maude-programmed tools.

Even if we focused the present work on adaptation issues of individual components, we also intend to develop a framework for adaptation of *ensembles*, i.e., massively parallel and distributed autonomic systems which act as a sort of swarm with emerging behavior. This could require to extend our *local* notion of control data to a *global* notion, where the control data of the individual components of an ensemble are treated as a whole, which will possibly require some mechanisms to amalgamate them for the manager, and to project them backwards to the components. Also, some mechanisms will be needed to coordinate the adaptation of individual components in order to obtain a meaningful adaptation of the whole system, in the spirit of the *overlapping adaptation* discussed in [19].

Last but not least, we intend to further investigate the connection of our work with the other approaches presented in the literature for adaptive, self-adaptive and autonomic systems: due to space limitation we have considered here just a few such instances.

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: SEAMS 2009, pp. 38–47. IEEE Computer Society (2009)
2. Andrade, L.F., Fiadeiro, J.L.: An architectural approach to auto-adaptive systems. In: ICDCS Workshops 2002, pp. 439–444. IEEE Computer Society (2002)
3. Broy, M., Leuxner, C., Sitou, W., Spanfelner, B., Winter, S.: Formalizing the notion of adaptive system behavior. In: Shin, S.Y., Ossowski, S. (eds.) SAC 2009, pp. 1029–1033. ACM (2009)
4. Cabri, G., Puviani, M., Zambonelli, F.: Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In: Smari, W.W., Fox, G.C. (eds.) CTS 2011, pp. 508–515. IEEE Computer Society (2011)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal Analysis and Verification of Self-Healing Systems. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 139–153. Springer, Heidelberg (2010)
7. Ghezzi, C., Pradella, M., Salvaneschi, G.: An evaluation of the adaptation capabilities in programming languages. In: Giese, H., Cheng, B.H. (eds.) SEAMS 2011, pp. 50–59. ACM (2011)
8. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology (2001)
9. IBM Corporation: An Architectural Blueprint for Autonomic Computing (2006)
10. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
11. Meseguer, J., Talcott, C.: Semantic Models for Distributed Object Reflection. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
12. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. Intelligent Systems and their Applications 14(3) (1999)
13. Pavlovic, D.: Towards Semantics of Self-Adaptive Software. In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) IWSAS 2000. LNCS, vol. 1936, p. 50. Springer, Heidelberg (2001)
14. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4(2) (2009)
15. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A programming paradigm for autonomic systems. CoRR abs/1105 0069 (2011)
16. Talcott, C.L.: Coordination models based on a formal model of distributed object reflection. In: Brim, L., Linden, I. (eds.) MTCoord 2005. ENTCS, vol. 150(1), pp. 143–157. Elsevier (2006)
17. Talcott, C.L.: Policy-based coordination in PAGODA: A case study. In: Boella, G., Dastani, M., Omicini, A., van der Torre, L.W., Cerna, I., Linden, I. (eds.) CoOrg 2006 & MTCoord 2006. ENTCS, vol. 181, pp. 97–112. Elsevier (2007)
18. Weyns, D., Malek, S., Andersson, J.: FORMS: a formal reference model for self-adaptation. In: Figueiredo, R., Kiciman, E. (eds.) ICAC 2010, pp. 205–214. ACM (2010)
19. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) ICSE 2006, pp. 371–380. ACM (2006)

Applying Design by Contract to Feature-Oriented Programming

Thomas Thüm[1], Ina Schaefer[2], Martin Kuhlemann[1], Sven Apel[3], and Gunter Saake[1]

[1] University of Magdeburg, Germany
[2] University of Braunschweig, Germany
[3] University of Passau, Germany

Abstract. *Feature-oriented programming* (FOP) is an extension of object-oriented programming to support software variability by *refining* existing classes and methods. In order to increase the reliability of all implemented program variants, we integrate *design by contract* (DbC) with FOP. DbC is an approach to build reliable object-oriented software by specifying methods with contracts. Contracts are annotations that document and formally specify behavior, and can be used for formal verification of correctness or as test oracles. We present and discuss five approaches to define contracts of methods and their refinements in FOP. Furthermore, we share our insights gained by performing five case studies. This work is a foundation for research on the analysis of feature-oriented programs (e.g., for verifying functional correctness or for detecting feature interactions).

1 Introduction

Feature-oriented programming (FOP) [21,7] is a programming paradigm supporting software variability by modularizing object-oriented programs along the features they provide. A feature is an end-user-visible program behavior [15]. Code belonging to a feature is encapsulated in a feature module. A feature module can introduce classes or modify existing classes by adding or refining fields and methods. A program variant is generated by composing the feature modules of the desired features. We use formal methods to increase the reliability of all program variants that can be generated from a set of feature modules.

Design by contract (DbC) [20] has been proposed as a means to obtain reliable object-oriented software. The key idea is to specify each method with a contract consisting of a precondition and a postcondition. The precondition formulates assumptions of the method that the caller of the method has to ensure. The postcondition provides guarantees that the method gives such that the caller can rely on it. Additionally, class invariants specify properties of objects that hold before and must hold after a method call. DbC can be used for formal specification and documentation of program behavior as well as for formal verification or testing of functional correctness. We integrate DbC with FOP to increase the reliability of FOP.

J, de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 255–269, 2012.

```
class Array {                                                    Base
  Item[] data; //@ invariant data != null;
    Array(Item[] data) { this.data = data; }
    /*@ requires \nothing;
      @ ensures (\forall int i; 0 < i && i < data.length;
      @   data[i-1].key <= data[i].key); @*/
    void sort() { /* heap sort algorithm */ }
}
class ArrayWithInverse extends Array { /* ... */ }
class Item {
    int key; Object value; //@ invariant value != null;
    Item(key, value) { this.key = key; this.value = value; }
}
```

Fig. 1. *Design by contract with Java and JML*: method contracts and class invariants are embedded in comments

FOP adds another dimension of modularization and code reuse to object-oriented programs besides inheritance. While in class-based inheritance, subclasses must satisfy the behavioral subtyping principle [17], method refinement (i.e., method overriding in FOP) is different in nature from code reuse by inheritance. A feature may change the behavior of an existing method arbitrarily to meet feature-specific requirements. For example, a security feature may restrict the allowed parameter values of a method by strengthening the precondition. Thus, when integrating DbC with FOP, the question arises how method contracts of refined methods should be defined.

We present and discuss five new approaches to specify contracts of methods which we refine using FOP. We consider the strengths and weaknesses of each approach with respect to strictness, expressiveness, complexity, and specification clones. Furthermore, we discuss the refinement of class invariants and evaluate the practical applicability of the presented approaches using five case studies. This paper is the first to focus on the specification of feature-oriented programs using DbC. Previous work focused on ensuring consistency of feature-oriented programs using type checking [3,10] and model checking [5,6]. With our systematic analysis of the different approaches to specify feature-oriented programs using DbC, we provide the foundation for future research on the formal analysis of feature-oriented programs, including the formal verification of functional correctness, feature interaction detection, and test case generation.

2 Background

Figure 1 shows our running example — a Java program that is annotated with the *Java Modeling Language (JML)* [16] to specify its behavior using DbC. Class `Array` is specified by an invariant (using the keyword `invariant`) that states that field `data` should not be null. Invariants have to be established by the class constructors, they can be assumed before every method call and have to be reestablished afterwards. Method `sort()` of class `Array` is specified by a method contract. The precondition of the contract is expressed in the `requires` clause

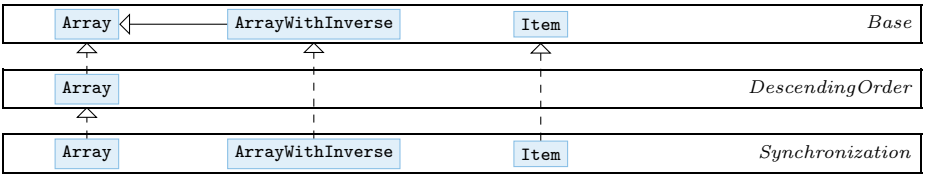


Fig. 2. Feature-oriented class refinement (dashed arrows) and object-oriented inheritance (solid arrows) are concepts for reuse that are orthogonal to each other

and has to be ensured by the caller of the method. Here, the precondition is simply true. In JML, behavioral subtyping [17] for subclasses is achieved by specification inheritance. This means that all subclasses inherit the invariants of their superclasses and that overriding methods must also satisfy the contracts of the overridden methods. The `ensures` clause expresses the postcondition of a contract and has to be guaranteed by the method. In our example, the postcondition states that the resulting array is sorted. Contracts can also be denoted by Hoare triples [13]. Given a method m with precondition ϕ and postcondition ψ, the contract of method m is denoted by $\{\phi\}m\{\psi\}$.

Feature-oriented programming (FOP) is an extension of object-oriented programming (OOP) aiming at better reuse capabilities across families of object-oriented programs [21]. Classes are split into pieces distributed over feature modules; modules that implement end-user-visible features. A particular program can be derived automatically by combining the feature modules of the required features [2]. A feature module can introduce new classes, methods, and fields. If a method with a particular name already exists in a previously composed feature module, the existing method is refined [2]. Method refinement is similar to overriding with object-oriented inheritance, but the FOP keyword `original` is used instead of `super`. The main difference is that `original` is bound at the time the feature modules are composed. Figure 2 visualizes the FOP refinement of the classes of Figure 1 (`Array`, `ArrayWithInverse`, `Item`) with the feature modules *Base*, *DescendingOrder*, and *Synchronization*. *Base* contains the classes `Array`, `ArrayWithInverse`, and `Item`. *DescendingOrder* contains a class refinement `Array` which refines class `Array` of *Base* to invert the sorting order of implemented arrays. *Synchronization* contains refinements for all classes of *Base*; as a result, these classes support multithreading.

3 Contracts for Feature-Oriented Programming

We present five approaches for the integration of DbC into FOP and discuss advantages and disadvantages of each approach.

Plain Contracting. The application of DbC to FOP should be as simple as possible to facilitate creation and maintenance of contracts for programmers.

```
refines class Array {                                          StableSort
    /*@ requires original;
      @ ensures original && ...sorting is stable...; @*/
    void sort() { /* merge sort algorithm */ }
}
```

Fig. 3. *Explicit contract refinement*: feature *StableSort* overrides method `sort()` with an implementation of a stable sorting algorithm. Both, precondition and postcondition maintain the refined contract indicated by the keyword `original` and refine it.

Plain contracting is the simplest possible approach allowing programmers to define contracts only for method introductions and not for method refinements. As a consequence, method refinements may not change the behavior of the refined method. Consider the example in Figure 1. Assume that an additional feature *Quicksort* refines the class `Array` by overriding the body of method `sort()` with a Quicksort implementation. The contract of method `sort()` does not have to be changed, because the new implementation does not affect sorting. Given a set of selected features and a total order on those features, a tool can decide for every method whether it is a method introduction or a method refinement [3]. Then, we can automatically check that no method refinement comes with a contract.

On the one hand, allowing programmers to introduce, but not to refine contracts comes with advantages. First, we only need to specify a method once even if it is refined by several other feature modules, and thus the effort for specification (i.e., writing contracts) is minimal. Second, the source code is easier to understand as the same contract holds in every possible combination of features. This is beneficial since a programmer needs to know the contract for every called method (e.g., to find out whether the precondition is fulfilled at every position where the method is called). On the other hand, this approach might be too restrictive. With plain contracting, we are not able to specify feature-oriented programs, where the refinement of a method also requires the refinement of a contract. For instance, if we replace an instable sorting algorithm with a stable one, we may need to express that callers can rely on this property if the according feature is present. In Section 6, we evaluate whether this restriction is an actual problem in practice.

Explicit Contract Refinement. When refining a method, we may also need to refine the corresponding contract if the method behavior is changed such that it no longer satisfies the original contract. The refinement of contracts can be supported by the same linguistic means as method refinement, which should raise the acceptance of DbC in FOP. Explicit contract refinement allows programmers to use the keyword `original` to refer the refined precondition and postcondition in the contract refinement.

As an example for explicit contract refinement, in Figure 3, we assume that feature *Base* is identical to the previous example and that a new feature *StableSort* replaces the sorting algorithm by a stable sorting algorithm; here, merge

sort. In order to provide a contract, which states that the result is sorted and the algorithm is stable, we refer to the existing postcondition and conjoin it with a definition of stability (which we left out for brevity). Keyword `original` may appear anywhere in the precondition or postcondition (not necessarily at the beginning) or it may not appear at all.

Explicit contract refinement is a flexible approach where contracts can be refined by including the previous contract if appropriate; preconditions and postconditions can be refined individually. However, the approach may lead to complex and less understandable specifications, especially, when several refinements for the same method contract exist and some, but not all refinements, refer to the previous contract. It may be unclear what a method actually needs to ensure and what it can rely on, because this may depend on the feature selection. In particular, contracts depend on the feature from which the method is called.

Consecutive Contract Refinement. Consecutive contract refinement is an approach with which new contracts can be defined for method refinements but contracts for refined methods may not be invalidated. The central idea of the approach is to adapt contract subtyping to FOP. Contract subtyping is widely used in OOP and ensures that contracts defined in a certain class must be fulfilled in all subclasses, too. The main difference to contract subtyping in OOP is that features may be present or not, and thus the feature selection influences the resulting method contract.

Given an original method m with precondition ϕ and postcondition ψ, we can refine m with a new method implementation m' with precondition ϕ' and postcondition ψ'. Then, the refined method m' needs to ensure the original contract $\{\phi\}m'\{\psi\}$ and the new contract $\{\phi'\}m'\{\psi'\}$. As a result, the method can be used in all places where method m is called, and the caller can rely on the contract of the refined method m. For example, re-consider the feature *StableSort* in Figure 3. With consecutive contract refinement, the example would look the same except for the replacement of '`original`' with '`true`' in the precondition and postcondition, because the contract of the refined method holds implicitly.

The main advantage of consecutive contract refinement compared to explicit contract refinement is that existing contracts remain valid even if a method is refined. This way, callers can rely on contracts defined in a particular feature independent on the presence of other features, because refinements cannot invalidate the contract. This advantage comes with a reduced applicability, since we cannot encode cases in which a feature weakens an existing contract.

Contract Overriding. Contract overriding is a special case of explicit contract refinement where the keyword `original` is never used. Contract overriding allows programmers to replace the contract when refining a method, but does not allow programmers to reference or reuse refined contracts. In contrast to consecutive contract refinement, contracts defined in previous features do not need to be fulfilled. In previous work, we used contract overriding to verify SPL products by proof composition [23]. In this previous work, we additionally enforced compatibility between contracts and their refinements. A contract refinement

is compatible to a previous contract, if every method that fulfills the refined contract also satisfies the contract of the refined method.

The main problem with contract overriding are specification clones, because there is no way to adapt original contracts. The CPA (copy, paste, adapt) principle is the only option to refine contracts, which may result in many specification clones and, thus, a high specification effort. Another serious disadvantage is that the meaning of a contract is unclear for callers, because it heavily depends on the actual feature selection and on the composition ordering. Furthermore, if two features refine the same method contract using contract overriding, we may get undesired contracts if both features are selected (known as feature interaction problem of FOP [2]). We could introduce derivative contracts (i.e., a contract that is only included if two or more features are selected) but derivative contracts can introduce further specification clones.

Pure-Method Refinement. Preconditions and postconditions in JML may also contain calls to methods that are free of side-effect and are guaranteed to terminate (known as *pure* method in JML [16]). If a pure method is used in a contract, the contract depends on the result of this (pure) method call. Pure methods called in contracts open a further possibility for contract refinement, because pure methods can be refined as any other method in FOP – this allows programmers to refine contracts as a spin-off. With pure-method refinement, instead of actually refining a contract itself, a pure method used in a contract is refined and, thus, indirectly contracts based on the feature selection are modified.

In Figure 4, the example of pure-method refinement is based on an publicly available case study[1], which we have decomposed into features. Class `ExamDataBase` stores the results of student exams. Array `students` contains the students and their points, whereas a `null`-value refers to a free position in the array. The method `consistent()` checks whether all students have at least zero points. The method `validStudent()` is used in the contract of method `consistent()` and is refined by a class refinement of feature module *BackOut*; this refinement allows students to back out from an exam. Hence, the contract of method `consistent()` is refined by changing the body of method `validStudent()`.

Pure methods in contracts support fine-grained contract refinement. Even parts of preconditions or postconditions can be refined, which would otherwise require to clone contracts and modify them. Such specification clones may lead to similar problems as code clones [14]. For example, when updating a contract, we may forget to update clones of this contract and introduce inconsistencies. Hence, specification clones should be avoided whenever possible requiring more sophisticated specification approaches such as pure-method refinement.

Pure-method refinement is expressive, because method refinements do neither depend on refined methods nor must relate to them in any way (e.g., weakening or strengthening existing contracts).

[1] http://verifythis.cost-ic0701.org/post?pid=database-system-for-managing-exams

Base

```
class ExamDataBase {
    /*@ ensures \result == (\forall int i; 0 <= i
      @     && i < students.length && validStudent(students[i]);
      @   students[i].points >= 0); @*/
    boolean consistent() {
        for(int i=0; i<students.length; i++)
            if (validStudent(students[i]) && students[i].points < 0)
                return false;
        return true;
    }
    /*@ pure @*/ boolean validStudent(Student student) {
        return student != null;
    }
}
class Student {
    //@ invariant matrNr > 0 && firstname != null && surname != null;
  int matrNr; String firstname, surname;
}
```

BackOut

```
refines class ExamDataBase {
    /*@ pure @*/ boolean validStudent(Student student) {
        return original(student) && !student.backedOut;
    }
}
refines class Student {
    //@ invariant !backedOut || backedOutDate != null;
  Date backedOutDate = null; boolean backedOut = false;
}
```

Fig. 4. *Pure-method refinement*: the contract of method `consistent()` contains a call to the pure method `validStudent()`. Feature *BackOut* refines the contract of `consistent()` indirectly by refining method `validStudent()`. By refining one pure method, we can refine several contracts indirectly at the same time.

A further advantage is that no new keywords and no linguistic concepts are needed for contract refinement, because traditional FOP mechanisms can be used. Hence, it is easy to understand the meaning of contracts, if the refinements of all pure methods therein are clear. The main disadvantage of pure-method refinement is that it strongly relies on the concept of pure methods being allowed to be called in contracts. Furthermore, the flexibility for refining methods by FOP may cause contracts which are hard to understand (e.g., if we have several refinements of the same method, some strengthening, some weakening, and some overriding).

4 Refinement of Invariants

DbC involves the specification of methods by contracts and classes by invariants usually expressing invariant properties of the fields. In the following, we assume that contract refinement is carried out with any of the previously discussed approaches and discuss how programmers can refine invariants analogously.

If invariants can be introduced in features, an invariant only needs to be established for the resulting program if the corresponding feature is selected (e.g., in Figure 4 feature *BackOut* introduces fields together with an invariant). Thus,

we can build variable specifications using invariant introductions. Similarly to contracts, invariants can be refined explicitly or implicitly (i.e., with or without a keyword referring to invariants defined in previously composed feature modules). When using *explicit invariant refinement*, we can use the keyword `original` to reference the previous definition of the invariant and combine it with the previous invariants. Applying *consecutive contract refinement* means that features can only add new invariants that need to hold as well. We can apply the concept of *pure-method refinement* to invariants. If an invariant contains a pure method call, the pure method can be refined using FOP method refinement. Finally, *contract overriding* can also be applied to invariants, where existing invariants can be overridden by features which we refer to as *invariant overriding*.

Allowing the refinement of invariants provides additional flexibility for the specification of feature-oriented programs. Every feature module can change invariants provided by previously composed feature modules. Depending on the approach chosen for refinement of contracts, we find it intuitive to refine invariants using the same means. However, the introduction and refinement of invariants allows that particular invariants only need to be fulfilled if a certain feature is present. As a result, it can be difficult to examine those combinations of features for which a certain invariant is present. The refinement of invariants has huge consequences as an invariant must hold for *all* methods of a class, and a change influences many callers and callees at the same time. Furthermore, the flexibility with invariant refinement can easily result in specifications that cannot be satisfied by any implementation. In Section 6, we evaluate whether the refinement of invariants is actually useful in practice.

5 Comparison

After presenting five alternative approaches of refining contracts, we now want to compare them based on properties directly related to specifications and give some intuition which approach is useful under which circumstances. We compare the approaches according to four properties which are different perspectives on the specification of programs: strictness, expressiveness, complexity, and specification clones. While strictness and expressiveness may indicate that an approach can not be applied to certain feature-oriented programs, the other criteria refer to properties that are nice to have.

Strictness can be used to classify all presented approaches regarding allowed and disallowed refinements from a logical point of view. Given a certain contract C, a refined contract C' may be strengthened with respect to method calls (e.g., by adding a further postcondition) or weakened (e.g., by requiring a further precondition). Strengthening means that every method fulfilling C' also fulfills C and weakening means that every method fulfilling C also fulfills C'. Further possibilities are to refine the contract with an equivalent one (e.g., by commuting preconditions or leaving the contract as-is) or to refine the contract in arbitrary way. In Figure 5, we illustrate the strictness relation by a Venn diagram. The intersection of weakened and strengthened contracts is the set of

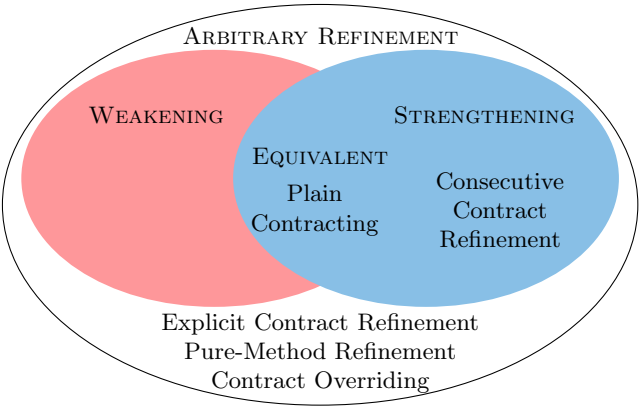


Fig. 5. Comparison of the presented approaches of contract refinement regarding strictness. Approaches may allow or disallow weakening and strengthening of contracts resulting in four categories. For example, disallowing both means to allow only contract refinements if they are equivalent to the original contract.

equivalent contracts. As plain contracting disallows any refinement of contracts, the contracts are equivalent for every method refinement. Consecutive contract refinement allows only to strengthen the original contract. All other presented approaches allow arbitrary refinements.

Expressiveness refers to whether we can specify all meaningful properties of feature-oriented programs. Given a particular program, we need to know whether we can express its specification with a certain approach or not. There is a connection to strictness: approaches allowing arbitrary refinements are more expressive than approaches allowing only strengthened contracts and similarly, strengthening is more expressive than equivalent contracts. In Table 1, we give an overview on the expressiveness of all presented approaches. The low expressiveness of plain contracting and consecutive contract refinement is simply based on their strictness. Contract overriding has a lower expressiveness compared to other approaches allowing arbitrary refinements, because derivative contracts may be needed if two features refine the same contract (see Section 3).

Complexity indicates whether it is easy for a programmer to manually retrieve the resulting contract of a certain method for a particular feature combination. An approach, in which determining the contract has the lowest complexity, is beneficial for programmers that need to create and maintain specifications because mistakes, such as wrong contracts or wrong implementations, can have expensive outcomes (e.g., it is expensive to detect errors using verification or testing). Thus, we expect contract specifications to have a low complexity. Contract overriding has the highest complexity, as contracts can be arbitrarily refined by each feature, and contracts can depend on the presence of every single feature. Contracts created by explicit contract refinement have a lower complexity since no derivative contracts are needed. Using pure-method refinement, contracts can

Table 1. Comparison of the presented approaches for the refinement of contracts. + + means that the approach is good with respect to the property (i.e., the approach has high expressiveness, contracts have a low complexity, specification clones can be avoided). Intuitively, – – refers to the worst and 0 to a neutral evaluation.

	Plain Contracting	Explicit Refinement	Consecutive Contract Ref.	Pure Method Refinement	Contract Overriding
Expressiveness	– –	+ +	0	+ +	+
Complexity	+ +	– –	+	0	– –
Specification Clones	+ +	0	+	+ +	– –

only be refined at predefined positions. Consecutive contract refinement only allows a programmer to strengthen contracts meaning that if a certain feature is selected, then all methods need to establish the contracts defined therein, independent of other features. Clearly, the complexity is even lower if we do not allow refinements at all using plain contracting, because a contract is either not present or the same for all feature selections.

Specification clones are identical or very similar contracts. We expect that specification clones lead to similar problems as code clones (see Section 3). Hence, a specification approach should help to avoid specification clones. We consider contract overriding as the worst approach regarding specification clones, as it provides no ability to reuse contracts such that the only option is to copy and adapt contracts. A better approach is the explicit refinement of contracts and invariants because the keyword `original` can be used to reference preconditions and postconditions of a previous contract. With consecutive contract refinement, all contracts are implicitly reused such that we expect even less specification clones. The best approaches in terms of avoiding clones are plain contracting and pure-method refinement. Plain contracting completely disallows contract refinements, and with pure-method refinement even parts of contracts can be refined which allows to reuse existing contracts.

6 Evaluation

In order to evaluate the practicability of the five proposed specification approaches, we performed two case studies by creating feature-oriented programs including their specifications from scratch and three case studies by decomposing already specified object-oriented programs into feature modules. All our case studies are implemented and specified in feature-oriented extensions of Java and JML, but we expect similar results for other object-oriented languages and contract-based specification languages. The advantage of Java and JML is that many tools as well as specified and verified sample programs exist. However, it turned out that most existing examples are too small to be decomposed into features (i.e., only three of them were suitable for decomposition).

Table 2. Results of case studies

	ExamDB	Paycard	DiGraph	BankAccount	IntList
Classes, fields	4, 10	8, 42	8, 13	2, 7	2, 2
Methods (pure)	29 (8)	18 (5)	48 (22)	10 (0)	12 (0)
Features, variants	4, 8	4, 6	4, 8	6, 24	5, 16
Method refinements (pure)	2 (2)	3 (1)	0 (0)	4 (0)	4 (0)
Contracts (in core features)	25 (17)	10 (4)	43 (27)	8 (2)	7 (1)
Invariants (in core features)	5 (4)	6 (2)	12 (12)	4 (1)	3 (2)
Contract refinements	0	1	0	2	1
Contracts with method calls (refined, multiple)	8 (7, 4)	2 (2, 0)	29 (0, 10)	0 (0, 0)	0 (0, 0)
Invariant refinements	0	0	0	0	0
Invariants with method calls (refined, multiple)	0 (0, 0)	0 (0, 0)	5 (0, 0)	0 (0, 0)	0 (0, 0)

In Table 2, we present some statistics of our feature-oriented sample programs. They have between two and eight classes consisting of two to 42 fields and ten to 48 methods. Some methods are declared as being pure. Our case studies have four to six features where six to 24 combinations of features are considered valid and can be used to generate different program variants. The programs are specified by seven to 43 contracts and three to twelve invariants.

With respect to *strictness and expressiveness* of the approaches, we found that four of five case studies could not be specified using plain contracting, because contract refinement was required. Only, the DiGraph case study could be specified with plain contracting; it does not contain a single method refinement as it is a library and the features chosen for decomposition do not cross-cut method implementations. But, method and contract refinement may be necessary when extracting further features or extending DiGraph with a new features. Contract strengthening is sufficient for three of five case studies. We specified the IntList and the Paycard case studies using consecutive contract refinement. Thus, for these case studies strengthening is sufficient. ExamDB and BankAccount rely on contract weakening. While contract strengthening is commonly used for OOP, it is not suited for any feature-oriented program. In larger programs, we expect even more examples where contract weakening is needed.

Our results show that some, but not all feature-oriented method refinements *require contract refinements*. For example, the BankAccount case study contains four method refinements, but for only two of them the contract was refined. Conversely, pure-method refinement requires the refinement of methods per definition, but some method refinements may be introduced only to refine contracts (i.e., the method refinement is not needed for implementation of features but only to express their specification). For example, in the ExamDB case study, we newly introduced two refinements of pure methods to actually refine seven contracts each containing a call to the pure method.

The *granularity of contract refinement* can influence the suitability of the individual approaches. The case study ExamDB requires fine-grained refinement of contracts. In Figure 4, the contract of method `consistent()` is refined using pure-method refinement. The contract quantifies over all valid students, and feature *BackOut* can actually influence which students are valid (students that are backed-out are considered as invalid). In this example, only a small part of a contract needs to be refined, while most of it remains unchanged. Hence, we used pure-method refinement for ExamDB to express these fine-granular refinements. All other approaches would lead to specification clones. But, we also observed the danger that pure-method refinement is applied *accidentally*. When decomposing an existing system into features, the implementation may require the refinement of certain methods. If one of such methods is declared as pure, it may also be used in contracts. But then, we may accidentally refine contracts or invariants simply by refining these methods. If we choose to disallow pure-method refinement, we also need to make sure that either no pure method can be refined or that no method referenced in contracts or invariants can be refined. The same holds if we create a feature-oriented program from scratch.

In the case study Paycard, we used a *combination of two approaches*. We used pure-method refinement to refine two contracts, because the refinement was fine-grained. But, for another contract refinement, we used consecutive contract refinement as the whole original contract should be established as-is and refined by a further contract. The experience with our case studies showed that even combinations of presented approaches may be useful.

Not a single case study required the *refinement of invariants* (see Figure 2). Still, in all case studies except from DiGraph, invariants were introduced by several, optional features resulting in invariants that only hold for products of particular feature combinations. But, we found no case where a feature needed to refine the invariant defined by another feature. However, we had to split invariants into several smaller invariants when decomposing the ExamDB and Paycard case studies into features. Splitting was possible as the invariant actually was a conjunction, which can always be decomposed into several invariants. We cannot conclude that the refinement of invariants can generally be avoided, but at least *in our case studies* the introduction of invariants by features was sufficient. This is a positive result according to the strong disadvantages of invariant refinement discussed in Section 4.

In our case studies, we also analyzed whether a *global specification* that holds for all program variants is sufficient as suggested by Liu et al. [18]. Their example is that every pacemaker variant shall generate a pulse when no heartbeat is detected. In Table 2, we observe that only between 14 and 68 percent of all contracts and between 25 and 100 percent of invariants are given in core features. A *core* feature is a feature that is included in every program variant [8]. The core features together build-up the part that is common to all program variants. From the above figures, we can conclude that in none of our case studies a global specification is sufficient and specifications in form of contracts should be given for every feature as we propose in this paper.

In summary, our evaluation showed that contract refinement is needed when applying DbC to FOP. It is not always sufficient to only strengthen contracts (already in our small case studies) such that an approach for contract refinement should also allow weakening. From our qualitative and quantitative analysis, pure-method refinement is the most promising approach because contracts can be strengthened or weakened and fine-grained refinements are supported as well. Pure-method refinement may be combined with consecutive contract refinement to better support coarse-grained refinements. In our experience, invariant introductions should be used instead of invariant refinements whenever possible.

7 Related Work

In previous work, we considered formal verification of feature-oriented programs based on JML specifications. We proposed proof composition with the proof assistant Coq for efficient deductive verification of all program variants and applied a specification approach similar to contract overriding [23]. For the detection of feature interactions, we composed specifications with implicit contract refinement and analyzed program variants using ESC [22]. In each work, we proposed one specification approach and focused on verification issues. Our experience was that it is not clear what is the best way to specify feature-oriented programs using DbC. In this work, we propose three further specification approaches and compare all approaches regarding practicability by means of five case studies.

Specification using DbC has been considered for other program modularization techniques than FOP. Bruns et al. [9] and Hähnle et al. [12] discuss DbC for delta-oriented programming (DOP). DOP is an extension of FOP where feature modules (known as delta modules) can also remove methods, fields, and classes. A delta module can add or remove invariants and contracts. Since a feature module only refines existing methods, it is not reasonable to consider the removal of contracts or invariants for FOP.

DbC has been applied to aspect-oriented programming [24,19,1]. The aspect-oriented around advice corresponds roughly to feature-oriented method refinement and thus aspect-oriented programming can be seen as a superset of FOP [4]. Zhao and Rinard [24] proposed Pipa, a DbC specification language for AspectJ. AspectJ programs with Pipa annotations are translated into Java programs with JML annotations to allow programmers to reuse existing JML tools. Lorenz and Skotiniotis [19] analyze advice contracts in terms of runtime assertions. They propose three advice categories with an according runtime assertion strategy each: *agnostic* and *obedient* disallowing contract refinement (similar to contract overriding with equivalent contracts) and *rebellious* allowing contract strengthening (similar to contract overriding with compatible contracts). Agostinho et al. [1] discuss the interaction between classes and aspects while proposing agnostic pieces of advice. All these approaches force programmers to create specification clones, because they do not support contract weakening, which is needed in two of our case studies. Furthermore, the absence of aspects is not considered, while optional features in FOP are essential for software variability.

Most specification approaches for OOP assume behavioral subtyping [17] for subclasses which are the means to reuse code. Dhara and Leavens [11] propose specification inheritance to achieve behavioral subtyping which also is pursued in Eiffel [20] and JML [16]. With consecutive implicit refinement, we transfered the notion of behavioral subtyping to feature-oriented method refinement, but two of five case studies cannot be specified using this approach, as it is too restrictive.

8 Conclusion

In order to increase the reliability of feature-oriented programs, we discussed five approaches to integrate DbC with FOP and evaluated them by means of five case studies. We found that feature-oriented method refinement often requires the refinement of contracts such that the program specification depends on the actual selection of features. In contrast, the refinement of invariants can be avoided in our case studies. Furthermore, we identified the trade-off between expressiveness and complexity: while high expressiveness allows programmers to specify arbitrary feature-oriented programs, the complexity of contracts increases.

Acknowledgment. We thank Fabian Benduhn and anonymous reviewers for comments on earlier drafts of this paper. Apel's work is supported by the German Research Foundation (DFG – AP 206/2, AP 206/4, and LE 912/13). Saake's work is supported by the German Research Foundation (DFG – SA 465/34-1).

References

1. Agostinho, S., Moreira, A., Guerreiro, P.: Contracts for Aspect-Oriented Design. In: Proc. Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT). ACM (2008)
2. Apel, S., Kästner, C.: An Overview of Feature-Oriented Software Development. J. Object Technology (JOT) 8(5), 49–84 (2009)
3. Apel, S., Kästner, C., Größlinger, A., Lengauer, C.: Type Safety for Feature-Oriented Product Lines. Automated Software Engineering (ASE) 17(3), 251–300 (2010)
4. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE Trans. Software Engineering (TSE) 34(2), 162–180 (2008)
5. Apel, S., Scholz, W., Lengauer, C., Kästner, C.: Detecting Dependences and Interactions in Feature-Oriented Design. In: Proc. Int'l Symposium Software Reliability Engineering (ISSRE), pp. 161–170. IEEE (2010)
6. Apel, S., Speidel, H., Wendler, P., von Rhein, A., Beyer, D.: Detection of Feature Interactions using Feature-Aware Verification. In: Proc. Int'l Conf. Automated Software Engineering (ASE), pp. 372–375. IEEE (2011)
7. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering (TSE) 30(6), 355–371 (2004)
8. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems 35(6), 615–708 (2010)

9. Bruns, D., Klebanov, V., Schaefer, I.: Verification of Software Product Lines with Delta-Oriented Slicing. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 61–75. Springer, Heidelberg (2011)
10. Delaware, B., Cook, W., Batory, D.: A Machine-Checked Model of Safe Composition. In: Proc. Workshop Foundations of Aspect-Oriented Languages (FOAL), pp. 31–35. ACM (2009)
11. Dhara, K.K., Leavens, G.T.: Forcing Behavioral Subtyping through Specification Inheritance. In: Proc. Int'l Conf. Software Engineering (ICSE), pp. 258–267. IEEE (1996)
12. Hähnle, R., Schaefer, I.: A Liskov Principle for Delta-oriented Programming. In: Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS), pp. 190–207. Technical Report 2011-26, Department of Informatics, Karlsruhe Institute of Technology (2011)
13. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Comm. ACM 12(10), 576–580 (1969)
14. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do Code Clones Matter? In: Proc. Int'l Conf. Software Engineering (ICSE), pp. 485–495. IEEE (2009)
15. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (1990)
16. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Software Engineering Notes (SEN) 31(3), 1–38 (2006)
17. Liskov, B.H., Wing, J.M.: A Behavioral Notion of Subtyping. Trans. Programming Languages and Systems (TOPLAS) 16(6), 1811–1841 (1994)
18. Liu, J., Basu, S., Lutz, R.: Compositional Model Checking of Software Product Lines using Variation Point Obligations. Automated Software Engineering (ASE) 18(1), 39–76 (2011)
19. Lorenz, D.H., Skotiniotis, T.: Extending Design by Contract for Aspect-Oriented Programming. Computing Research Repository (CoRR), abs/cs/0501070 (2005)
20. Meyer, B.: Applying Design by Contract. Computer 25(10), 40–51 (1992)
21. Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 419–443. Springer, Heidelberg (1997)
22. Scholz, W., Thüm, T., Apel, S., Lengauer, C.: Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report. In: Proc. Int'l Workshop Feature-Oriented Software Development (FOSD), pp. 7:1–7:8. ACM (2011)
23. Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S.: Proof Composition for Deductive Verification of Software Product Lines. In: Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST), pp. 270–277. IEEE (2011)
24. Zhao, J., Rinard, M.: Pipa: A Behavioral Interface Specification Language for AspectJ. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 150–165. Springer, Heidelberg (2003)

Integration Testing of Software Product Lines Using Compositional Symbolic Execution

Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer

Department of Computer Science & Engineering, University of Nebraska-Lincoln, Lincoln, Nebraska, USA

Abstract. Software product lines are families of products defined by feature commonality and variability, with a well-managed asset base. Recent work in testing of software product lines has exploited similarities across development phases to reuse shared assets and reduce test effort. The use of feature dependence graphs has also been employed to reduce testing effort, but little work has focused on code level analysis of dataflow between features. In this paper we present a compositional symbolic execution technique that works in concert with a feature dependence graph to extract the set of possible interaction trees in a product family. It composes these to incrementally and symbolically analyze feature interactions. We experiment with two product lines and determine that our technique can reduce the overall number of interactions that must be considered during testing, and requires less time to run than a traditional symbolic execution technique.

1 Introduction

Software product line (SPL) engineering is a methodology for developing families of software programs through the managed reuse of a common and variable set of assets [18]. Variability at the application level is expressed in terms of features (functional units) that are included or excluded from the individual programs. The result is a set of similar, but unique program instantiations; in a mobile phone product line, features such as the display drivers, messaging capabilities, network support and video can be combined in different ways on top of the core features found in all phones (e.g. phone dial). While uniqueness arises from the different combination of variable features in each program, similarity comes from both the commonality found in all instantiations, as well as from matching subsets of features (i.e. *partial products*) between programs.

Variability, and the ability to generate many products from a core set of features, provides flexibility and enables reuse during development, but causes problems for validation. Although individual features may be validated and tested in multiple programs within the product line, this does not guarantee that specific combinations of features will work properly when composed. Research has shown that some faults – termed *interaction faults* – only occur under specific combinations of features [2,13] and several SPL testing techniques have attempted to account for this. For instance, Bertolino et al. [1] and Geppert et al. [6] propose

J, de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 270–284, 2012.

a specification based technique to concretize a parameterized use case (based on variability), but this is an exhaustive approach that tests each product individually. This is a limitation, since the variability space grows exponentially with the number of features. If there are 4 choices for each of 10 features, then more than one million instantiations of the SPL would need to be tested to cover all possible combinations.

Kim et al. [11] use a dependency analysis to determine which features are relevant for each test within a test suite, reducing the number of products tested per test case. This technique does not consider coverage of the entire feature model, nor does it target the specific interactions; it only reduces the per-test number products. A study by Reisner et al. [20] shows that in some configurable systems – SPLs can be viewed as a type of configurable software system – analysis of control flow can reduce the possible set of configuration options that should be tested together. They do not consider other types of dependencies such as data flow, nor do they apply their approach to product lines. And neither of these studies targets specific interactions for test generation; they only reduce the number of feature combinations that should not be tested together.

In our earlier research [3], we proposed a mapping between the variability space of an SPL and a relational model in order to leverage ideas from *combinatorial interaction testing (CIT)*, a model-based sampling technique that guarantees to test all pairs or t-way combinations of features within the product line. Instead of testing all program instantiations in the example above, we can test all pairs of features with approximately 24 SPL instances, or all triples of features with around 130, using a common CIT generation tool [2]. Since empirical evidence suggests that lower order interactions are responsible for most interaction faults this provides some justification for CIT sampling [13].

While CIT provides a notion of coverage of the variability space, it also suffers from limitations. First, there is an expectation that all possible programs in the product line can be composed. But there may be features or groups of features that are not developed until later phases of the SPL lifetime. Second, since CIT operates at the feature combination level there is no guarantee that testing of an instance will execute the interacting code; this will depend on the quality of the test. Finally, CIT does not consider the direction of the interactions in its model, yet at the code level, interactions may happen between features in different directions. For instance, it is likely that data flows in both directions between a multi-media messaging feature of a phone and its video feature. If we have three features (f_1, f_2, f_3), there are six directed 2-way possible interactions.

When testing a software product line to uncover interactions, we should test from a perspective that avoids these limitations. Uncovering interactions during integration testing – where features are composed as partial products – appears to make sense from a combinatorial sense. We can test only the interactions themselves and combine products in a way that avoids redundancy. Uzuncaova et al. [24] use this idea by reusing a partial product's integration test results to generate a smaller test suite for a larger partial product. And Reis et al. [19] apply integration testing over an SPL at the specification level to avoid

redundantly testing common partial products. Finally, Stricker et al. [23] present the ScenTED-DF methodology which uses dataflow between products to drive integration testing at the model level.

In this paper, we present a new method of analyzing software product lines for test generation. It uses ideas from CIT to drive coverage of feature interaction tuples, reduces the variability space through the use of a code-based dependency analysis, and uses directed symbolic execution to analyze possible feature combinations. The result is a method that generates constraints for all partial products at a lower cost than a full symbolic execution of an SPL code base. We also find, that by counting directed interactions, we have a more precise model of what should be tested. Finally, if we consider the constraints arising from symbolic execution, these can be used to inform a test generation technique to focus on the parts of the system that may have faults. The contributions of this work are: (1) a dataflow informed compositional symbolic integration testing method for SPLs; (2) the first discussion of interaction testing that incorporates directions; and (3) a feasibility study that shows we can reduce the number of interactions to test, and that the compositional technique uses less time than traditional symbolic execution.

2 Background

Software product lines are families of software systems designed for a specific domain, with a managed set of assets and well defined variability model [18]. The products all share some commonality, but are customized by variable elements of the system. Product lines vary in when they are configured. Some may be configured by the developer at build time, others allow changes through recompilation, while some use run-time constructs to change during execution.

A key artifact of a software product line is the feature (or variability) model. This is one differentiator from a general configurable system. There are many formalisms that have been developed to represent these. In this paper we use the Orthogonal Variability Model (OVM) developed by Pohl et al. [18]. In OVM *Variation points (VP)* are shown as triangles and *variants (v)* are shown as rectangles. Variants will map directly to features in this paper. Dependencies are shown as solid lines (mandatory) or dashed (optional). Alternative choices are shown with arcs which are annotated with the the minimum and maximum cardinality of that VP. When there is no annotation, exactly one variant can be selected for the variation point. Additional constraints are allowed between parts of the model in the form of excludes or requires.

2.1 Symbolic Execution

Symbolic execution [12] is a path-sensitive program analysis technique that computes program output values as expressions over symbolic input values and constants. Symbolic execution of the code fragment:

```
y = x;
if (y > 0) then y++;
return y;
```

would use a symbolic value X to denote the value of variable `x` on entry to the fragment. Symbolic execution determines that there are two possible paths (1) when $X > 0$ the value $X + 1$ is returned and (2) when $!(X > 0)$ the value X is returned. The analysis represents the behavior of the fragment as the pairs $(X > 0, RETURN == X + 1)$ and $(!(X > 0), RETURN == X)$. The first element of a pair encodes the conjunction of constraints along an execution path – the *path condition*. The second element defines the values of the locations that are written along the path in terms of the symbolic input variables, e.g. $RETURN == X$ means that the original value for `x` is returned.

The state of a symbolic execution is a triple (l, pc, s) where l, the current location, records the next statement to be executed, pc, the path condition, is the conjunction of branch conditions encoded as constraints along the current execution path, and $s : M \times expr$ is a map that records a symbolic expression for each memory location, M, accessed along the path.

Computation statements, $m_1 = m_2 \odot m_3$, where the $m_i \in M$ and $\odot$ is some operator, when executed symbolically in state (l, pc, s) produce a new state $(l + 1, pc, s')$ where $\forall m \in M - \{m_1\} : s'(m) = s(m)$ and $s(m_1) = s(m_2) \odot s(m_3)$. Branching statements, *if* $m_1 \odot m_2$ *goto* d, when executed symbolically in state (l, pc, s) branch the symbolic execution to *two* new states $(d, pc \wedge (s(m_1) \odot s(m_2)), s)$ and $(l + 1, pc \wedge \neg(s(m_1) \odot s(m_2)), s)$ corresponding to the "true" and "false" evaluation of the branch condition, respectively.

An automated decision procedure is used to check the satisfiability of the updated path conditions and, when a path condition is found to be unsatisfiable, symbolic execution along that path halts. Decision procedures for a range of theories used to express path conditions, such as, linear arithmetic, arrays, and bit-vectors are available, e.g., Z3 [5].

2.2 Symbolic Method Summary

Several researchers [8,17] have explored the use of method summarization in symbolic execution. In [8] summarization is used as a mechanism for optimizing the performance of symbolic execution whereas [17] explores the use of summarization as a means of abstracting program behavior to avoid symbolic execution. We adopt the definition of method summary in [17], but we forgo their use of over-approximation.

The building block for a method summary is the representation of a single execution path through method, m, encoded as the pair (pc, w). This pair provides information about the externally visible state of the program that is relevant to an execution of m at the point where m returns to its caller. As described above, the pc encodes the path condition and w is the projection of s onto the set of memory locations that are written along the executed path. We can view w a conjunction of equality constraints between names of memory locations and symbolic expressions or, equivalently, as a map from locations to expressions.

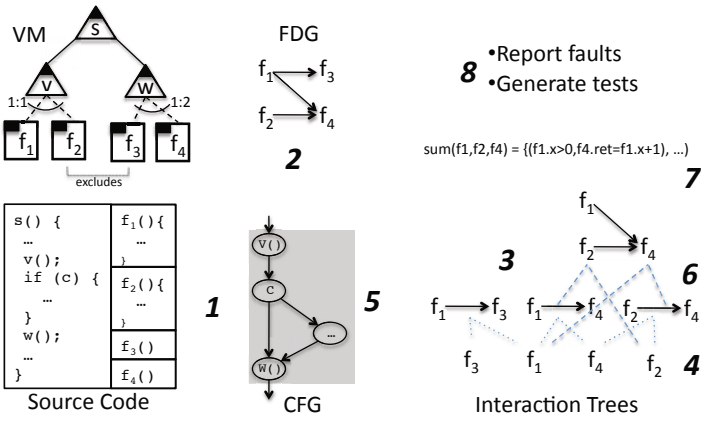


Fig. 1. Conceptual Overview of Compositional SPL Analysis

Definition 1 (Symbolic Summary [17]). *A symbolic summary, for a method m, is a set pairs $m_{sum} : \mathcal{P}(PC \times S)$ where*

$$\forall (pc, w) \in m_{sum} : \forall (pc', w') \in m_{sum} - \{(pc, w)\} : pc \wedge pc' \textit{ is unsatisfiable.}$$

Unfortunately, it is not always possible to calculate a summary that completely accounts for the behavior of all methods. For example, methods that iterate over input data structures that are unconstrained cannot be analyzed effectively – since the length of paths are not known. We address this using the standard technique of bounding the length of paths that are analyzed.

3 Dependence-Driven Compositional Analysis

Our technique exploits an SPL's variability model and the inter-dependence of feature implementations to reduce the cost of applying symbolic execution to reason about feature interactions. Figure 1 provides a conceptual overview.

As explained in Section 1 an SPL is comprised of a source code base and an OVM. The OVM and its constraints (e.g., the excludes between f_2 and f_3) defines the set of features that may be present in an instance of the SPL.

Our technique begins (step are denoted by large bold italic numerals in the figure) by applying standard control flow and dependence analyses on the code base. The former results in a control flow graph (CFG) and the latter results in a program dependence graph (PDG). In step 2, the PDG is analyzed to calculate a feature dependence graph (FDG) which reflects inter-feature dependences. The edges of the FDG are pruned to be consistent with the OVM, e.g., the edge from f_2 to f_3 is not present.

Step 3 involves the calculation, from the FDG, of the hierarchy of all k-way feature interaction trees. The structure of this hierarchy reflects how lower-order interactions can be composed to create higher-order interactions. For instance,

how the interaction among f_1, f_2, and f_4 can be constructed by combining f_1 with an existing interaction for f_2 and f_4.

The interaction tree hierarchy is used to guide the calculation of symbolic summaries for all interaction trees in a compositional fashion. This begins, in Step 4, by applying symbolic execution to the source code of the individual features in isolation. When composing two existing summaries, for example f_1 and f_3, to create a 2-way interaction tree, a summary of the behavior of the common SPL code which leads between those summaries must be calculated. Step 5 achieves this by locating the calls to the features in the CFG and calculating a chop [21] – shown as the shaded figure in the CFG – the edges of the chop are used to guide a customized symbolic execution to produce an edge summary. In step 6, a pair of existing lower-order interaction summaries and the edge summary are composed to produce a higher-order summary – such a summary is illustrated at point 7 in the figure.

In step 8, summaries can be exploited to detect faults, via comparison to fault oracles, or to generate tests by solving the constraints generated by symbolic execution and composition. We describe the major elements next.

3.1 Relating SPL Models to Implementations

An SPL implementation can be partitioned into *regions* of code that implement each feature; the remaining code implements the common functionality shared by all SPL instances. There are many implementation mechanisms for realizing variability in a code base [10]. Our methodology can target these by adapting the summary computation for Step 4 and feature dependence graph construction for Step 2, but for simplicity it suffices to view features as methods where common code makes calls on those methods.

In the remainder of this section, we assume the existence of a mapping from in the OVM to methods in a code base; we use the name of a feature to denote the method when no confusion will arise. Features can be called from multiple points in the common code, but to simplify the presentation of our technique, we assume each feature is called from a single call site.

Given a pair of features, f_1 and f_2, where the call to f_2 is reachable in the CFG from the call to f_1, their *common region* is the source code chop [21] arising when the calls are used as the chop criterion. This chop is a single-entry single-exit sub-graph of the program control flow graph (CFG) where the entry node is the call to f_1 and the exit node is the call to f_2. The CFG paths within the chop overapproximate the set of feasible program executions that can lead from the return of f_1 to the call to f_2. These chops play an important role in accounting for the composite behavior of features as mediated by common code.

3.2 Calculating Feature Interactions

We leverage the concept of program dependences, and the PDG [16], to determine inter-feature dependences. A PDG is a directed graph, (S, E_{PDG}), whose vertices are program statements, S, and $(s_i, s_j) \in E_{PDG}$ if s_i defines the value

Algorithm 1. Computing k-way Interaction Trees

```
1: interactionTrees(k, (F, E))
2:   T := ∅
3:   for (f_i, f_j) ∈ E
4:     T ∪= tree(f_i, f_j)
5:   for i = 3 to k + 1
6:     for t_{i-1} ∈ T ∧ |t_{i-1}| = i − 1
7:       for v ∈ F − v(t_{i-1})
8:         if (root(t_{i-1}), v) ∈ E ∧ consistent(v(t_{i-1} ∪ v)) then
9:           T ∪= tree(t_{i-1}, (root(t_{i-1}), v))
10:        else
11:          for (v, v') ∈ E ∧ v' ∈ v(t_{i-1})
12:            if consistent(v(t_{i-1} ∪ v)) then T ∪= tree(t_{i-1}, (v, v'))
13:        endif
14:  return T
15: end interactionTrees()
```

of a location that is subsequently read at s_j. A feature dependence graph (FDG) is an abstraction of the PDG for an SPL implementation.

Definition 2 (Feature Dependence Graph). *Given a PDG for an SPL, (S, E_{PDG}), the FDG, (F, E_{FDG}), is a directed graph whose vertices are features, F, and $(f_i, f_j) \in E_{FDG}$ iff $\exists s_i, s_j \in S : s_i \in S(f_i) \wedge s_j \in S(f_j) \wedge (s_i, s_j) \in E_{PDG}$ where $S(f)$ is the set of statements in feature f.*

We capture the interaction among features by defining a tree that is embedded in the FDG. The intuition is that the root is the sink of a set of feature dependence edges. The output values of that root feature reflect the final interaction effects, and are defined in terms of the input values of the features that form the leaves of the tree.

Definition 3 (Interaction Tree). *Given an FDG, (F, E_{FDG}), a k-way interaction tree is an acyclic, connected, simple subgraph, (F', E'), where $F' \subseteq F$, $E' \subseteq E_{FDG}$, $|F'| = k$, and where $\exists r \in F' : \forall v \in F' : r \in v.(E')^*$. We call the common reachable vertex the root of the interaction tree.*

The set of all k-way interaction trees for an SPL can be constructed as shown in Algorithm 1. The algorithm uses a constructor *tree*() which, optionally, takes an existing tree and adds edges to it expanding the set of vertices as appropriate. For a tree, t, the set of vertices is $v(t)$ and the root is $root(t)$. Before adding a tree, the set of features in the tree must be checked to ensure they are consistent with the OVM; this is done using the predicate *consistent*().

The algorithm accepts k and an FDG and returns the set of k-way interactions. It builds the set of interactions incrementally. For an i-way interaction, it extends an $i-1$-way interaction by adding a single additional vertex and an edge. While other strategies for building interaction trees are possible, this approach has the advantage of efficiency and simplicity. Based on our case studies, reported in Section 4, this approach is sufficient to enable significant improvement over more standard analyses of an SPL code base.

Interaction trees can be organized hierarchically based on their structure.

Definition 4 (Interaction Hierarchy). *Given a k-way interaction tree, $t_k = (F, E)$, where $k > 1$, we can define a pair of interaction trees $t_i = (F_i, E_i)$ and $t_j = (F_j, E_j)$, such that $F_i \cap F_j = \emptyset$, $|F_i| + |F_j| = k$, and $\exists (f_i, f_j) \in E$. We say that t_k is the parent of t_i and t_j and, that t_i and t_j are the children of t_k.*

The base case of the hierarchy, where $k = 1$, is simply each feature in isolation. There are many ways to construct such an interaction hierarchy, since for any given k-way interaction tree cutting a single edge partitions the tree into two children. As discussed below, the hierarchy resulting from Algorithm 1 enjoys a structure that can be exploited in generating summaries of interaction pattern behavior. The parent (child) relationships among interaction trees can be recorded at the point where the *tree*() constructor calls are made in Algorithm 1.

3.3 Composing Feature Summaries

Our goal is to analyze program paths that span sets of features in an SPL to support fault detection and test generation. Our approach to feature summarization involves two distinct phases: (1) the application of bounded symbolic execution to feature implementations in isolation to produce feature summaries, and (2) the matching and combination of feature summaries to produce summaries of the behavior of interaction patterns.

Phase (1) is performed by applying traditional symbolic execution where the length of the longest branch sequence is bounded to d – the depth. For each feature, f, this results in a summary, f_{sum}, as defined in Section 2.

When performing symbolic execution of f there are three possible outcomes: (a) a complete execution of f which returns normally as analyzed within d branches, (b) an exception, including assertion violations, is detected before d branches are explored, and (c) the depth bound is reached. In our work, we only accumulate the outcomes falling into (a) into f_{sum}.

Case (b) is interesting, because it *may* indicate a fault in feature f. The isolated symbolic execution of f allows for any possible state on entry to the feature, however, it is possible that a detected exception is infeasible in the context of a system execution. In future work, we will preserving results from case (b) and attempt to determine their feasibility when composed in interaction patterns with other features – this would reduce and, when interaction patterns are sufficiently large, eliminate false reports of exceptions.

For phase (2) we exploit the structure of the interaction hierarchy resulting from the application of Algorithm 1 to generate a summary for a k-way interaction. As discussed above, such an interaction has (potentially several) pairs of children. It suffices to select any of those pairs.

Within each pair there is a $k-1$-way interaction, i, which we assume has a summary $i_{sum} = (pc_i, w_i)$, and single feature, f, summarized as $f_{sum} = (pc_f, w_f)$, which is connected by a single edge connected to either $root(i)$ or one of i's leaves, l. To compose i_{sum} and f_{sum} we must characterize the behavior of the FDG edge.

The existence of an edge (f, f') means that there is a common region beginning at the return from f and ending at the call to f'. Calculating the chop that

Algorithm 2. Edge Summary (left) and Composing Summaries (right)

```
1: eSum(E, l, e, pc, s, w, d)
2: if |pc| > 0
3:   if branch(l)
4:     l_t := target(l, true)
5:     if SAT(cond(l, s)) ∧ (l, l_t) ∈ E
6:       eSum(E, l_t, e, pc ∧ cond(l, s), s, w, d − 1)
7:     l_f := target(l, false)
8:     if SAT(¬cond(l, s)) ∧ (l, l_f) ∈ E
9:       eSum(E, l_f, e, pc ∧ ¬cond(l, s), s, w, d − 1)
10:  else
11:    if l = e
12:      sum ∪ = (pc, π(s, w))
13:    else
14:      s := update(s, l)
15:      w ∪ = write(l)
16:      eSum(E, succ(l), e, pc, s, w, d)
17:    endif
18: endif
19: if pc = true return sum
20: end eSum()
```

```
1: cSum(s, s')
2: s_c := ∅
3: for (pc, w) ∈ s
4:   for (pc', w') ∈ s'
5:     eq := true
6:     for l ∈ read(pc')
7:       if ∃l ∈ dom(w)
8:         eq := eq ∧ input(s', l) = w(l)
9:     if SAT(pc ∧ eq ∧ pc')
10:      for l ∈ dom(w')
11:        if ∃l ∈ dom(w)
12:          w := w − (l, _)
13:      endfor
14:      s_c ∪ = (pc ∧ eq ∧ pc', w ∧ w')
15:    endif
16:  endfor
17: end cSum()
```

circumscribes the CFG for this region allows us to label branch outcomes that lie within the chop and to direct the symbolic execution along paths from f that reach f'.

Algorithm 2(left) defines this approach to calculating edge summaries. It consists of a customized depth-bounded symbolic execution that only explores a branch if that branch lies within the chop for the common region. The algorithm makes use of several helper functions. Functions determine whether an instruction is a branch, $branch()$, the target of a branch, $target()$, and the symbolic expression for a branch given a symbolic state, $cond()$. Functions to calculate the successor of an instruction, $succ()$, the set of locations written by an instruction, $write()$, and updating the symbolic state based on an instruction, $update()$, are also used. The $SAT()$ predicate determines whether a logical formula is satisfiable. Finally, the $\pi()$ function projects a symbolic state onto a set of locations.

$eSum(E_{chop}, succ(f), f', true, \emptyset, \emptyset, d)$ returns the symbolic summary for edge (f, f') where the parameters are as follows. E_{chop} is the set of edges in the CFG chop bounded by the return of f and the call to f', $succ(f)$ is the location at which initiate symbolic execution and f' is the call that terminates symbolic execution. $true$ is the initial path condition. The next two parameters are the initial symbolic state and the set of locations written on the path – both are initially empty. d is the bound on the length of the path condition that will be explored in producing the summary.

To produce a symbolic summary for the k-way interaction, we now compose i_{sum}, f_{sum}, and the edge summary computed by $eSum()$. There are two cases to consider. If the feature, f', is connected to $root(i)$ with an edge, $(root(i), f')$ we compose summaries in the following order: $i_{sum}, (root(i), f')_{sum}, f'_{sum}$. If the feature, f', is connected to a leaf of i, l_i, with an edge, (f', l_i) we compose summaries in the following order: $f'_{sum}, (f', l_i)_{sum}, i_{sum}$.

Order matters in composing summaries because the set of written locations of two summaries may overlap and simply conjoining the equality constraints on the values at such locations will likely result in constraints that are unsatisfiable. We keep only last write of locations in a composed summary to honor the sequencing of writes and reads of locations that arise due to the order of composition.

Consider the composition of summary s with summary s', in that order. Let $(pc, w) \in s$ and $(pc', w') \in s'$ be two elements of those summaries. The concern is that $dom(w) \cap dom(w') \neq \emptyset$, where $dom()$ extracts the set of locations used to index into a map. Our goal is to eliminate the constraints in w on locations in $dom(w) \cap dom(w')$. In general, pc' will read the value of at least one location, l, and that location may have been written by the preceding summary. In such a case, the input value referenced in pc' should be equated to $w(l)$. Algorithm 2(right) composes two summaries taking care of these two issues.

In our approach, the generation of a symbolic summary produces "fresh" symbolic variables to name the values of inputs. A map, $input()$, records the relationship between input locations and those variables. We write $input(s, l)$ to denote a summary s and a location l to access the symbolic variable. For a given path condition, pc, a call to $read(pc)$ returns the set of locations referenced in the constraint – it does this by mapping back from symbolic variables to the associated input locations. We rely on these utility functions in Algorithm 2(right).

The algorithm considers all pairs of summary elements and generates, through the analysis of the locations that are written by the first summary and read by the second summary, a set of equality constraints that encode the path condition of the second summary element in terms of the inputs of the first. The pair of path conditions along with these equality constraints are checked for satisfiability. If they are satisfiable, then the cumulative write effects of the summary composition are constructed. All of the writes of the later summary are enforced and the writes in the first that are shadowed by the second are eliminated – which eliminates the possibility of false inconsistency.

3.4 Complexity and Optimization of Summary Composition

From studying the Algorithm 2 it is apparent that the worst-case cost of constructing all summaries up to k-way summaries is exponential in k. This is due to the quadratic nature of the composition algorithm.

In practice we see quite a different story, in large part because we have optimized summary composition significantly. First, when we can determine that a pair of elements from a summary that might potentially match we ensure that for any shared features the summaries agree on the values for the elements of those summaries; this can be achieved through a string comparison of the summary constraints which is much less expensive than calling the SAT solver. Second, we can efficiently scan for constraints in one summary that are not involved in another summary and those can be eliminated since they were already found to be satisfiable in previous summary analyses.

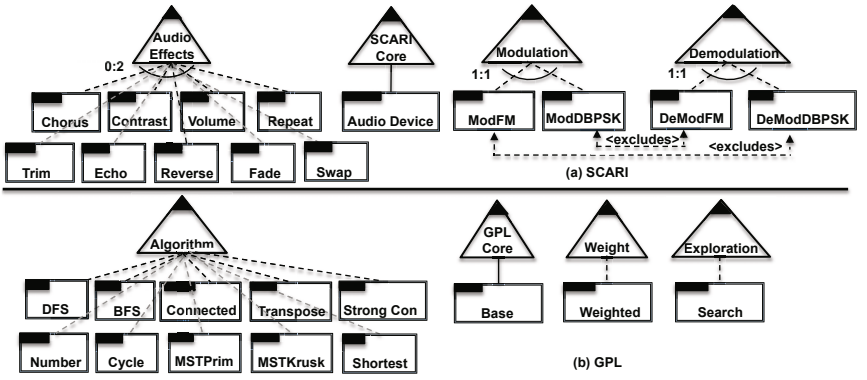


Fig. 2. Feature Models for (top) SCARI and (bottom) GPL

4 Case Study

We have designed a case study for evaluating the feasibility of our approach that ask the following two research questions. (**RQ1**): What is the reduction from our dependency analysis on the number of interactions that should be tested in an SPL? (**RQ2**): What is the difference in time between using our compositional symbolic technique versus a traditional directed technique?

4.1 Objects of Analysis

We selected two software product lines. The first SPL is based on the implementation of the Software Communication Architecture-Reference Implementation (SCARI-Open v2.2) [4] and the second is a graph product line, GPL [11,14] used in several other papers on SPL testing.

The first product line, SCARI, was constructed by us as follows. First we began with the Java implementation of the framework. We removed the non-essential part of the product line (e.g. logging, product installation and launching) and features that required CORBA Libraries to execute. We kept the core mandatory feature, Audio Device, and transformed four features that were written in C (ModFM, DemodFM, Chorus and Echo), into Java. We then added 9 other features which we translated from C to Java from the GNU Open Source Radio [7] and the Sound Exchange (SoX), site [22]. Table 1 shows the origin of each feature and the number of summaries for each. We used the example function for assembling features, to write a configuration program that composes the features together into products. The feature model is shown in Figure 2(a).

The graph product line (GPL) [14] has been used for various studies on SPLs. We start with the version found in the implementation site for [11]. To fit our prototype tool, we re-factored some code so that every feature is contained in a method. We removed several features because either we could not find a method in the source code or because JPF would not run. We made the method Prog

Table 1. SCARI Size by Feature

Features	Origin	LOC	No. Summaries
Chorus	[4]	30	6
Contrast	[22]	14	5
Volume	[22]	47	5
Repeat	[22]	12	3
Trim	[22]	11	6
Echo	[4]	31	5
Reverse	[22]	14	4
Fade	[22]	9	4
Swap	[22]	27	4
AudioDevice	[4]	13	3
ModFM	[4]	19	4
ModDBPSK	[7]	6	2
DemodFM	[4]	18	4
DemodDBPSK	[7]	6	3
Total		**257**	**58**

Table 2. GPL Size by Feature

Features	LOC	No. Summaries
Base	85	56
Weighted	32	148
Search	35	19
DFS	23	41
BFS	23	6
Connected	4	8
Transpose	27	3
StronglyConnected	19	9
Number	2	2
Cycle	40	19
MSTPrim	92	4
MSTKruskal	106	3
Shortest	102	3
Total	**590**	**321**

our main entry point for the program. We did not include any constraints for simplicity. Figure 2(b) shows the resulting feature model and Table 2 shows the number of lines of code and the number of summaries by feature.

4.2 Method and Metrics

Experiments are run on an AMD Linux computing cluster running CentOS 5.3 with 128GB memory per node. We use Java Pathfinder (JPF) [15] to perform SE with the Choco solver for SCARI and CVC3BitVector for GPL. We adapt the information flow analysis (IFA) package [9] in Soot [25] for our FDG. In SCARI we use the configuration program for a starting point of analysis. In GPL we use the Prog program, which is an under-approximation of the FDG.

For RQ1 we compute the number of possible interactions (directed and undirected) at increasing values for k, obtained directly from the feature model. We compare this with the number that we get from the interaction trees. For RQ2, we compare the time that is required to execute the two symbolic techniques on all of the trees for increasing values of k. We compare incremental SE (**IncComp**) and a full direct SE (**DirectSE**). We set the depth for SE at 20 for IncComp and allow DirectSE k-times that depth since it works on the full partial-product each time, while IncComp composes k summaries each computed at depth 20. DirectSE does not use summaries, but in the SPLs we studied there is no opportunity for summary reuse *within* the analysis of a partial product – our technique reuses summaries *across* partial products.

4.3 Results

RQ1. Table 3 compares the number of interactions obtained from just the OVM with the number of interaction trees obtained through our dependency analysis. We present k from 2 to 5. The column labelled UI is the number of interactions calculated from all k-way combinations of features. In SCARI there are only three true points of variation given the model and constraints, therefore we see the same number of interactions for $k = 3$ and 4. For $k = 5$, we have fewer interactions since there are 5 unique 4-way feature combinations in a single product with 5 features, but only a single 5-way combination. The DI column

Table 3. Reduction for Undirected (U) and Directed (D) Interactions (I)

Subject	k	UI	DI	Feasible UI	Feasible DI	UI Reduction	DI Reduction
SCARI	2	188	376	85	85	54.8%	77.4%
	3	532	3192	92	92	82.7%	97.1%
	4	532	12768	162	162	69.5%	98.7%
	5	164	19680	144	144	12.2%	99.3%
GPL	2	288	576	21	27	92.7%	95.3%
	3	2024	12144	29	84	98.6%	99.3%
	4	9680	232320	31	260	99.7%	99.9%
	5	33264	3991680	20	525	99.9%	100.0%

represents the number of directed interactions or all permutations ($k! \times UI$). The next two columns are feasible interactions obtained from the interaction trees. Feasible UI, removes direction, counting all trees with the same features as equivalent. Feasible DI is the full tree count. The last two columns give the percent reduction. For the undirected interactions we see a reduction of between 12.2% and 99.9% across subjects and values of k, and the reduction is more dramatic in GPL (92.7%-99.9%). If we consider the directed interactions, which would be needed for test generation, there is a reduction ranging from 77.4% to 100%. In terms of absolute values we see a reduction in GPL from over 3 million directed interactions at $k = 5$, down to 525, an order 4 magnitude of difference. DIs are useful to detect more behaviors. For example, given a one-second-sound file, trim→repeat removes 1-second-sound and generates an empty file; repeat→trim repeats the sound once and outputs a 1-second-sound file.

RQ2. Table 4 compares the performance of DirectSE and IncComp in terms of time (in seconds). It lists the number of directed (D) and undirected (U) interactions (I) for each k, that are feasible based on the interaction trees. Some features in the feature models may have more than one method. In RQ1 based on the OVM we reported interactions only at the feature level. However in this table, we consider all methods within a feature and give a more precise count of the interactions; we list all of the interactions (both directed and undirected) between features. The next two columns present time. For Direct SE we re-start the process for each k, but for the IncComp technique we use cumulative times because we must first complete $k - 1$ to compute k. Although both techniques use the same time for single feature summaries, they begin to diverge quickly. DirectSE is 3 times slower for $k = 5$ on SCARI, and 4 times slower on GPL. Within SCARI we see no more than a 3 second increase to compute $k + 1$ from k (compared to 14-35 seconds for DirectSE) and in GPL we see at most 750 (12 mins). For DirectSE it requires as long as 3160 (about 1 hour).

The last column of this table shows how many feasible paths were sent to the SAT solver (SAT). We saw (but don't report) a similar number for DirectSE which we attribute to our depth bounding heuristic. The number for SMT represents the total number of possible calls that were made to the SAT solver. However, we did not send all possible calls, because our matching heuristic culled out a number which we show as Avoided Calls.

Table 4. Time Comparisons for SCARI and GPL

Subject	k	Feasible UI	Feasible DI	DirectSE	IncComp	
				Time (sec)	Time (sec)	SAT/SMT, Avoided Calls
SCARI	1	14	14	6.75	6.75	58
	2	85	85	14.48	9.63	430/1780, 0
	3	92	92	17.67	10.06	844/2226, 1587
	4	162	162	36.09	10.93	1505/2909, 3442
	5	144	144	35.87	11.70	2075/3523, 5696
GPL	1	49	49	41.77	41.77	321
	2	60	76	67.25	56.28	663/985, 0
	3	81	310	184.76	82.00	1441/1901, 1809
	4	82	1725	727.34	216.63	5814/7342, 5396
	5	52	8135	3887.23	965.92	27444/34147, 19743

5 Conclusions and Future Work

In this paper we have presented a compositional symbolic execution technique for integration testing of software product lines. Using interaction trees to guide incremental summary composition we can efficiently account for all possible interactions between features. We consider interactions as directed which gives us a more precise notion of interaction than previous research. In a feasibility study we have shown that we can (1) reduce the number of interactions to be tested by a factor of between 12.2% and 99.9% over an uninformed model, and (2) reduce the time taken to perform symbolic execution by as much as factor of 4 over a directed symbolic execution technique. Another advantage of this technique is that since our results and costs are cumulative, we can keep increasing k as time allows, making our testing stronger, without any extraneous work along the way.

As future work we plan to exploit the information gained from our analysis to perform directed test generation. By using the complete paths we can generate test cases from the constraints that can be used with more refined oracles. For paths which reach the depth bound, we plan to explore ways to characterize these partial paths to guide other forms of testing, such as random testing, to explore the behavior which is otherwise unknown.

Acknowledgements. This work is supported in part by the National Science Foundation through awards CCF-0747009 and CCF-0915526, the Air Force Office of Scientific Research through awards FA9550-09-1-0129 and FA9550-10-1-0406, and the National Aeronautics and Space Administration under grant number NNX08AV20A.

References

1. Bertolino, A., Gnesi, S.: PLUTO: A Test Methodology for Product Families. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 181–197. Springer, Heidelberg (2004)
2. Cohen, M.B., Colbourn, C.J., Gibbons, P.B., Mugridge, W.B.: Constructing test suites for interaction testing. In: Proc. of the Intl. Conf. on Soft. Eng., pp. 38–48 (May 2003)
3. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: Proc. of the Workshop on the Role of Arch. for Test. and Anal., pp. 53–63 (July 2006)

4. Communication Research Center Canada, http://www.crc.gc.ca/en/html/crc/home/research/satcom/rars/sdr/products/scari_open/scari_open
5. 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)
6. Geppert, B., Li, J., Rößler, F., Weiss, D.M.: Towards Generating Acceptance Tests for Product Lines. In: Dannenberg, R.B., Krueger, C. (eds.) ICSR 2004. LNCS, vol. 3107, pp. 35–48. Springer, Heidelberg (2004)
7. GNU Radio, http://gnuradio.org/redmine/wiki/gnuradio
8. Godefroid, P.: Compositional dynamic test generation. In: Proc. of the ACM Symposium on Principles of Programming Languages, pp. 47–54 (2007)
9. Halpert, R.L.: Static lock allocation. Master's thesis, McGill University(April 2008)
10. Jaring, M., Bosch, J.: Expressing product diversification – categorizing and classifying variability in software product family engineering. Intl. Journal of Soft. Eng. and Knowledge Eng. 14(5), 449–470 (2004)
11. Kim, C.H.P., Batory, D., Khurshid, S.: Reducing combinatorics in testing product lines. In: Asp. Orient. Soft. Dev., AOSD (2011)
12. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
13. Kuhn, D., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Trans. on Soft. Eng. 30(6), 418–421 (2004)
14. Lopez-Herrejon, R.E., Batory, D.: A Standard Problem for Evaluating Product-Line Methodologies. In: Dannenberg, R.B. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001)
15. NASA Ames. Java Pathfinder (2011), http://babelfish.arc.nasa.gov/trac/jpf
16. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. In: Proc. of the Soft. Eng. Symp. on Practical Soft. Develop. Envs., pp. 177–184 (1984)
17. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Intl. Symp. on Foun. of Soft. Eng., pp. 226–237 (2008)
18. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
19. Reis, S., Metzger, A., Pohl, K.: Integration Testing in Software Product Line Engineering: A Model-Based Technique. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 321–335. Springer, Heidelberg (2007)
20. Reisner, E., Song, C., Ma, K.-K., Foster, J.S., Porter, A.: Using symbolic evaluation to understand behavior in configurable software systems. In: Intl. Conf. on Soft. Eng., pp. 445–454 (May 2010)
21. Reps, T., Rosay, G.: Precise interprocedural chopping. In: Proc. of the ACM Symposium on Foundations of Soft. Eng., pp. 41–52 (1995)
22. Sox. Sound Exchange (2011), http://sox.sourceforge.net/
23. Stricker, V., Metzger, A., Pohl, K.: Avoiding Redundant Testing in Application Engineering. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 226–240. Springer, Heidelberg (2010)
24. Uzuncaova, E., Garcia, D., Khurshid, S., Batory, D.: Testing software product lines using incremental test generation. In: Intl. Symp. on Soft. Reliab. Eng, pp. 249–258 (2008)
25. Vallée-Rai, R., Gagnon, E.M., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)

Combining Related Products into Product Lines

Julia Rubin[1,2] and Marsha Chechik[1]

[1] University of Toronto, Canada
[2] IBM Research in Haifa, Israel
mjulia@il.ibm.com, chechik@cs.toronto.edu

Abstract. We address the problem of refactoring existing, closely related products into product line representations. Our approach is based on *comparing* and *matching* artifacts of these existing products and *merging* those deemed similar while explicating those that vary. Our work focuses on formal specification of a product line refactoring operator called *merge-in* that puts individual products together into product lines. We state sufficient conditions of model *compare*, *match* and *merge* operators that allow application of *merge-in*. Based on these, we formally prove correctness of the *merge-in* operator. We also demonstrate its operation on a small but realistic example.

1 Introduction

Numerous companies develop and maintain families of related software products. These products share a common, managed set of features that satisfy the specific needs of a particular market segment and are referred to as software product lines (SPLs) [4]. SPLs often emerge from experiences in successfully addressed markets with similar, yet not identical needs. It is difficult to foresee these needs a priori and hence to structure and manage the SPL development upfront [11]. As a result, SPLs are usually developed in an ad-hoc manner, using available software engineering practices such as duplication (the "clone-and-own" paradigm where artifacts are copied and modified to fit the new purpose), inheritance, source control branching and more. However, these software engineering practices do not scale well to product line development, resulting in massive rework, increased time-to-market and lost opportunities.

Software Product Line Engineering (SPLE) is a software engineering discipline aiming to provide methods for dealing with the complexity of SPL development [4,18,5]. SPLE practices promote *systematic software reuse* by identifying and managing *commonalities* – artifacts that are part of each product of the product line, and *variabilities* – artifacts that are specific to one or more (but not all) individual products across the whole product portfolio. Commonalities and variabilities are controlled by *feature models* [7] (a.k.a. *variability models*) which specify program functionality units and relationships between them. A product of the product line is identified by a unique and legal combination of features, and vice versa.

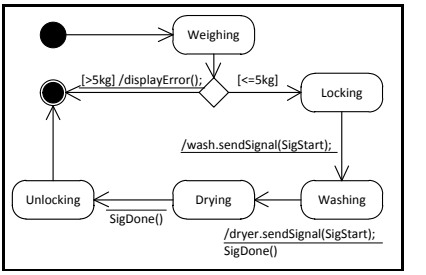
SPLE approaches can be divided into two categories: *compositional*, which implement product features as distinct fragments and allow generating specific product by composing a set of fragments, and *annotative*, which assume that there is one "maximal" product in which annotations indicate the product feature that a particular

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 285–300, 2012.

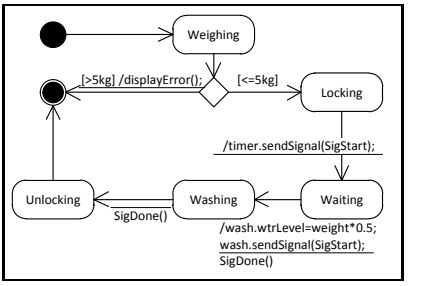
fragment realizes [8,3]. A specific product is obtained by removing fragments corresponding to discarded features. We follow the annotative approach here.

A number of works, e.g., [18,5], promote the use of annotative SPLE practices for *model-driven development* of complex systems. They are built upon the idea of explicating and parameterizing *variable* model elements by features. The parameterized elements are included in a product only if their corresponding features are selected, allowing coherent and uniform treatment of the product portfolio, a reduced number of duplications across products, better understandability and reduced maintenance effort, e.g., because modifications in the common parts can be performed only once.

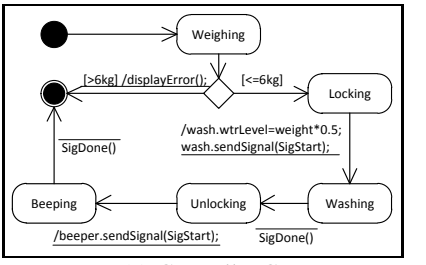
Example. Consider three fragments of UML statechart controllers depicted in Fig. 1. These models were inspired by a real-life SPL developed by a partner (since partner-specific details are confidential, we move the problem into a familiar domain of washing machines). `Controller` A in Fig. 1(a) weighs the laundry and displays an error message if the weight is more than 5 kg. Otherwise, it locks the washing machine and sends a signal to the `wash` engine, responsible for performing the washing cycle. When washing is done, the `Controller` signals the `dryer` to perform the drying cycle, after which it proceeds to unlock the washing machine and finish. `Controller` B in Fig. 1(b) differs from the one in Fig. 1(a) by using the `timer` component to delay the wash cycle and by setting the `wtrLevel` attribute of the `wash` engine to the desired water level based on the weight of the laundry. This model also lacks the dryer capability. Similarly to the one in Fig. 1(b), `Controller` C in Fig. 1(c) uses the `wtrLevel` attribute to set the desired water level of the `wash` engine based on the laundry weight. However, it allows laundry weights up to 6 kg. It also lacks both the dryer and the timer capabilities but initiates an acoustic notification at the end of the program by invoking the `beeper` engine.



(a) Controller A.



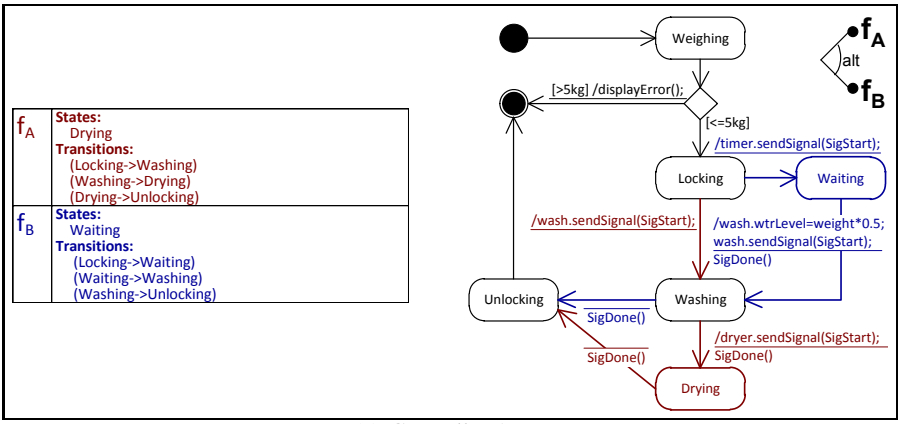
(b) Controller B.



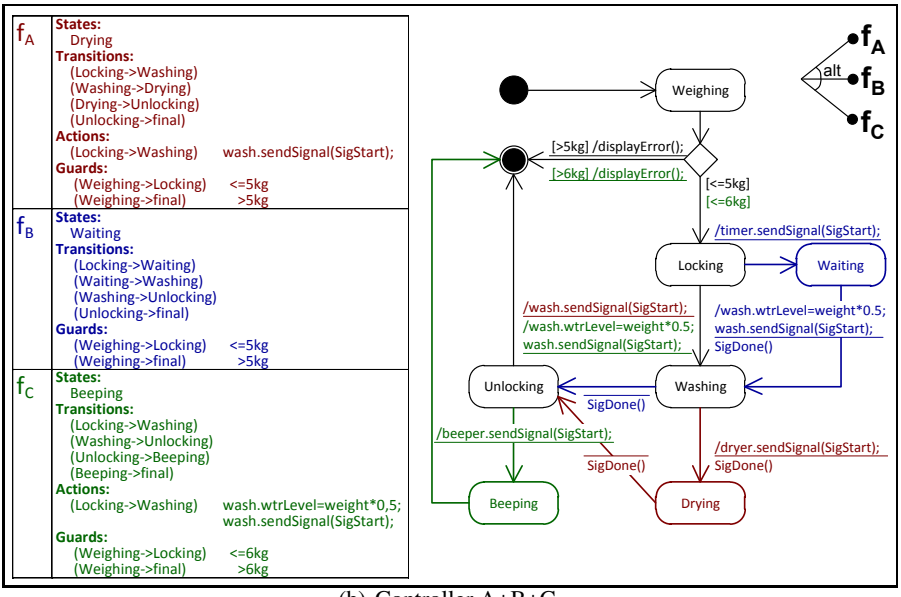
(c) Controller C.

Fig. 1. Washing Machine Controllers

These controllers have a large degree of similarity and can be refactored into SPLE representations where duplications are eliminated and variabilities are explicated. An example of a possible refactoring is given in Fig. 2(b), where the `Drying`, `Waiting`



(a) Controller A+B.



(b) Controller A+B+C.

Fig. 2. Possible Refactorings of the Washing Machine Controllers in Fig. 1

and `Beeping` states and their corresponding transitions are annotated by a set of features depicted in the right upper part of the figure. The refactored product line in our example encapsulates only the original input products, thus we have just three alternative features representing these products – f_A, f_B and f_C. The set of annotations specifies elements to be included given a particular feature selection. E.g., selecting f_A filters out all elements not annotated with that feature, which results in `Controller` A in Fig. 1(a). Likewise, selecting feature f_B (f_C) results in `Controller` B (`Controller` C) in Fig. 1(b) (Fig. 1(c)).

The annotations themselves are shown in a table on the left-hand side of the figure (see "State" and "Transitions" entries in the table). While the transition between `Locking` and `Washing` states exists in both `Controller` A and C (Fig. 1(a,c)), the corresponding actions on the transition are different and thus are also annotated by features in the combined version (see "Actions" entry in the table). Likewise, laundry weight guards on the transitions exiting the `Weighing` state are annotated by the corresponding features as well (see "Guards" entry in the table).

Product Line Refactoring Framework. Despite the benefits of applying SPLE practices which include improved time-to-market and quality, reduced portfolio size, engineering costs and more [4], it is impractical to assume that existing (legacy) product line systems can be abandoned altogether for creating new ones that take advantage of the SPLE reuse techniques. Thus, a transition process which involves identification and extraction of common and variable artifacts together with *variability models* that control them, becomes a necessity [12,1].

In our work, we propose a generic framework for mining legacy product lines and automating their refactoring to contemporary feature-oriented SPLE approaches, initially suggested in [19]. We consider those refactorings that just include the set of existing products rather than allowing novel feature combinations (e.g., a product with both the timer and the beeper capabilities). Our approach is based on *comparing* elements of the input products to each other (by calculating a weighted similarity of their corresponding sub-elements), *matching* those whose similarity is above a preset threshold and *merging* these together.

Our refactoring framework is applicable to a variety of model types, such as UML, EMF or Matlab/Simulink, and to different *compare*, *match* and *merge* operators. In this paper, we develop a generic model representation and a generic and parameterizable *compare* / *match* / *merge* infrastructure underlying the refactoring framework. Using them, we prove that our refactoring approach is *semantically correct*, i.e., it can generate exactly the original products, regardless of a particular implementation used and parameters chosen. The main contribution of this paper is thus the formal foundation that underlays the parameterizable and configurable, yet semantically correct refactoring framework.

There are multiple ways to *merge-in* input products into a product line, even if we only consider those refactorings that maintain the original set of input products. The resulting refactorings vary *syntactically*, depending on how elements are matched and combined. For example, in Fig. 2(b), transitions from `Locking` to `Washing` states of `Controllers` A and C (Fig. 1 (a,c)) are matched to each other and combined, while their corresponding actions are annotated by features. Instead, these transitions do not have to be matched, so that the generated result has two separate transitions, each annotated by the corresponding feature. Also, the `Unlocking` state of `Controller` A in Fig. 1(a) could be matched and combined with the `Beeping` state of `Controller` C in Fig. 1(c) because of their structural similarity – both transition to the final state of the statechart.

In this work, we formally prove that all these syntactically different refactorings are able to produce the set of original input products and thus are "correct". Elsewhere [20], we focus on techniques for distinguishing between multiple possible refactorings based

on their qualitative properties and choosing a desired one which satisfies the set of defined objectives (e.g., one objective might be to decrease the size of the produced result, while another – to keep a low number of annotated elements per diagram). In [20], we also instantiate our approach on product lines defined in UML – a common specification language in automotive, aerospace & defense, and consumer electronics domains, and demonstrate its applicability on several large-scale examples.

The remainder of this paper is organized as follows. We introduce our data model and give the necessary background on product lines representations in Sec. 2. We give formal foundations of model merging in Sec. 3 and define our merging-based product line refactoring technique in Sec. 4. We prove semantic correctness of the technique in Sec. 5. We conclude the paper with a discussion of related work in Sec. 6, presenting a summary and future research directions in Sec. 7.

2 Preliminaries

In this section, we describe our representation of models and model elements and fix our notation for representing product line models annotated by features.

Model Representation. Following XMI principles [17], we define models to be trees of typed elements. Each element has a unique *id* which identifies it within the model and a *role* which defines the relationship between the element and its parent. For example, in UML, an element of type `Behavior` can have an `Entry` action or `Do` activity roles in a state. In addition, a single element can fulfill several roles in a model: a `Behavior` can be a `Do` activity of a state and an `Effect` of a transition at the same time. To allow reusing elements for different roles, we employ a cross-referencing mechanism where an element of type `Ref` represents the referenced element by carrying its id. Cross-referencing, combined with roles, allows representing labeled graphs using trees: an element can be linked to multiple different elements, each time in a distinct role.

Element types, denoted by $\mathbb{T}$, and roles, denoted by $\mathbb{R}$, are defined by the domain model. For UML, types include `Class`, `State`, `OpaqueBehavior`, etc. Roles include `PackagedElement`, `Subvertex`, `Effect`, etc. If the types `Ref` and `String` are not defined by the domain model, we add them to $\mathbb{T}$ as well.

We differ from [17] by representing all element attributes, as first-class model elements. That is, an element's name is represented by a separate model element of role `Name` and type `String`. The implication of our representation is that elements' attributes now have their own ids and thus, an element can have multiple attributes in the same role, e.g., multiple names or `Effects` for a transition. These qualities are required for defining the product line *merge-in* operator in Sec. 4. A formal representation of our notations is given by Def. 1 below.

Definition 1. *(Model Element) A* model element m *is a tuple* $\langle m|_{id}, m|_t, m|_r, m|_v, m|_s \rangle$, *where* $m|_{id}$ *is a numeric identifier of the element,* $m|_t \in \mathbb{T}$ *is the element's type,* $m|_r \in \mathbb{R}$ *is the element's role,* $m|_v$ *is the element's value – either* `String` *or an id of another element (representing a reference), and* $m|_s$ *is a (nested) list of sub-elements.*

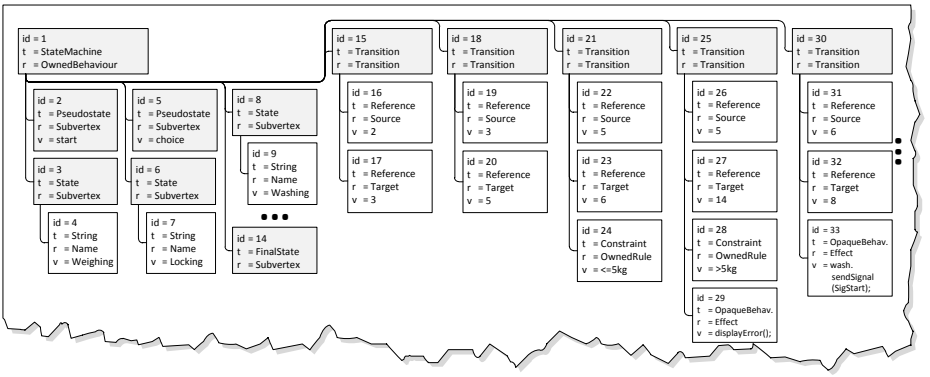


Fig. 3. Partial representation of the Statechart in Fig. 1(a)

Fig. 3 shows partial representation of the `Controller` A statechart in Fig. 1(a), where states `Drying` and `Unlocking`, together with their incoming and outgoing transitions, are omitted to save space. In this figure, sub-elements are represented as element's children in the tree.

We refer to types that have no owned properties, such as `String` or `Ref`, as *atomic*. Other types, such as `Class`, `State` or `Transition`, are *compound*. Elements of atomic and compound types are referred to as *atomic* and *compound elements*, respectively. While atomic elements have values, values of compound elements are determined from values of their sub-elements. Thus, two compound elements may be equal (i.e., have the same type and role, like elements with ids 3 and 6 in Fig. 3) but not equivalent, as they might have different sub-elements.

Definition 2. *(Equivalence) Given a universe of model elements* $\mathbb{M}$*, let* $M_1, M_2 \in 2^{\mathbb{M}}$ *be distinct sets of elements.* $m_1 \in M_1$*,* $m_2 \in M_2$ *are* equal*, denoted by* $m_1 \cong m_2$*, iff* $m_1|_t = m_2|_t$*,* $m_1|_r = m_2|_r$ *and* $m_1|_v = m_2|_v$*. Equal atomic elements are* equivalent*. Compound elements are* equivalent*, denoted by* $m_1 = m_2$*, iff* $m_1 \cong m_2$*, and their corresponding trees of sub-elements are isomorphic wrt. equality.*

Definition 3. *(Model and Model Equivalence) A set of elements* $M \in 2^{\mathbb{M}}$ *is a* model *iff all elements in* M *are connected in a tree structure by the* sub-elements *relationship, and each* $m \in M$ *has a unique id. Models* M_1 *and* M_2 *are* equivalent*, denoted by* $M_1 = M_2$*, iff their corresponding root elements are equivalent.*

Product Line Engineering. Next, we describe the formal semantics of the *annotative* SPLE approach.

Definition 4. *(Feature Model and Configuration – simplified version of [23]) Given a universe of elements* $\mathbb{F}$ *that represent features, a* feature model $\mathcal{FM} = \langle \mathcal{F}, \varphi \rangle$ *is a set of features* $\mathcal{F} \in 2^{\mathbb{F}}$ *and a propositional formula* φ *defined over the features from* $\mathcal{F}$*. A* feature configuration $\widehat{\mathcal{FM}}$ *of* $\mathcal{FM}$ *is a set of selected features from* $\mathcal{F}$ *that respect* φ *(i.e.,* φ *evaluates to* true *when each variable* f *of* φ *is substituted by* true *if* $f \in \widehat{\mathcal{FM}}$ *and by* false *otherwise.)*

Definition 5. *(Product Line – adapted from [2]) A* product line $\mathcal{PL} = \langle \mathcal{FM}, \mathcal{M}, \mathcal{R} \rangle$ *is a triple, where* $\mathcal{FM}$ *is a feature model,* $\mathcal{M} \in 2^{\mathbb{M}}$ *is a domain model, and* $\mathcal{R} \subseteq \mathcal{F} \times \mathcal{M}$ *is a set of relationships that* annotate *elements of* $\mathcal{M}$ *by features of* $\mathcal{F}$.

Fig. 2(a) presents a snippet of a domain model, whose elements are connected to features from a feature model using annotation relationships. In this case, features f_A and f_B are alternative to each other, i.e., the propositional formula φ which specifies their relationship is $(f_A \vee f_B) \wedge \neg(f_A \wedge f_B)$. Thus, the only two valid feature configurations are $\{f_A\}$ and $\{f_B\}$.

A *specific product* derived from a product line *under a particular configuration* is a set of elements annotated by features from this configuration. For example, the statechart in Fig. 1(a) can be derived from the product line in Fig. 2(a) under the configuration $\{f_A\}$.

In this work, we assume that *common* product line elements, i.e., elements that are present in all products derived from a product line, are annotated by all features of $\mathcal{F}$. *Variable* elements are annotated by some, but not all, features of $\mathcal{F}$. To avoid clutter, we do not display annotation relationships for common product line elements in Fig. 2.

We denote by Δ the mapping between an element of the product line model and the corresponding element of the product model. We denote by Δ^{-1} the inverse mapping. For example, let m and $\hat{m}$ refer to the transition between `Locking` and `Washing` states in Fig. 1(a) and Fig. 2(a), respectively. Then, under the configuration $\{f_A\}$, $\Delta(m) = \hat{m}$ and $\Delta^{-1}(\hat{m}) = m$.

Definition 6. *(Product Derivation – adapted from [2]) Let* $\mathcal{PL} = \langle \mathcal{FM}, \mathcal{M}, \mathcal{R} \rangle$ *be a product line and let* $\widehat{\mathcal{FM}}$ *be its feature configuration. A set of model elements* $\hat{M}$ *is* derived *from the product line* $\mathcal{PL}$ under the configuration $\widehat{\mathcal{FM}}$*, denoted by* $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$*, iff the following properties hold:*

(a) An element belongs to the derived model if and only if this element is annotated by a feature of the feature configuration $\widehat{\mathcal{FM}}$ *(under which the derivation was performed):* $\forall m \in \mathcal{M}, \Delta(m) \in \hat{M} \Leftrightarrow \exists f \in \widehat{\mathcal{FM}} \cdot (f, m) \in \mathcal{R}$.

(b) Only one element can be derived from a given domain model element: $\forall m \in \mathcal{M}, \exists! \hat{m} \in \hat{M} \cdot \hat{m} = \Delta(m)$.

(c) Only derived elements are present in the derived model: $\forall \hat{m} \in \hat{M}, \exists! m \in \mathcal{M} \cdot \hat{m} = \Delta(m)$.

(d) Each element of the derived model preserves the type/role/value of its corresponding domain model element: $\hat{m} = \Delta(m) \Rightarrow \hat{m} \cong m$.

(e) Each element of the derived model preserves those sub-elements of its corresponding domain model element that were annotated by the features from $\widehat{\mathcal{FM}}$*:* $\forall \hat{m} \in \hat{M}, \hat{m}^c \in \hat{m}|_s \Leftrightarrow \Delta^{-1}(\hat{m}^c) \in \Delta^{-1}(\hat{m})|_s \wedge \exists f \in \widehat{\mathcal{FM}} \cdot (f, \Delta^{-1}(\hat{m}^c)) \in \mathcal{R})$.

It is easy to show that a feature model configuration uniquely identifies the derived product model.

Lemma 1. *(Uniqueness) Let* $\mathcal{PL} = \langle \mathcal{FM}, \mathcal{M}, \mathcal{R} \rangle$ *be a product line,* $\widehat{\mathcal{FM}}$ *be a feature configuration and* $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$*. Then, for each* $\hat{M}' = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$*,* $\hat{M}' = \hat{M}$.

Table 1. State Similarity Weights $\mathbb{W}$ Used by *Compare* for Fig. 1

Element	Name	Type	Depth	Actions	Transitions
Weight	0.2	0.05	0.1	0.3	0.35

Proof. Assume to the contrary that $\hat{M}' \neq \hat{M}$ and assume without loss of generality that $\exists \hat{m} \in \hat{M}$ such that $\hat{m} \notin \hat{M}'$. By Def. 6(c), $\hat{m} \in \hat{M}$ implies that $\exists m \in \mathcal{M} \cdot \hat{m} = \Delta(m)$. By Def. 6(a), this means that $\exists f \in \widehat{\mathcal{FM}} \cdot (f, m) \in \mathcal{R}$. Since $\hat{M}'$ was derived from $\mathcal{PL}$ under the same configuration $\widehat{\mathcal{FM}}$, $\Delta(m) \in \hat{M}'$ by Def. 6(a), which implies that $\exists \hat{m}' \in \hat{M}' \cdot \hat{m}' = \Delta(m)$ by Def. 6(b). Since $\hat{m} = \Delta(m) = \hat{m}'$, we conclude that $\hat{m} \in \hat{M}'$ which creates a contradiction.

3 Model Merging

In this section, we formalize properties of *model merging* [22,16]. Model merging is an operation which consists of (1) *compare*, which determines how similar model elements are to each other, (2) *match*, which detects pairs of elements that should constitute a match and (3) *merge*, which puts information contained in input models together while keeping a single copy of matched elements. We specify the minimal set of properties that these three *model merging* steps should satisfy in order to be used for combining individual products into product lines.

Compare is a heuristic function that calculates the similarity degree, a number between 0 and 1, for each pair of input model elements. It receives models M_1, M_2 and a set of empirically computed weights $\mathbb{W} = \{w_R \mid R \in \mathbb{R}\}$ which represent the contribution of sub-elements in role R to the overall similarity of their owning elements.

For the example in Fig. 1, a similarity degree between two states is calculated as a weighted sum of the similarity degrees of their names, entry and exit actions, do activities, incoming and outgoing transitions, etc.[1] Comparing `Locking` states from Fig. 1(a,b) to each other yields a relatively high similarity degree of 0.85, as these elements have identical names and similar incoming transitions. However, their outgoing transitions have different actions and lead to non-similar states; thus, the states are not identical. Comparing `Drying` and `Waiting` states yields a lower number, as these states have different names and different incoming and outgoing transitions.

Definition 7. *(*Compare*) Let $M_1, M_2 \in 2^{\mathbb{M}}$ be models.* Compare$(M_1, M_2, \mathbb{W})$ *is a total function that produces a set of triples $C \subseteq (M_1 \times M_2 \times [0..1])$ that satisfy the following properties:*

(a) The similarity degree of equal elements is 1*:* $(m_1 = m_2) \Rightarrow (m_1, m_2, 1) \in C$.
(b) The similarity degree of elements having different types or roles is 0*:*
$(m_1|_t \neq m_2|_t) \vee (m_1|_r \neq m_2|_r) \Rightarrow (m_1, m_2, 0) \in C$.
(c) While comparing, references are substituted by the elements they refer to:
$m_1|_t = m_2|_t = \texttt{Ref} \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow (M_1[m_1|_v], M_2[m_2|_v], x) \in C)$;
$m_1|_t = \texttt{Ref} \wedge m_2|_t \neq \texttt{Ref} \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow (M_1[m_1|_v], m_2, x) \in C)$;
$m_1|_t \neq \texttt{Ref} \wedge m_2|_t = \texttt{Ref} \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow (m_1, M_2[m_2|_v], x) \in C)$.

[1] Some *compare* algorithms, e.g., [16], might perform several iterations until they stabilize and calculate the final similarity degree between elements.

(d) $\text{compare}_{T,R}$ *are domain-specific functions, used to calculate the similarity degree between atomic elements of type* T *in role* R *(e.g., elements' names):* $m_1|_t = m_2|_t = T$, $m_1|_r = m_2|_r = R$, T *is atomic* $\Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow x = \text{compare}_{T,R}(m_1, m_2))$.

(e) The similarity degree of compound elements is calculated as a weighted sum of their sub-elements' similarity: $m_1|_t = m_2|_t = T$, T *is compound* $\Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow x = \sum_{\{R\}} w_R * s_R)$, *where* $\{R\}$ *is a set of possible roles for sub-elements of* T, w_R *is the contribution of sub-elements in role* R *to the overall similarity of* T ($\sum_{\{R\}} w_R = 1$), *and* s_R *is the calculated similarity between sub-elements of* m_1 *and* m_2 *in role* R.

Modifying weights $\mathbb{W}$ can produce syntactically different matches. To obtain the model in Fig. 2(b), we calculated state similarity using weights in Table 1, which were set empirically. Decreasing the weight of the name similarity between states while increasing the weight of the similarity of their corresponding incoming and outgoing transitions could, for example, result in lowering the similarity degree between `Washing` states in Fig. 1(a,c) from 0.8 to 0.7, as their incoming and outgoing transitions differ significantly. This can subsequently lead to not matching these states and thus, unlike in the model in Fig. 2(b), each would be present in the resulting refactoring.

Match is a heuristic function that receives pairs of model elements together with their similarity degree and returns those pairs that are considered similar, using empirically determined *similarity thresholds* $\mathbb{S} = \{S_T \mid T \in \mathbb{T}\}$. Matched elements are combined together by the *merge* function, while unmatched are copied to the result without modification.

Definition 8. *(*Match*) Let* $M_1, M_2 \in 2^{\mathbb{M}}$ *be models and let* C *be a set of triples produced by* compare$(M_1, M_2, \mathbb{W})$. *Then,* match$(M_1, M_2, C, \mathbb{S})$ *is a function that produces a set of pairs* $S \subseteq (M_1 \times M_2)$ *that satisfy the following properties:*

(a) Each element from M_1 *can be matched with only one element of* M_2, *and vice versa:* $(m_1, m_2) \in S \Rightarrow \forall (m_1', m_2') \in S (m_1'|_{id} = m_1|_{id} \Leftrightarrow m_2'|_{id} = m_2|_{id})$.

(b) Only identical atomic elements are matched: $m_1|_t = m_2|_t = T$, T *is atomic* $\Rightarrow (m_1, m_2) \in S \Leftrightarrow (m_1, m_2, 1) \in C$.

(c) Compound elements are matched only if their similarity degree exceeds the threshold that is set for their type: $m_1|_t = m_2|_t = T$, T *is compound* $\Rightarrow (m_1, m_2) \in S \Leftrightarrow (m_1, m_2, x) \in C \wedge x \geq S_T$.

(d) If two elements are matched, their parent elements are matched as well (e.g., it is not possible to match transition guards without matching the owning transitions): $(m_1, m_2) \in S \Rightarrow (\exists m_1^p \in M_1, m_2^p \in M_2 \cdot m_1 \in m_1^p|_s \wedge m_2 \in m_2^p|_s \Rightarrow (m_1^p, m_2^p) \in S)$.

(e) Either root elements of M_1 *and* M_2 *are matched with each other, or one of them has no match at all:* $\neg\exists m_1^p \in M_1 \cdot m_1 \in m_1^p|_s \wedge \neg\exists m_2^p \in M_2 \cdot m_2 \in m_2^p|_s \Rightarrow ((m_1, m_2) \in S \vee \neg\exists m_1' \in M_1 \cdot (m_1', m_2) \in S \vee \neg\exists m_2' \in M_2 \cdot (m_1, m_2') \in S)$.

Consider the above example where `Washing` states had the calculated similarity degree of 0.8 and 0.7 for two different settings of *compare* weights $\mathbb{W}$. Setting the state similarity threshold to 0.75 results in matching the states to each other in the former case and not matching in the latter. Likewise, the transitions between `Locking` and `Washing` states in Fig. 1(a,c) can be matched, resulting in the refactoring in Fig. 2(b), where the corresponding actions are parameterized by features, or not matched, resulting in two separate parameterized transitions.

Merge is a function that receives two models together with pairs of their matched elements and returns a merged model that contains all elements of the input, while matched elements are unified and appear in the resulting model only once.

We denote by σ the mapping from an element of an input model to its corresponding element in the merged result, and say that σ *transforms* an input model element to its corresponding element in the result. We denote by σ_1^{-1} and σ_2^{-1} the reverse mappings from an element in the merged result to its *origin* in the first and second models, respectively (or $\emptyset$ if such an element does not exist in one of them). For example, let m_1, m_2 and m denote the states `Washing` in the models in Fig. 1(a), 1(b) and 2(a), respectively. Then, $\sigma(m_1) = \sigma(m_2) = m$, $\sigma_1^{-1}(m) = m_1$ and $\sigma_2^{-1}(m) = m_2$.

Definition 9. *(*Merge*) Let $M_1, M_2 \in 2^{\mathbb{M}}$ be models, C be a set of triples produced by* compare*$(M_1, M_2, \mathbb{W})$ and S be a set of pairs produced by* match*$(M_1, M_2, C, \mathbb{S})$. Then,* merge*$(M_1, M_2, S)$ is a function that produces the merged model $\bar{M}$ and satisfies the following properties:*

(a) Matched elements are transformed to the same element in the output model $\bar{M}$: $(m_1, m_2) \in S \Leftrightarrow \sigma(m_1) = \sigma(m_2)$.

(b) Each input model element is transformed to exactly one element of $\bar{M}$: $\forall m_1 \in M_1, \exists! \bar{m} \in \bar{M} \cdot \bar{m} = \sigma(m_1)$ *and* $\forall m_2 \in M_2, \exists! \bar{m} \in \bar{M} \cdot \bar{m} = \sigma(m_2)$.

(c) Each element of $\bar{M}$ is created from an element of M_1 and/or an element of M_2. Moreover, no two distinct elements of an input model can be transformed to the same element in the result: $\forall \bar{m} \in \bar{M} \cdot (\exists! m_1 \in M_1 \cdot m_1 = \sigma_1^{-1}(\bar{m})) \vee (\exists! m_2 \in M_2 \cdot m_2 = \sigma_2^{-1}(\bar{m}))$.

(d) Each element of $\bar{M}$ preserves the type, role and value of its corresponding original elements. (By Def. 7(b) and 8(b), only elements with the same type, role and value can be matched: atomic elements are matched only if identical, while compound elements do not have values.) $\forall m \in M_1 \cup M_2, \forall \bar{m} \in \bar{M}, \bar{m} = \sigma(m) \Rightarrow \bar{m} \cong m$.

(e) Each element of $\bar{M}$ preserves sub-elements of its corresponding original elements: $\forall \bar{m} \in \bar{M}, \bar{m}^c \in \bar{m}|_s \Leftrightarrow (\sigma_1^{-1}(\bar{m}^c) \in \sigma_1^{-1}(\bar{m})|_s) \vee (\sigma_2^{-1}(\bar{m}^c) \in \sigma_2^{-1}(\bar{m})|_s)$.

While the *compare* and *match* functions rely on heuristically set weights $\mathbb{W}$ and similarity degrees $\mathbb{S}$, *merge* is not heuristic: its output is uniquely defined by the input set of matched elements. For this work, we rely on *union-merge* [22] realization of the *merge* function. *Union-merge* unifies matched elements and copies unmatched elements "as is" to the result. Since our data model in Sec. 2 represents attributes of model elements as separate entities, an element in the merged result can have several attributes of the same type fulfilling the same role (which, for example, is not allowed by UML for effects on a transition or state *do* activities). We use this property of the data model to capture annotative product line representations generated when merging individual products into product lines.

4 Product Line Refactoring

In this section, we define the *merge-in* operator, which is used to put together input products into a product line. It constructs a product line by adding input products one by one and has two parameters: an (already constructed) product line and the next model to add[2]. For the example in Fig 1, combining `Controller` A and B in Fig. 1(a,b) results

[2] The first product is implicitly converted into a "primitive" product line – a product line with only one feature and a set of annotations that relate all model elements to that feature.

in a product line A+B depicted in Fig. 2(a), with features f_A and f_B. Selecting the first one derives the original statechart of `Controller` A, while selecting the second – that of `Controller` B. Subsequent *merge-in* of `Controller` C (Fig. 1(c)) into this product line produces a representation depicted in Fig. 2(b), out of which all three original statecharts can be derived.

Definition 10. *(*Merge-in *Construction)* $\mathcal{PL}' = \langle \mathcal{FM}', \mathcal{M}', \mathcal{R}' \rangle$ *is a product line constructed by* merging-in *a product* M *into the product line* $\mathcal{PL}$ *(denoted by* $\mathcal{PL}' = \mathcal{PL} \oplus_{\mathbb{W},\mathbb{S}} M$*), using the rules below:*

(a) A new feature f_M*, representing the* merged-in *product* M*, is added as an alternative to all existing features: if* $\mathcal{FM} = \langle \mathcal{F}, \varphi \rangle$ *then* $\mathcal{FM}' = \langle \mathcal{F}', \varphi' \rangle$*,* $\mathcal{F}' = \mathcal{F} \cup \{f_M | f_M \in \mathbb{F}, f_M \notin \mathcal{F}\}$*, and* $\varphi' = (\varphi \vee f_M) \wedge \bigwedge_{f \in \mathcal{F}} \neg(f_M \wedge f)$.

(b) The domain model is generated by merging *the existing domain model with the newly added model* M*: if* $C = \text{compare}(\mathcal{M}, M, \mathbb{W})$ *and* $S = \text{match}(\mathcal{M}, M, C, \mathbb{S})$ *then* $\mathcal{M}' = \text{merge}(\mathcal{M}, M, S)$.

(c) The set of annotation relationships is enhanced by the relationships that annotate elements that originated in M *by* f_M*:* $\mathcal{R}' = \{(f, \sigma(m)) \mid f \in \mathcal{F}, m \in \mathcal{M}, (f, m) \in \mathcal{R}\} \cup \{(f_M, \sigma(m)) \mid m \in M\}$.

We refer to $\mathcal{PL}$ *as the* original *product line and to* $\mathcal{PL}'$ *as the* constructed *product line.*

5 Correctness of Product Line Refactoring

In this section, we prove the correctness of the *merge-in* operator introduced in Sec. 4. Specifically, we show that *merge-in* produces minimal behavior-preserving product line refinements [2], that is, the input product models are the only ones which can be derived from the refactored product line model (Theorem 1).

In what follows, let $\mathbb{W}$ be a set of weights used by the *compare* function and $\mathbb{S}$ be a set of similarity thresholds used by the *match* functions. Let $\mathcal{PL} = \langle \mathcal{FM}, \mathcal{M}, \mathcal{R} \rangle$ be a product line.

***Merge-in* Monotonicity.** Lemma 2 below shows that any feature configuration that contains only features from the original product line $\mathcal{PL}$ is also a valid feature configuration for the constructed product line $\mathcal{PL}'$, i.e., it complies to the constrains φ defined on the features of $\mathcal{PL}'$. For the example in Fig. 2, this means that a feature configuration of the product line A+B in Fig. 2(a), e.g., $\{f_A\}$, is also a valid feature configuration for the "extended" product line A+B+C in Fig. 2(b).

Lemma 2. *Let* $\widehat{\mathcal{FM}}$ *be a subset of* $\mathcal{FM}$*. Then,* $\widehat{\mathcal{FM}}$ *is a feature configuration of* $\mathcal{FM}$ *if and only if it is a feature configuration of* $\mathcal{FM}'$.

Proof. By construction of φ' (Def. 10(a)), $\varphi' = (\varphi \vee f_M) \wedge \bigwedge_{f \in \mathcal{F}} \neg(f_M \wedge f)$. Since $f_M \notin \widehat{\mathcal{FM}}$, $\neg(f_M \wedge f)$ evaluates to *true* for every f, and $\varphi' = (\varphi \vee \mathit{false}) = \varphi$. Thus, $\widehat{\mathcal{FM}}$ respects φ if and only if it respects φ'.

Lemma 3 shows that, under configurations used in Lemma 2, a model derived from $\mathcal{PL}$ is equal to the one derived from $\mathcal{PL}'$. That is, under the configuration $\{f_A\}$, the same model of `ControllerA` in Fig. 1(a) is derived from both product lines A+B and A+B+C (Fig. 2(a) and (b), respectively).

Lemma 3. *Let $\widehat{\mathcal{FM}}$ be a subset of $\mathcal{FM}$. If $\widehat{\mathcal{FM}}$ is a feature configuration for $\mathcal{FM}$, $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$ and $\hat{M}' = \Delta(\mathcal{PL}', \widehat{\mathcal{FM}})$, then $\hat{M} = \hat{M}'$. That is, given a feature configuration that contains only features from $\mathcal{PL}$, a set of elements that is generated from $\mathcal{PL}$ is equivalent to that generated from $\mathcal{PL}'$, under the same configuration.*

Proof. To prove the lemma, we show that $f = \Delta(\sigma(\Delta^{-1}(.)))$ is an isomorphism between the elements of $\hat{M}$ and the elements of $\hat{M}'$ that respects $\cong$. That is, we prove the following four statements, showing that f is an edge-preserving bijection. The construction of the corresponding elements in $\hat{M}$ and $\hat{M}'$ is schematically sketched in Fig. 4.

1. Any element of $\hat{M}$ has the corresponding equal element in $\hat{M}'$: $\forall \hat{m} \in \hat{M}, \exists! \hat{m}' \in \hat{M}' \cdot \hat{m}' = f(\hat{m}) \wedge \hat{m}' \cong \hat{m}$.

Let $\hat{m} \in \hat{M}$. By Def. 6(a), this means that there exists an element $m \in \mathcal{M}$, and a feature $f \in \mathcal{FM}$, such that $(f, m) \in \mathcal{R}$ and $\hat{m} = \Delta(m)$. By Def. 9(b), m is transformed by *merge* to an element $\bar{m}' \in \mathcal{M}'$, such that $\bar{m}' = \sigma(m)$. By Def. 10(c), this element is annotated by the same feature as m: $(f, \sigma(m)) \in \mathcal{R}'$. Thus, $\Delta(\sigma(m)) \in \hat{M}$ by Def. 6(a). Since $\hat{m}$ is derived from m, $m = \Delta^{-1}(\hat{m})$. It follows that $\Delta(\sigma(\Delta^{-1}(\hat{m}))) \in \hat{M}'$. Let's denote that element by $\hat{m}'$. There exists only one such element by Def. 6(b,c) and 9(b). $\hat{m}' \cong \hat{m}$ by Def. 6(d) and 9(d).

2. Any element of $\hat{M}'$ has the corresponding equal element in $\hat{M}$: $\forall \hat{m}' \in \hat{M}', \exists! \hat{m} \in \hat{M} \cdot \hat{m}' = f(\hat{m}) \wedge \hat{m}' \cong \hat{m}$.

Let $\hat{m}' \in \hat{M}'$. By Def. 6(a), this means that there exist an element $\bar{m}' \in \mathcal{M}'$, and a feature $f \in \mathcal{FM}$, such that $(f, \bar{m}') \in \mathcal{R}'$ and $\hat{m}' = \Delta(\bar{m}')$. By Def. 9(c), there are three possible cases: (1) $\sigma_1^{-1}(\bar{m}') \in \mathcal{M}, \sigma_2^{-1}(\bar{m}') = \emptyset$; (2) $\sigma_1^{-1}(\bar{m}') = \emptyset, \sigma_2^{-1}(\bar{m}') \in M$; (3) $\sigma_1^{-1}(\bar{m}') \in \mathcal{M}$, $\sigma_2^{-1}(\bar{m}') \in M$.

For cases (1) and (3), $(f, \bar{m}') \in \mathcal{R}'$ implies that $(f, \sigma_1^{-1}(\bar{m}')) \in \mathcal{R}$ by Def. 10(c), and thus, $\Delta(\sigma_1^{-1}(\bar{m}')) \in \hat{M}$ by Def. 6(a). Let's denote this element by $\hat{m}$. It is easy to see that $f(\hat{m}) = \hat{m}'$ (that is $\Delta(\sigma(\Delta^{-1}(\hat{m}))) = \hat{m}'$. There exists only one such element $\hat{m}$ by Def. 6(b,c) and 9(c). $\hat{m}' \cong \hat{m}$ by Def. 6(d) and 9(d). For case (2), $\sigma_1^{-1}(\bar{m}') = \emptyset$ implies by Def. 10(c), that $\bar{m}'$ is annotated by f_M, and, since $f_M \notin \widehat{\mathcal{FM}}$, $\Delta(\bar{m}') \notin \hat{M}'$, which, together with $\hat{m}' = \Delta(\bar{m}')$, creates a contradiction to $\hat{m}' \in \hat{M}'$.

3. Any sub-element of $\hat{m}$ has the corresponding sub-element in $f(\hat{m})$: $\forall \hat{m} \in \hat{M} (\hat{m}^c \in \hat{m}|_s \Rightarrow f(\hat{m}^c) \in f(\hat{m})|_s)$.

Since $\hat{m}^c \in \hat{m}|_s$, by Def. 6(a,e), there exist elements $m, m^c \in \mathcal{M}$, and features $f, f^c \in \mathcal{FM}$, such that $(f, m) \in \mathcal{R}$, $(f^c, m^c) \in \mathcal{R}$, $\hat{m} = \Delta(m)$, $\hat{m}^c = \Delta(m^c)$ and $m^c \in m|_s$ (it is also possible that $f = f^c$). By Def. 9(b,e), $\sigma(m^c) \in \sigma(m)|_s$. By Def. 10(c), $(f, \sigma(m)) \in \mathcal{R}'$ and $(f^c, \sigma(m^c)) \in \mathcal{R}'$, which, by Def. 6(a,e), implies that $\Delta(\sigma(m^c)) \in \Delta(\sigma(m))|_s$. Since $m^c = \Delta^{-1}(\hat{m}^c)$ and $m = \Delta^{-1}(\hat{m})$, $f(\hat{m}^c) \in f(\hat{m})|_s)$, as desired.

4. Any sub-element of $\hat{m}'$ has the corresponding sub-element in $\hat{m}$: $\forall \hat{m}' \in \hat{M}' (\hat{m}'^c \in \hat{m}'|_s \Rightarrow \exists \hat{m}, \hat{m}^c \in \hat{M} \cdot \hat{m}' = f(\hat{m}) \wedge \hat{m}'^c = f(\hat{m}^c) \wedge \hat{m}^c \in \hat{m}|_s$.

Let $\hat{m}'^c, \hat{m}' \in \hat{M}'$ be elements such that $\hat{m}'^c \in \hat{m}'|_s$. By Def. 6(a,e), there exist elements $\bar{m}', \bar{m}'^c \in \mathcal{M}'$, and features $f, f^c \in \mathcal{FM}$, such that $(f, \bar{m}') \in \mathcal{R}'$, $(f^c, \bar{m}'^c) \in \mathcal{R}'$, $\hat{m}' = \Delta(\bar{m}')$, $\hat{m}'^c = \Delta(\bar{m}'^c)$ and $\bar{m}'^c \in \bar{m}'|_s$ (it is also possible that $f = f^c$). Similarly to case 2, $\sigma_1^{-1}(\bar{m}') \neq \emptyset$ and $\sigma_1^{-1}(\bar{m}'^c) \neq \emptyset$. By Def. 9(e), either $\sigma_1^{-1}(\bar{m}'^c) \in \sigma_1^{-1}(\bar{m}')|_s$ or there exist $m_1, m_2 \in M$, such that $\sigma_1^{-1}(\bar{m}'^c)$ is matched with m_1, $\sigma_1^{-1}(\bar{m}')$ is matched with m_2, and $m_1 \in m_2|_s$. The later case is impossible by Def. 8(a,d,e) – we omit the details due to the space limitations. For the former case, since $(f, \bar{m}') \in \mathcal{R}'$, $(f^c, \bar{m}'^c) \in \mathcal{R}'$, by Def. 10(c), $(f, \sigma_1^{-1}(\bar{m}')) \in$

$\mathcal{R}$, $(f^c, \sigma_1^{-1}(\bar{m}'^c)) \in \mathcal{R}$ and thus, by Def. 6(a,e), $\Delta(\sigma_1^{-1}(\bar{m}'^c))) \in \Delta(\sigma_1^{-1}(\bar{m}')))|_s$. Let's denote these elements by $\hat{m}^c$ and $\hat{m}$, respectively. $f(\hat{m}^c)) = \Delta(\bar{m}'^c) = \hat{m}'^c$ and $f(\hat{m})) = \Delta(\bar{m}') = \hat{m}'$, implies $\hat{m}^c \in \hat{m}|_s$, as desired.

The above lemma implies that our construction preserves the behavior of the original product line model: the set of models derived from $\mathcal{PL}$ can still be derived from $\mathcal{PL}'$, as shown by the following corollary.

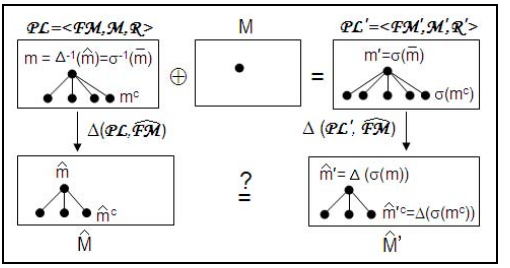


Fig. 4. A sketch for the proof of Lemma 3

Corollary 1. *Let $\lfloor \mathcal{PL} \rfloor$ denote a set of all models derived from a product line $\mathcal{PL}$. That is, $\lfloor \mathcal{PL} \rfloor = \{\Delta(\mathcal{PL}, \widehat{\mathcal{FM}}) \mid \widehat{\mathcal{FM}}$ is a feature configuration of $\mathcal{FM}\}$. Then, a set of models derived from $\mathcal{PL}$ can be derived from $\mathcal{PL}'$ as well: $\lfloor \mathcal{PL} \rfloor \subseteq \lfloor \mathcal{PL}' \rfloor$.*

Proof. For each $\hat{M} \in \lfloor \mathcal{PL} \rfloor$, there exists a configuration $\widehat{\mathcal{FM}}$, such that $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$. By Lemmas 2 and 3, $\hat{M} = \Delta(\mathcal{PL}', \widehat{\mathcal{FM}})$. Thus, $\hat{M} \in \lfloor \mathcal{PL}' \rfloor$.

For the example in Fig. 2, the above corollary means that both `Controller` A and `Controller` B that can be derived from the product line A+B in Fig. 2(a) can still be derived from the constructed product line A+B+C in Fig. 2(b), after `Controller` C was *merged-in* to it.

***Merge-in* Behavior Preservation.** We now show that model M which we *merge-in* into the original product line $\mathcal{PL}$ can be derived from the constructed product line $\mathcal{PL}'$. That is, when we *merge-in* `Controller` C in Fig. 1(c) into the product line A+B in Fig. 2(a), we can derive it back from the constructed product line A+B+C in Fig. 2(b).

Since f_M is the feature that annotates elements of the *merged-in* model (f_C in our example), we first show that $\{f_M\}$ is a valid feature configuration (Lemma 4). Then, Lemma 5 shows that the original model M is derived from the constructed product line $\mathcal{PL}'$ under that configuration.

Lemma 4. *$\{f_M\}$ is a feature configuration for $\mathcal{PL}'$.*

Proof. By construction of $\mathcal{FM}'$ (Def. 10(a)), $f_M \in \mathcal{F}'$. We now show that $\{f_M\}$ respects $\varphi' = (\varphi \vee f_M) \wedge \bigwedge_{f \in \mathcal{F}} \neg(f_M \wedge f)$. Since $f \notin \{f_M\}$ for any $f \in \mathcal{F}$, $\neg(f_M \wedge f)$ evaluates to *true* for every $f \in \mathcal{F}$. Since $f_M = true$, $\varphi \vee f_M$ also evaluated to *true*. It follows that $\{f_M\}$ respects φ' and is a feature configuration for $\mathcal{PL}'$.

Lemma 5. *Let $\{f_M\}$ be a feature configuration. Then, a model that is derived from $\mathcal{PL}'$ under that configuration is equivalent to M. That is, $M = \Delta(\mathcal{PL}', \{f_M\})$.*

The proof of this lemma, similarly to the proof of Lemma 3, shows that $f = \sigma(\Delta(.))$ is an isomorphism between the elements of M and the elements of $\hat{M}'$, and is omitted.

Finally, Theorem 1 shows that our *merge-in* operator is behavior preserving: the set of product models that are derived from the constructed product line $\mathcal{PL}'$ is equal to the set of models that are derived from the original product line $\mathcal{PL}'$, in addition to the *merged-in* model M.

Theorem 1. $\lfloor \mathcal{PL}' \rfloor = \lfloor \mathcal{PL} \rfloor \cup \{M\}$.

Proof. We first prove that $\lfloor \mathcal{PL}' \rfloor \subseteq \lfloor \mathcal{PL} \rfloor \cup \{M\}$. Let $\hat{M} \in \lfloor \mathcal{PL}' \rfloor$ be a model derived from $\mathcal{PL}'$. Then there exists a feature configuration $\widehat{\mathcal{FM}'}$, such that $\hat{M} = \Delta(\mathcal{PL}', \widehat{\mathcal{FM}'})$. Let f_M be in $\mathcal{FM}' \setminus \mathcal{FM}$.

1. If $f_M \notin \widehat{\mathcal{FM}'}$, then $\widehat{\mathcal{FM}'} \subseteq \mathcal{FM}$. Thus, by Lemma 3, $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}'})$, which implies that $\hat{M} \in \lfloor \mathcal{PL} \rfloor$.
2. If $f_M \in \widehat{\mathcal{FM}'}$, then, by construction of $\mathcal{FM}'$ (Def. 10(a)), $\widehat{\mathcal{FM}'} = \{f_M\}$. By Lemma 5, $M = \Delta(\mathcal{PL}', \widehat{\mathcal{FM}'})$. Thus, by Lemma 1, $\hat{M} = M$.

We now show that $\lfloor \mathcal{PL} \rfloor \cup \{M\} \subseteq \lfloor \mathcal{PL}' \rfloor$. $\lfloor \mathcal{PL} \rfloor \subseteq \lfloor \mathcal{PL}' \rfloor$ by Corollary 1. By the construction of $\mathcal{FM}'$ (Def. 10(a)), $f_M \in \mathcal{F}'$. Thus, by Lemmas 4 and 5, $\{f_M\}$ is a valid feature configuration for $\mathcal{PL}'$ and $M = \Delta(\mathcal{PL}', \{f_M\})$, which implies that $M \in \lfloor \mathcal{PL}' \rfloor$.

For the example in Fig. 2, where `Controller` C in Fig. 1(c) is *merged-in* into the product line A+B containing `Controller` A and B, this means that `Controller` A, B, and C, and only them, can be derived from the constructed product line A+B+C in Fig. 2(b).

6 Related Work

A general theory of product line refinement was introduced in [2] where the authors established product line properties supporting stepwise and compositional product line development and evolution. Our approach instantiates this theory by providing a concrete refactoring technique for combining products into product lines. We prove that our refactoring is the minimal behavior-preserving product line refinement, according to the definition in [2].

Several works (e.g., [9,10]) capture guidelines and techniques for manually transforming legacy product line artifacts into SPLE representations. Instead, our goal is to introduce automation into the refactoring process by *comparing*, *matching* and *merging* artifacts to each other. While no automated approach can replace a human product line designer and produce a solution which is as good as a hand-crafted one, automation can assist the designer and speed-up the refactoring process.

Similarly to us, Koschke et. al. [11] and Ryssel et. al. [21] introduce automatic approaches to re-organize product variants into annotative representations while identifying variation points and their dependencies. The former work reasons about components, interfaces and their grouping into subsystems. The latter works on Matlab models. Our work differs from both [11] and [21] by exploring product line commonalities and variabilities for any type of model that can be represented as XMI and by providing a formal proof of correctness of our approach.

Feature-oriented refactoring [13,15] focuses on identifying the code for a feature and factoring the code out into a single module or aspect aiming at decomposing a program into features. Since our aim is consolidation of variants into single-base product line representations, these are out of the scope for our work. Similarly, UML model refactoring (e.g., [6,24]) and code refactoring techniques (e.g., [14]), while closely related to our work, usually focus on improving the internal structure and design of a software system rather than on identifying and restructuring the system's common and variable parts.

7 Conclusion and Future Work

Extracting product line representations from existing legacy product line systems can support product line engineering adoption: reusing and leveraging knowledge accumulated in the legacy systems during their development lifetime can be more efficient than "starting from scratch". In this work, we formally specified a simple data model and a refactoring technique for transforming individual products into more compact product line representations. Our data model, inspired by XMI principles, is powerful enough to accommodate labeled-graph representations, in particular, UML. At the same time, it is flexible enough to support product line notations where several alternative elements can fulfill the same role, which is not allowed by UML itself.

Relying on the data model, we formally stated necessary and sufficient conditions allowing us to use model *compare*, *match* and *merge* operators for combining individual products into product lines. We proved that once these conditions are satisfied, the *merge-in* can be safely applied for combining products into product lines, as it produces representations that encode precisely the set of initial products. This provides formal foundation that underlays the parameterizable and configurable, yet semantically correct refactoring framework. The applicability of the framework to real-life examples, as well as techniques for distinguishing between different possible refactorings, is studied elsewhere [20].

There are several directions for continuing this work. First, we are interested in exploring more sophisticated refactoring techniques that are able to detect fine-grained features in the combined products. This would allow us to create new products in the product line by "mixing" features from different original products. We also plan to enhance *model merging* techniques with additional capabilities, such as using code-level clone detection techniques for comparing statechart actions and activities. We are also interested in devising alternative methods of calculating graph similarity, e.g., by counting the number of identical or similar sub-graphs and more.

References

1. Beuche, D.: Transforming Legacy Systems into Software Product Lines. In: Proc. of SPLC 2011 Tutorial (2011)
2. Borba, P., Teixeira, L., Gheyi, R.: A Theory of Software Product Line Refinement. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 15–43. Springer, Heidelberg (2010)

3. Boucher, Q., Classen, A., Heymans, P., Bourdoux, A., Demonceau, L.: Tag and Prune: a Pragmatic Approach to Software Product Line Implementation. In: Proc. of ASE 2010 (2010)
4. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley (2001)
5. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley (2004)
6. Hosseini, S., Azgomi, M.A.: UML Model Refactoring with Emphasis on Behavior Preservation. In: Proc. of TASE 2008, pp. 125–128 (2008)
7. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU/SEI-90TR-21 (1990)
8. Kastner, C., Apel, S.: Integrating Compositional and Annotative Approaches for Product Line Engineering. In: Proc. of GPCE Wrksp. on Modul., Comp. and Gen. Tech. for PLE (GPLE 2008), pp. 35–40 (2008)
9. Kim, K., Kim, H., Kim, W.: Building Software Product Line from the Legacy Systems: Experience in the Digital Audio and Video Domain. In: Proc. of SPLC 2007 (2007)
10. Kolb, R., Muthig, D., Patzke, T., Yamauchi, K.: Refactoring a Legacy Component for Reuse in a Software Product Line: a Case Study: Practice Articles. J. of Software Maintenance and Evolution 18(2), 109–132 (2006)
11. Koschke, R., Frenzel, P., Breu, A.P., Angstmann, K.: Extending the Reflection Method for Consolidating Software Variants into Product Lines. Soft. Quality Control 17(4) (2009)
12. Krueger, C.W.: Easing the Transition to Software Mass Customization. In: van der Linden, F.J. (ed.) PFE 2002. LNCS, vol. 2290, pp. 282–293. Springer, Heidelberg (2002)
13. Liu, J., Batory, D., Lengauer, C.: Feature Oriented Refactoring of Legacy Applications. In: Proc. of ICSE 2006, pp. 112–121 (2006)
14. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE TSE 30(2), 126–139 (2004)
15. Murphy, G.C., Lai, A., Walker, R.J., Robillard, M.P.: Separating Features in Source Code: an Exploratory Study. In: Proc. of ICSE 2001, pp. 275–284 (2001)
16. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and Merging of Statecharts Specifications. In: Proc. of ICSE 2007, pp. 54–64 (2007)
17. OMG, http://www.omg.org/spec/XMI/2.1.1/ (last Accessed: January 2011)
18. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
19. Rubin, J., Chechik, M.: From Products to Product Lines Using Model Matching and Refactoring. In: Proc. of SPLC Wrksp. (MAPLE 2010) (2010)
20. Rubin, J., Chechik, M.: Quality of Behavior-Preserving Product Line Refactorings (2011); Under review
21. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Extraction of Feature Models from Formal Contexts. In: Proc. of SPLC 2011, pp. 4:1–4:8 (2011)
22. Sabetzadeh, M., Easterbrook, S.: View Merging in the Presence of Incompleteness and Inconsistency. Requirement Engineering 11, 174–193 (2006)
23. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse Engineering Feature Models. In: Proc. of ICSE 2011 (2011)
24. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M.: Refactoring UML Models. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 134–148. Springer, Heidelberg (2001)

Tracing Your Maintenance Work – A Cross-Project Validation of an Automated Classification Dictionary for Commit Messages

Andreas Mauczka, Markus Huber Christian Schanes, Wolfgang Schramm, Mario Bernhart, and Thomas Grechenig

Research Group for Industrial Software, Vienna University of Technology
Vienna 1040, Austria
{andreas.mauczka,markus.huber,christian.schanes,wolfgang.schramm,
mario.bernhart,thomas.grechenig}@inso.tuwien.ac.at
http://www.inso.tuwien.ac.at/

Abstract. A commit message is a description of a change in a Version Control System (VCS). Besides the actual description of the change, it can also serve as an indicator for the purpose of the change, e.g. a change to refactor code might be accompanied by a commit message in the form of "Refactored class XY to improve readability". We would label the change in our example a perfective change, according to maintenance literature. This simplified example shows how it is possible to classify a change by its commit message. However, commit messages are unstructured, textual data and efforts to automatically label changes into categories like perfective have only been applied to a small set of projects within the same company or the same community. In this work, we present a cross-project evaluated and valid mapping of changes to the code base and their purpose that is usable without any customization on any open-source project. We provide further the Eclipse Plug-In Subcat which allows for a comfortable analysis of projects from within Eclipse. By using Subcat, we are able to automatically assess if a commit to the code was e.g. a bug fix or a refactoring. This information is very useful for e.g. developer profiling or locating bad smells in modules.

1 Introduction

Software is constantly evolving. Leading and monitoring software development projects is a difficult task and performance indicators become mandatory for deciding on a course of action, e.g. is now the time to refactor some of my code or do I need to intensify my quality assurance work, because my development team spends a majority of their time troubleshooting and bug fixing. Managers or project leads need to be well informed to enhance their decision making process and to have an accurate view of the current state of the project. By gathering the information that is actually available in the form of meta data in the Version Control System (VCS), conclusions about the software development and maintenance (e.g. which modules are error prone, which modules have not

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 301–315, 2012.

been refactored recently) can be drawn. Since our approach does not rely on the code itself, it can be applied to any programming language and early in the software life cycles, when code metrics might not be conclusive yet.

In the following paper we use meta data which can be mined from a VCS that uses commit messages to accompany any change (a commit) to the code base. From the textual information in these commit messages, we mine information about the software maintenance and evolution process in open source projects. We base our work on the assumption that commit messages hold information that should give evidence of the purpose of the source code change (see Section 5).

We present in this paper two contributions to the analysis of meta data in a VCS. First, we implemented the tool "Subcat" to classify commit messages based on a set of keywords (we refer to the master set of all keywords as a dictionary). Subcat is a generic analysis tool that can be configured to classify any commit message into arbitrary categories based on a dictionary. It further is possible to customize Subcat to fit any personal project vocabulary by adding categories or keywords to the dictionary.

Subcat provides different kinds of reports (categorization per file or per module) to visualize the results of this categorization. Subcat also generates statistics on the authors of the commit messages or statistics on the words used in the commit messages. We provide Subcat as an Eclipse plugin for integrated analysis by developers or project managers during early software lifecycle stages in the Eclipse IDE and as a standalone command line tool for the mining of large scale software projects (see Section 2).

Second, we used the reports generated by Subcat to create an optimized and cross-project valid dictionary that allowed us to automatically classify commits into Swanson's maintenance categories [10] for the open source domain. To achieve this, we defined an algorithm to incrementally train and improve this dictionary with certain keywords. After training the dictionary on a number of projects, we evaluated this dictionary against a larger set of open source projects (see Section 3 for the algorithm and Section 4 for the results of the evaluation).

Subcat can be used in two different contexts. Subcat can be used by practitioners in the open-source domain to analyze modules based on maintenance characteristics. E.g. when a module has a lot of corrective changes, but no perfective changes, some refactoring of the code might be in order (see Figure 1). Furthermore, Subcat provides Maintenance Profiles of the development team. This means that one can see at first glance, whether a developer is mainly fixing bugs or keeping the code clean.

Subcat can also be used by researchers. By using Subcat's corrective classification mechanism, we are able to track bugs within the repository additionally to a normal bug tracker. This allows for research on the difference of bug granularity in repositories and bug trackers like bugzilla (a bug fix in the repository may not correspond to a bug report in the tracker). Additionally, we can use Subcat to analyze how developer profiles change over time in a project (E.g. a developer starts in a project to fix bugs that annoy him and ends up implementing a whole new feature see Figure 2 for an example). Subcat provides this

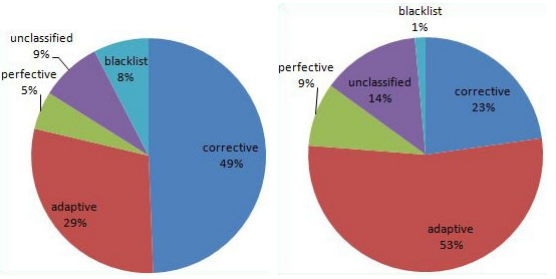


Fig. 1. Visualization of the classified activities in two different software modules

information, which can be combined with mailing list analysis. This can provide a whole new insight into how a developer changes over time in an open-source project.

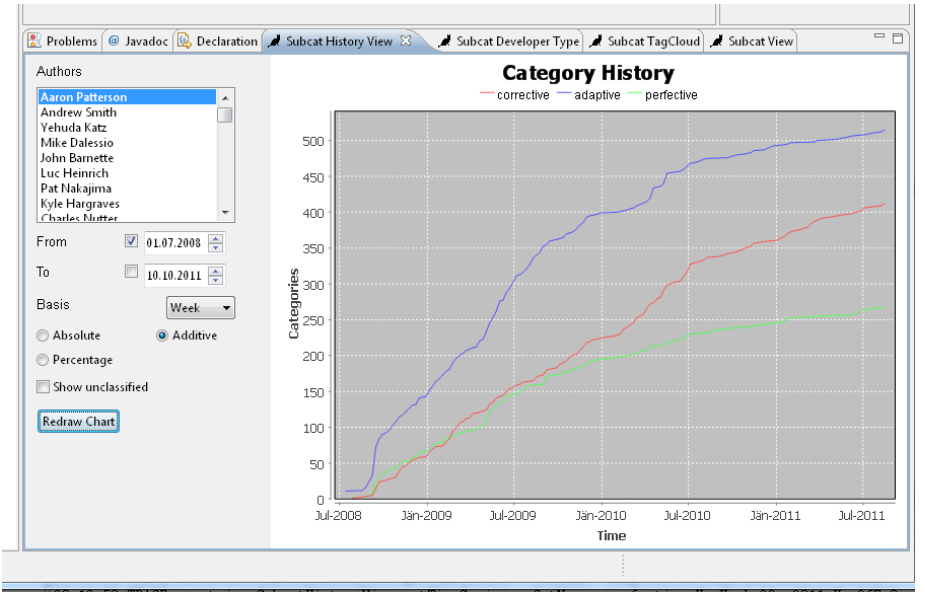


Fig. 2. Developer Profile in Subcat

2 Automated Classification Approach

A dictionary, as used in the context of this paper, is a set of categories. A category is a group of keywords that share a common meaning and therefore are indicators for this category, e.g. the word "fix" is a keyword for the maintenance category "corrective". In the context of this work, we apply Swanson's maintenance categories to group our keywords.

We propose the following procedure to create a dictionary:

Pre-Processing the Meta Information. The meta information for our analysis was derived from the commit messages in the VCS. As these messages are written in natural language, we have to normalize them to be able to extract sensible information (e.g. we want to match "this **fixes** a re-ocurring crash" and "I **fixed** an overflow" to its lemma "fix" - a head word under which the word would be found in a dictionary). We use WordNet[1] to normalize the commit messages

Initializing the Dictionary. We generate an initial seed for a dictionary by referring to prior work (Mockus and Votta in [9] and Hassan in [5]). This initial seed only contains words that hold a high likelihood of indicating a maintenance category

Training the Dictionary. To be able to categorize as many changes as possible with a high accuracy for a single project, we use a defined algorithm to train the dictionary. We employ the algorithm to train the dictionary on additional open source projects to further increase the accuracy of the dictionary (see Section 3)

Evaluating the Dictionary. After the initial training, we use the dictionary on another set of projects to evaluate cross-project validity. We do not further change the dictionary during this step. Only blacklist items (keywords that filter out administrative changes) are introduced (see Section 4)

2.1 Classification Rules

The research area of the identification and classification of maintenance tasks in the software development process has evolved for decades. In [10], Swanson defines a maintenance task as an activity that can be assigned to one of the following three categories:

Corrective Software Maintenance. Activities that are necessary to fix processing failures, performance failures or implementation failures

Adaptive Software Maintenance. Activities that focus on changes in the data environment or changes in the processing environment

Perfective Software Maintenance. Activities that strive to decrease processing inefficiency, enhance the performance or increase the maintainability

For the development of the automated classification in this work, Swanson's original definition of maintenance tasks is used and slightly extended. An additional category, the "Blacklist" is introduced. We use the Blacklist to filter all commits, which underlying modifications were not carried out by humans or which do not actually include any source code modifications. For example commits generated by the "cvs2svn" repository-converter[2] or commits that just "tag" a version. In addition we merged the implementation category, as presented by Hindle et al. [6]

[1] http://wordnet.princeton.edu/
[2] http://cvs2svn.tigris.org/

with Swanson's adaptive maintenance category. As a result we are able to map every commit to exactly one category. Using Swanson's original maintenance classification provides a categorization into a few, well defined categories and is therefore a suitable starting point to develop an automated classification algorithm.

As mentioned above, our algorithm relies on two sources of information to carry out the classification, namely the commit message and the dictionary. The **commit message** is attached to every commit and encapsulates the information about the intention of the modification. The **dictionary** defines the knowledge base for the classification including the categories. The different categories are defined by a set of keywords that indicate that a commit message may belong to this category. In addition, every word has an associated weight. The weight value constitutes how strong the indication is. The same word can be contained in multiple categories. See Figure 3 for a sample dictionary that is used to classify a commit message.

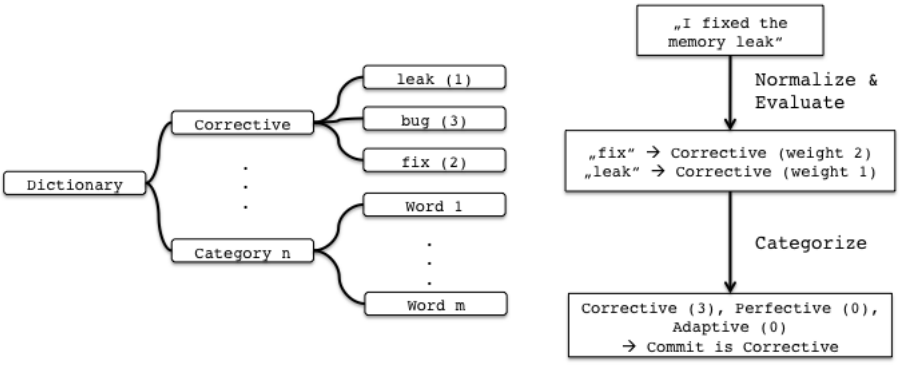


Fig. 3. Example for Dictionary and Classification

To implement the blacklist feature, "absolute categories" have been introduced. If a commit message contains a word (e.g. "cvs2svn") that is included in the listing of an absolute category, the commit is instantly assigned to this category, ignoring the weighting mechanism and the normal categories.

2.2 Categorization Tool - Subcat

Subcat is a tool implemented to generically categorize commit messages based on their content. It consists of two parts, the command line tool Sublex and the Eclipse plugin, which we describe in detail in the following sections.

Sublex is the tool that implements all relevant functionality to classify commits from a generic data source. Due to the modular design of the command line tool different Versioning Systems are supported, if adapters for pre-processing a logfile to the generic data source format are available. The adapter for Subversion is supplied together with Sublex and ships also as a part of the Eclipse plugin.

The results of the classification are reports in the CSV-format. Sublex offers the following reports:

Categorization-Report. The categorization report contains all commits and their corresponding classification results in detail. It is the base for the detail reports that follow. It can be used by analysts to generate their own statistics based on the report data. The displayed information per row are: commit including the revision, the category it has been assigned to, the author, the date of the change, the length of its commit message, the overall number of added and deleted lines for the commit, the score of the commit for each category from the dictionary, the affected modules, the affected files and the revised commit message.

Author-Report. The author report shows the analysis of commits (including the assigned maintenance categories) per author. Its purpose is to analyze the profiles of developers in the project. For example if an author is responsible for perfective maintenance or if perfective maintenance is distributed evenly on the team.

Dictionary-Report. This report provides required information to create and improve dictionaries by showing statistical information for every unique word found in any of the parsed commit messages. The report provides the lemma for the word, the average number of appearances of the word in the commit messages it was found in, the total number of appearances in all the commit messages, the total number of classified and unclassified commits the word was found in

Lemma-Report. The lemma-report is the second required report for creating and improving dictionaries. It includes an entry for every unique lemma, together with the number of classified and unclassified commits the lemma was found in

Modules-Report. This report shows categorization statistics about the modules of a project. Module structure to be analyzed can be parametrized. E.g. the project has the structure of /util/login/security. We configure a module depth of 2. There will be a row for /util/* and one row for util/login/* in the report

Control-Report. This report was used to manually validate the analysis result during our research. It contains every original commit message and the category it was assigned to

Eclipse Plugin. The Eclipse plugin has been implemented to integrate Subcat into the Eclipse IDE to give analysts and developers, but also project managers, a familiar environment for maintenance analysis. The integration into Eclipse allows a comfortable comparison between our reports and any other metric suites a user might employ. E.g. a project manager can view results for code metrics

next to the results of the categorization of the commits of a module and use both of these views in his decision making process. Furthermore, the usability of Subcat is improved by using the point and click paradigm to generate reports, e.g. for authors and modules, as a user can navigate through the proper views in the IDE (see Figure 2).

The Eclipse plugin uses the generic data source adapter for Subversion and the classification functionality and the logic from the command line tool Sublex. Due to its generic approach the classification functionality can be used without adaptations in a different context. The complete plugin project is split into three individual plugins. The generic data source adapter and Sublex and the main plugin which integrates the categorization functionality into the Eclipse workbench. This corresponds to the MVC design pattern (see Buschmann et al. [2]).

3 Generation of a Cross-Project Valid Dictionary

To build a representative dictionary, a set of projects to provide initial keywords and to train our dictionary are required. We further need another set of projects to ensure cross-project validity of the dictionary.

3.1 Criteria and Selection of Open Source Projects

Eight open source projects were chosen to build, test and cross verify the dictionary. The following criteria were used to select the projects:

Number of Commits. For our analysis we only considered projects with at least 30,000 commits[3]

Number of Developers. To show the categorization of the developer role in a project and also to increase the variance of different commit message style only projects with at least 30 developers[4] are considered.

Subversion Repository. Our approach is currently based on Subversion repositories. Therefore only projects with access to their Subversion repositories are included.

Table 1 shows the key figures of the selected projects.

3.2 Populating the Dictionary

As a starting point to create the dictionary we analyzed the log of the FreeBSD-Project and used exemplary keywords from prior work (see [5] and [9]) for the categorization. In the next step we ranked the keywords by occurrence. The top three ranked keywords of each category are included in the first dictionary:

[3] The number of commits is the number of commits in the log.

[4] The number of developers represents the number of distinct author names in the log.

Table 1. Key figures of the analyzed open source projects

App. Name	App. Type	# Devs	# Commits
Boost	Prog. Library	294	63,616
Enlightenment	Window Manager	187	51,884
Evolution	E-Mail-Client	431	37,500
FreeBSD	OS	536	150,595
Firebird	RDBMS	43	51,509
GCC	Compiler-Suite	426	102,672
Python	Interpreter	216	83,100
Wireshark	Packet Analyzer	43	34,067

Corrective: fix, bug, problem
Adaptive: new, change, patch
Perfective: style, move, removal

This first dictionary constituted the "seed" to create a more exhaustive dictionary. This initial dictionary only categorized a low number of commits, leaving a large number of commits uncategorized. Starting with this seed, we set up an algorithm with the goal to increase the ratio of classified commits to 80% while maintaining adequate values for a self-evaluated precision (0.8) and recall (0.8). Values beyond these thresholds yield diminishing results - either less commits will be classified, or precision and recall will suffer. An early attempt at the algorithm had to be abandoned, because of a too conservative approach in adding words to the dictionary (stagnation at about 65% of categorized commits). For the final algorithm we used a more open and flexible approach so that more words would qualify for the dictionary. We further introduced weighting of keywords and rulesets for ambiguous, yet strongly indicative words. The following list describes step wise our final algorithm to create the dictionary:

1. Classify the commit using the "seed" dictionary
2. If the total percentage of classified commits is greater than 80%, EXIT
3. Count the appearances of all words in the commit messages of the non-classified commits and order them by frequency
4. Choose a set of words from the top of the list and add these as a test set to the existing dictionary
5. Count the number of appearances of every word in the test set in each category
6. If the number of appearances of a word in a category is at least 1.5 times of the appearances of the same word in the other categories, add it to the dictionary with a weight of 2 and remove it from the test set
7. If the number of appearances of a word in two classes is at least 1.5 times of the appearances of the same word in the third class, add it to the dictionary to both classes with a weight of 1 and remove it from the test set
8. If neither 6 or 7 are true, remove the word from the test set and do not add it to the dictionary
9. Go to Step 2

This algorithm achieved a classification rate of 80.34 % after 21 iterations on the FreeBSD project. The output is the following dictionary (weights of keywords in brackets, default weight 1).

Corrective: active, against, already, bad, block, bug, build, call, case, catch, cause(2), character, compile, correctly, create, different, dump, error(2), except, exist, explicitly, fail, failure(2), fast, fix(2), format, good, hack, hard, help, init, instead, introduce, issue, lock, log, logic, look, merge, miss(2), null(2), oops(2), operation, operations, pass, previous, previously, probably, problem, properly, random, recent, request, reset, review, run, safe, set, similar, simplify, special, test, think, try, turn, valid, wait, warn(2), warning, wrong(2)

Adaptive: active, add(2), additional(2), against, already, appropriate(2), available(2), bad, behavior, block, build, call, case, catch, change(2), character, compatibility(2), compile, config(2), configuration(2), context(2), correctly, create, currently(2), default(2), different, documentation(2), dump, easier(2), except, exist, explicitly, fail, fast, feature(2), format, future(2), good, hack, hard, header, help, include, information(2), init, inline, install(2), instead, internal(2), introduce, issue, lock, log, logic, look, merge, method(2), necessary(2), new (2), old(2), operation, operations, pass, patch(2), previous, previously, probably, properly, protocol(2) provide(2), random, recent, release(2), replace(2) ,request, require(2), reset, review, run, safe, security(2), set, similar, simple(2), simplify, special, structure(2), switch(2), test, text(2), think, trunk(2), try, turn, useful(2), user(2), valid, version(2), wait

Perfective: cleanup(2), consistent(2), declaration(2), definition(2), header, include, inline, move(2), prototype(2), removal(2), static(2), style(2), unused(2), variable(2), warning, whitespace(2)

Blacklist: cvs2svn, cvs, svn

The analysis further showed that the word "documentation" was assigned to the adaptive category by the algorithm. Since "documentation" is a perfective task per definition, the word "documentation" was moved from **adaptive** back to **perfective**. The implications warrant further research however.

This final dictionary was used to classify the FreeBSD-project again and precision and recall were measured based on modification records (MR) as shown in Table 2.

Table 2. Recall and precision of the classification for the FreeBSD-project

Class	MR	%	Recall	Precision
Corrective	54,015	35.86%	0.92	0.85
Adaptive	56,046	37.21%	0.91	0.80
Perfective	8,484	5.63%	0.86	0.80

We then used the dictionary and the algorithm on the "Boost" project (inital classification rate 74.94%), thereafter on "Enlightenment" project (initial classfication rate 72.80%) and altered the dictionary until it achieved 80%

of classified changes. We decided to train the dictionary on two other projects to achieve a greater classification ratio and to work out project-individual language issues (e.g. ambiguously connotated lemmas). After this training phase, the dictionary was used with the "Evolution", "Firebird", "GCC", "Python" and "Wireshark" projects and scored a classification rate of over 80% for each project, without adaption.

Table 3. Recall and precision of the analysis for various open source projects

Project	# MR	Recall	Precision
Enlightenment	51,884	0.90	0.80
Evolution	37,500	0.96	0.92
Firebird	51,509	0.95	0.90
GCC	102,672	0.92	0.83
Python	83,100	0.93	0.85
Wireshark	34,067	0.92	0.85
FreeBSD	150,595	0.90	0.82
Boost	63,616	0.94	0.88

4 Evaluation of the Dictionary

To evaluate our results, we did a survey with five professional Software Developers. The developers are working for different companies since between two to five years (2,2,4,4 and 5). Our survey was structured as follows:

- Five questionnaires, each with the 21 changes in the code (7 of each category).
- Five questionnaires, each with the same changes in the code, but with their corresponding commit messages

4.1 Inter-rater Agreement

To measure inter-rater Agreement of the developers, we used Fleiss' Kappa on six commits that were identical in each questionnaire. Table 4 shows the agreement amongst the developers for these six commits (two commits per category). The resulting Fleiss' Kappa for this matrix is $\boldsymbol{K = 0.48}$. This indicates a **moderate agreement** according to Landis and Koch's Benchmark [8] between the developers themselves.

If a commit is assigned to two categories, its count is split between the categories.

4.2 Conducting the Evaluation

We conducted the survey in two rounds. Table 5 and Table 6 show the agreement between developers and the automated classification tool. If a developer chose two categories, a point was split between these categories.

Table 4. Matrix showing the agreements amongst the developers for the six common commits in evaluation round two

Commit/Category	**Adap.**	**Corr.**	**Perf.**
Corr. 1	1.0	4.0	0.0
Corr. 2	0.5	4.5	0.0
Perf. 1	1.0	2.0	2.0
Perf. 2	0.0	0.0	5.0
Adap. 1	4.5	0.0	0.5
Adap. 2	4.0	0.0	1.0

Table 5. Agreements between developers and classification tool for the evaluation round one

	Automated Classification			
Developers	**Adap.**	**Corr.**	**Perf.**	
Adaptive	**11.0**	4.5	0.5	16.0
Corrective	4.5	**12.0**	0.5	17.0
Perfective	7.5	8.5	**24.0**	40.0
	23.0	25.0	25.0	**47.0**

Table 7 shows the summarized results of the evaluation rounds one and two. The columns show the total number of agreements between the developers and the automated classifications for each category and the Cohen's Kappa-value.

4.3 Interpretation of the Evaluation

The following conclusions can be drawn from these results:

- Both the agreements in the adaptive category as well as the agreements in the perfective category stayed constant for both rounds. In contrast, the number of agreements for the corrective category has significantly risen between round one and two. From this fact we conclude that corrective maintenance tasks are most difficult to spot just by looking at the source code and without reading the commit message
- The number of agreements for the perfective category is almost perfect in both rounds. We therefore conclude that our classification tool excels at identifying perfective maintenance tasks (a finding similar to Mockus et al's inspection change finding in [9])
- The Kappa-value has risen from 0.46 to 0.61 from round one to round two. This means that with the additional information of the commit message, the developers have converged their decisions with the decisions of the automated classification. Curiously this affected mainly corrective changes
- 0.46 and 0.61 both indicate a **moderate agreement** according to the El Emam Benchmark - see "SPICE Software Process Assessment Kappa benchmark" as introduced in[3]

Table 6. Agreements between developers and classification tool for the evaluation round two

	Automated Classification			
Developers	**Adap.**	**Corr.**	**Perf.**	
Adaptive	**12.0**	1.5	0.0	13.5
Corrective	3.5	**19.0**	1.0	23.5
Perfective	8.5	4.5	**24.0**	37.0
	24.0	25.0	25.0	**55.0**

Table 7. Comparison of evaluation rounds one and two

	Agreement			
Round	**Adap.**	**Corr.**	**Perf.**	**Kappa**
Round 1	11	12	24	0.46
Round 2	12	19	24	0.61

5 Related Work

In 2000 Mockus and Votta presented a study [9] that followed an approach similar to this work. They propose the importance of a textual description to understand the reasons behind software changes. The evaluation performed in our work strongly suggests the truth of that statement. They further state that other factors might also influence the change classification. We aim to find new classification rules to improve the classification algorithm. The classification algorithm and the dictionary that we developed solely focus on the "textual description field" but our implemented tool Subcat was built already keeping in mind an extension of the classification algorithm also involving other aspects, such as size of the commit, measured in changed lines of code, or interval.

In 2008 German and Hindle released a study about the taxonomy of large commits [6]. They define large commits as commits that include a large number of files. In their study, they manually classified large commits from nine open source projects by their intentions. They started by extending Swanson's categories by the categories "implementation" and "non functional". During their work they observed that these categories did not suffice for the categorization of the intention of large commits and developed a new set of categories which they call the "Categories of Large Commits". In [7] , Kemerer and Slaughter presented a set of methods and techniques to study software evolution. Bevan et al. introduced a system called Kenyon [1]. They imply resource intensive logistical constraints, e.g. the extraction of analysis specific facts, the storage of the results of the extraction. These tasks have to be performed for each change or configuration separately. Kenyon is designed to support these logistical tasks. It provides support for different software configuration management systems and retrieves consistent source code configurations. This issue is solved by implementing different plugins for every software configuration management system.

The plugins themselves are very lightweight because they can all utilize the same libraries, that hold the core functionalities.

A good overview and description of existing change metrics for commits and their different subtypes is provided by German et al. in [4]. They also presented a framework for the classification of change metrics. They present five different groups of change metrics: entity change metrics, MR-scoped change metrics, time-based change metrics, event-triggered change metrics and change metrics that do not measure code. Additionally they divide the change metrics into modification-unaware and modification-aware metrics. The classification presented in this paper can be the basis for the computation of related change metrics. The software that we present, in a first step is only capable of classifying changes using the classification algorithm. In the future the software can be extended in a way that it automatically calculates some of the metrics presented by [4].

6 Conclusion

The presented work provides a tool, Subcat, and a dictionary for cross-project analysis of software evolution based on an automated classification. To achieve these two goals, we completed the following tasks:

Classification Algorithm. We developed a classification algorithm which uses a dictionary as its base of decision-making. The classification algorithm uses a set of commits as its input and returns an assignment between the commits and the categories that are defined in the dictionary. It is based on the analysis of the natural language in the commit messages and follows a lexical approach.

Dictionary. We presented a dictionary for our classification algorithm that is capable of assigning commits to Swanson's maintenance categories. The cross project validity of the dictionary has been proven on five different open source software projects. We instantly reached a percentage of successfully classified commits of over 80% for each of the projects, without having to adopt the dictionary.

Evaluation. We evaluated the dictionary and the automated classification by using a two-step evaluation process. In a first step we evaluated the decisions of the automated classification and the dictionary against our own manual classification. We reached an average recall of **0.93** and an average precision of **0.86**. In the second evaluation step we evaluated the dictionary against the opinion of five professional software developers. We have proven a **moderate agreement** between the decisions of the automated classification and the decisions of the developers. This result is similar to the result achieved by Mockus et al. in [9] and proves that the approach presented is valid for cross-project analysis in the open source project landscape.

Subcat. We developed a command line tool, which implements the classification process. We defined a generic input format for the commits, to ensure the reusability for data, extracted from different VCS. The command line tool delivers the results of the classification as CSV-based reports. We presented an

Eclipse plugin which integrates the automated classification directly into the Eclipse workbench. It provides easy access to the classification functionality and includes a rudimentary visualization of the results of the classification.

The successful evaluation of the lexical-approach on generic open source projects has many implications. Researchers can use Subcat for a new definition of maintenance and software evolution metrics, not only in open-source projects, but also in any project using non-obscure commit messages. Or fellow researchers can use Subcat to comfortably analyze for example developer profiles over time in open source projects, e.g. which developer does the bug fixing, who is implementing the new features. Subcat can not only be used for profiling or software evolution metrics though. Additionally, Subcat has been adapted to work with GIT repositories recently.

Additionally to the main purpose, the categorization of changes into software maintenance categories, Subcat can be easily adapted (by changing the dictionary) for any other studies on commit messages in repositories. The author and dictionary report and to some extent the lemma report are especially useful for this purpose.

6.1 Discussion

There are many extensions to Swanson's classification of maintenance tasks. In our work, we adhere to Swanson's original category set, because it is manageable and well defined. During our research we studied a lot of commits and recognized that often, one commit holds multiple, non-associated changes that would have to be assigned to different categories. This indication warrants further research. In the survey based evaluation we used Cohen's Kappa as a measure. The definition of a nominal scale implies that every item can be classified to exactly one category. As indicated earlier, there are cases where a commit can not be distinctly classified to one category but includes different activities that stretch between two or even all of the three maintenance categories.

For detailed results per project, please contact the authors at the given mail address.

6.2 Future Work

In this paper we use WordNet solely for matching words and lemmas. WordNet also possesses the possibility for cognitive matching which could be included in the matching algorithm. Furthermore, Subcat is capable of measuring further details of the commits (e.g. commit size, length of commit messages). These parameters provide possibilities for further tuning of the categorization mechanism.

References

1. Bevan, J., Whitehead Jr., E.J., Kim, S., Godfrey, M.: Facilitating software evolution research with kenyon. In: Proceedings of the 10th European Software Engineering Conference, pp. 177–186 (2005)

2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns, vol. 1. John Wiley and Sons (1996)
3. Emam, K.E.: Benchmarking kappa for software process assessment reliability studies. Empirical Software Engineering 4, 113–133 (1999)
4. German, D.M., Hindle, A.: Measuring fine-grained change in software: Towards modification-aware change metrics. In: Proceedings of the 11th IEEE International Software Metrics Symposium, p. 28 (2005)
5. Hassan, A.E.: Automated classification of change messages in open source projects. In: Proceedings of the 2008 ACM Symposium on Applied Computing, pp. 837–841 (2008)
6. Hindle, A., German, D.M., Holt, R.: What do large commits tell us? a taxonomical study of large commits. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 99–108 (2008)
7. Kemerer, C.F., Slaughter, S.: An empirical approach to studying software evolution. IEEE Transactions on Software Engineering 25 (1999)
8. Landis, J.R., Koch, G.G.: The measurement of observer agreement for categorial data. Biometrics 33, 159–174 (1977)
9. Mockus, A., Votta, L.G.: Identifying reasons for software changes using historic databases. In: Proceedings of the International Conference on Software Engineering, pp. 120–130 (2000)
10. Swanson, E.B.: The dimensions of maintenance. In: Proceedings of the 2nd International Conference on Software Engineering, ICSE 1976, pp. 492–497 (1976)

Cohesive and Isolated Development with Branches

Earl T. Barr[1], Christian Bird[2], Peter C. Rigby[3], Abram Hindle[4],
Daniel M. German[5], and Premkumar Devanbu[1]

[1] UC Davis, Davis CA, USA
[2] Microsoft, Redmond WA, USA
[3] McGill University, Monteal QC, Canada
[4] University of Alberta, Edmonton AB, Canada
[5] University of Victoria, Victoria BC, Canada

Abstract. The adoption of distributed version control (DVC), such as Git and Mercurial, in open-source software (OSS) projects has been explosive. Why is this and how are projects using DVC? This new generation of version control supports two important new features: distributed repositories and histories that preserve branches and merges. Through interviews with lead developers in OSS projects and a quantitative analysis of mined data from the histories of sixty project, we find that the vast majority of the projects now using DVC continue to use a centralized model of code sharing, while using branching much more extensively than before their transition to DVC. We then examine the Linux history in depth in an effort to understand and evaluate how branches are used and what benefits they provide. We find that they enable natural collaborative processes: DVC branching allows developers to collaborate on tasks in *highly cohesive* branches, while enjoying *reduced interference* from developers working on other tasks, even if those tasks are *strongly coupled* to theirs.

1 Introduction

Version control (VC) is tool support for concurrent, collaborative software processes. VC allows developers to create a *branch*, an isolated workspace, from a particular state of the source code. They can share this branch and work on their tasks within it without impacting the rest of the project and later merge (or integrate) their changes back into the main line of development.

Intuitively, branches should be *cohesive* (*i.e.* collect related changes [26]) allowing a team to work together on a focused task and *isolated* from the rest of the project so that rapid and volatile development is not interrupted or impacted by external changes. The rich history provided by recent VC and their adoption by a number of projects provide a unique opportunity to address these intuitions and quantitatively measure how cohesive and isolated branches are in practice.

The evolution of VCs is marked by *increasing fidelity of the histories they record*. A *commit* is the write of a change into VC history. First generation VC, such as RCS, record the history of individual file commits. This enabled rolling back changes to a single file and reviewing file-specific changes. Second generation, or centralized VC

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 316–331, 2012.

(CVC), such as Subversion, stored sets of file changes committed together (*i.e.*, a *changeset*) in its history. This allows a related set of changes to be rolled back, and also enables the conjoint history of a set of related files to be reconstructed.

Recently, a new generation of VC, distributed version control (DVC), has transformed the use of VC and has achieved widespread adoption. In DVC, every copy of a project is a repository, with its own history and the power to exchange source code changes with other repositories. In contrast with CVC, DVC is distributed in the sense that it allows the change of changesets unmediated by a central repository. DVC also preserves the history of branches after their promotion into the mainline of development. Consider Fig. 1 in which circles represent commits to the repository. Arcs denote the temporal ordering of commits. "Mainline" denotes the main line of development from which releases are made and to which features, like "my-branch", are merged. The dashed edges depict relationships that were untracked, and forgotten in CVC [1]. In DVC, a commit always tracks its immediate predecessor commits, across both branches and merges; for DVC, the dashed edges are indistinguishable from the edges along a branch. This branch history allows us to augment developer studies with quantitative studies of branch cohesion and isolation. We can use this branch history to crosscheck qualitative results on branch usage. We can also use these measures to shed light on whether differences in how a project uses branches correlate with defect rates or schedules delays.

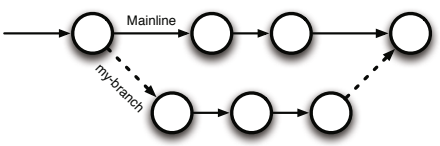


Fig. 1. DVC history preserves branches and merges

Open-source software (OSS) projects have rapidly adopted DVC. Our first research question, RQ1, asks "Why did OSS projects rapidly adopt DVC?" We use interviews to show that developers had previously wanted to make heavier use of branches but were dissuaded by "merge pain", the difficulty of resolving conflict that arises during branch integration, and buttress this observation by showing that branch usage has markedly increased in those projects that made the transition from CVC to DVC. We also note that almost all projects making the switch have continued to use a centralized repository, calling into question the conventional wisdom that DVC's support for distributed workflows has been the principal cause of the rapid transition to DVC.

Without branches, developers must share a single mainline and deal with the conflicts that sharing entails. In practice, projects developed workflows to avoid or mitigate conflict, such as baton passing or the "commit bit". We can demonstrate the benefit of branching by simulating a lack of branching. We observe that branched history of a DVC can be linearized onto a single "mainline" in which the conflicts and interruptions that branching avoids become manifest. This linearized history overapproximates the actual conflict and allows us to bound the cohesion and isolation that branches afford.

Ideally, when a task is identified, developers create a branch to work on the task together. But does this occur in practice? Is work performed in a branch more cohesive

[1] This limitation was addressed in Subversion 1.5.

than all changes across the repository during the same time period? Thus, RQ2 is "How cohesive are branches?" To investigate this question, we use directory distance of the files modified in a branch to measure its cohesion. Then we compare actual branches in the Linux history against the baseline, background cohesion of linearized sequences of commits. If actual branches are no more cohesive than these commit sequences, then branches are either unlikely to be cohesive or directory distance is a poor proxy for branch cohesion. To form these commit sequences, we picked a random starting point on the linearized branch history. In §4.2, we found that actual, observed branches are significantly more cohesive than background commit sequences.

RQ3 asks "How successfully do DVC branches protect developers from interruption?" VC is good about flagging syntactic conflict; *semantic conflict* occurs when mainline has changed in such a way as to invalidate assumptions made during the development of a branch. Cross branch coupling causes semantic conflict. To merge a feature branch into mainline is to *promote* that branch. When promoting a branch, programmers must review mainline to try to find semantic conflict. To measure semantic conflict, we measure the number of commits in a branch being considered for promotion that modified a file that has also been modified in mainline, since the branch forked from mainline. Against a linearized DVC history, we measure and bound how often the semantic conflict would interrupt a developer in the absence of branching or procedures to ameliorate it.

We make three principal contributions in this paper: 1) We present compelling evidence from study of sixty projects (RQ1) that branching and not distribution has driven the rapid adoption of DVC; 2) We define two new measures: branch cohesion and distracted commits, a type of task interruption that occurs when integration work intrudes into development; and 3) We apply these measures to the Linux history and (RQ2) quantify the cohesiveness of branches and (RQ3) the effective isolation they provide against the interruptions intrinsic to concurrent development.

2 Theory

In April, 2005, development simultaneously began on two open source DVC systems, Git and Mercurial. Their popularity has exploded, and by 2011, a large portion of open source projects have already migrated to a DVC. According to Debian (a Linux distribution), of the 55% of projects that report their VC (9,132 projects), 44% (3994 projects) use DVC [33], indicating that it has achieved widespread acceptance and adoption[2]. VC has a profound effect on workflow, and adoption of a new VC is not a trifling matter [12], as evidenced by the amount of discussion surrounding decisions to change, the work required to move from one to another, and the change in project workflows, all of which we have observed in OSS projects. For examples see GNOME's move to Git [25], Python's move to mercurial [7], and the project that KDE created solely to evaluate and eventually create tools for a migration to Git [13].

[2] The data we report here comes from the repository that contains the Debian packaging scripts. In practice, we observe that for the majority of projects, this repository is indistinguishable from the upstream repository.

Research Question 1: Why did OSS projects rapidly adopt DVC?

In §4.1, we present compelling evidence that DVC support for branching drove the transition to DVC. Our interviews show that the impetus is cohesion and isolation. But how cohesive are branches and how well do they isolate developers?

Cohesion. If developers use branches to isolate tasks, we expect to find that branches are cohesive and encapsulate related changes. Two reasons to expect developers to work with cohesive branches is that their histories are easier to understand when faced with maintenance tasks and they are easier to revert if the branch has a problem. On the other hand, developers could be using branches merely to isolate their development work, without separating that work into cohesive tasks.

Research Question 2: How cohesive are branches?

Coupling and Interruption. Developing a new feature often requires making changes to modules that are coupled to other modules. If different features, under simultaneous construction by different developers, affect coupled modules, the tasks may require coordination, as one developer's work can cause other developers' code to become unstable. Ideally, uninvolved developers should be isolated from these changes until the feature has achieved some degree of stability. At the same time, a developer working on a new feature should still have access to VC to commit incremental changes, and rollback, as necessary. Berczuk [2] makes this point in his discussion of *configuration management patterns*, where he argues that developers should checkpoint changes at frequent intervals to a location separate from the "team version control," and that only tested and stable code should be integrated. When the feature is ready, its integration must not be too difficult or the productivity gained from working on an isolated branch is lost. Indeed, Perry *et al.* [24] claim that tool support for integration is important because "integration too often is painful and distracting" and because development lines diverge when parallel development goes on too long.

When branches are not used, all changes occur on the mainline and a developer may need to merge and integrate changes that are unstable and transitory or only tangentially related to her work. The attendant interruptions can slow development. The use of branches allows a developer to control and minimize the frequency of such interruptions.

Integration interruptions are a form of task interruption. Prior literature has shown that task interruptions seriously impact developer productivity. Recovering from interruptions can be difficult and time-consuming: developers must mentally juggle goals, decisions, hypotheses, and interpretations related to their task, or risk inserting bugs. In a study at Microsoft [16], 62% of developers said that recovering from interruptions is a substantial problem. Van Solingen [28] found that interruptions are most problematic when a developer is checking in changes or updating their working code base. DeMarco observed that resuming after an interrupt often takes at least 15 minutes [10].

Parnin *et al.* [22] instrumented Visual Studio and Eclipse to observe the time taken to resume development tasks. While they found some strategies for mitigating the effects, developers began editing within a minute of restarting a task only 10% of the time and took over 30 minutes in 30% of the cases. While these papers consider the effect of interruptions in broader terms, they do support the claim that task interruptions diminish productivity.

Research Question 3: How successfully do DVC branches protect developers from interruption?

3 Methodology

We used a mixed method research strategy [9] in our study of branches in DVC. We began with interviews of developers (the qualitative phase) to help develop hypotheses regarding the motivations for DVC adoption and then empirically evaluated these hypotheses by gathering data and performing statistical analyses (the quantitative phase). For us, the advantage of a mixed method approach is that the qualitative investigation allowed us to collect answers to fundamental questions related to the "how" and "why" of DVC adoption. The answers then provided insight and added meaning to our quantitative results that might otherwise have been missed in a purely quantitative study. This increased our confidence in the findings and provided a richer context that can aid in understanding whether our results generalize.

In an effort to understand what has motivated the rapid transition to DVC from an operational point of view, we observed the development activities in projects that switched to DVC and interviewed a number of lead developers from these projects regarding their switch. We sent personalized requests for fifteen minute interviews to the three most active developers in a number of large and mature projects that had used CVC for multiple years and had recently moved or decided to move to DVC. Following these interviews, we gathered data from the development history of these projects and quantitatively evaluated hypotheses based on their responses.

Interviewing project leaders was critical in understanding why people switched to DVC, the perceived benefits and drawbacks of the switch, and (in cases where the projects have used DVC for some time) how it has affected the policy and development process of the projects. The data mining of the VC history and developer mailing lists allowed us to provide quantitative evidence of the effects of DVC. We interleave quotations from interviews and numerical findings from data mining to triangulate and provide a balanced perspective.

We conducted semi-structured interviews of four projects and six people. Semi-structured interviews make use of an *interview guide* that contains general groupings of topics and questions rather than a pre-determined exact set and order of questions [17]. Semi-structured interviews are often used in an exploratory context when there are clear research questions [17,31]. The responses from these interviews help develop hypotheses and focus quantitative analysis. We

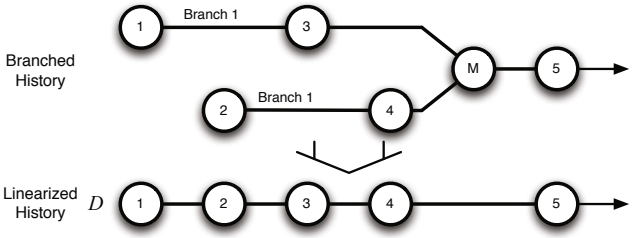


Fig. 2. Branches projected onto D, a single timeline by date. The merge change M that joins the two branches falls out since the work to merge each change occurs, piecemeal, as each change is recorded.

extracted themes from the interviews using a modified version of Creswell's guidelines [9] for coding. The interview guide that we used can be found at `http://www.cabird.com/public/vcinterviewquestions.pdf`[3]. We minimally copy-edited the quotes for readability. We eliminated false starts and superfluous crutch words; we used standard notation, delimiting clarifying comments with brackets and marking the suppression of unnecessary phrases with an ellipsis [17].

For the quantitative mined data, we developed measures and modified existing ones to best examine the impact of DVC in the context of our dimensions. The data used, the definition of the measure, and attendant threats to validity are discussed in §4. We chose to examine 60 projects that had transitioned to DVC. These projects were drawn from lists of projects using DVC on Wikipedia and GitWiki and include such notable projects as Wine, Samba, Perl, Ruby on Rails, and the Glasgow Haskell Compiler. These projects vary in age from 21 years (in the case of Perl) to 6 months (pthreads-stubs in X.Org) with a median of 4.5 years. The number of contributors as recorded by the repositories ranges from 1462 (Wine) to 1 (dri2proto in X.Org). The commits to these projects number from 139,187 (Samba) to just 6 (pthread-stubs in X.Org). As such, our selection of projects for analysis spans a broad spectrum of OSS projects in terms of size, age, and development activity. All projects have used DVC for at least 5 months at the time of this study; the majority of them for over one year.

We use Linux to evaluate hypotheses and questions regarding advanced DVC usage because the Linux kernel project has never used a CVC and its developers are generally very experienced with history-preserving branching. Linux started using Git in 2005; we have 3.5 years of Linux VC data and the corresponding data from Linux kernel Mailing List (LKML). Over this period, there were 4K developers, 118K commits, and 443K mail messages for Linux.

4 Evaluation

In this section, we answer each of our research questions. To begin, we introduce our branch linearization technique on which much of our analysis rests. To linearize

[3] At the request of the participants, the interviews in their entirety are confidential.

a branched DVC history, we project the concurrent sequence of changes in a DVC history onto the single timeline D, as shown in Fig. 2. The commits along this timeline represent concurrent work that actually occurred *across* branches. Conflict or interruption, that occurs along this timeline, bounds the work needed to avoid conflict or recover from interruption. This work was previously largely unobservable (apart from mutterings in mailing lists/interviews/change-log messages), handled by policies and procedures such as baton passing and patch rework on a project's mailing list [32]. To measure the cohesion (§4.2) and isolation (§4.3) of branches, we compare the cohesion and isolation of their *within* branch changes against that of *across* branch changes, in the form of simulated branches drawn from D.

4.1 Rapid DVC Adoption

Pundits claim that support for distributed (changeset flows unmediated by a central repository), as opposed to centralized, development is the root cause of this rapid transition [23,8]. We have observed something different. The vast majority of these projects do not appear to be making use of distribution. Of the sixty projects whose VC use we examined, all but Linux continue to use a centralized model organized around a single public repository, except the `xemacs` and `gnome` projects which publish two repositories. Although these projects continue to use a centralized style of development, we *have* observed a dramatic shift in their use of branches.

Lead developers from prominent open source projects (§3) indicated that, prior to using DVC, branches were "painful and difficult" to integrate:

> " The biggest complaint associated with Subversion is associated with branching and merging. The one feature that Git has that our users would really like is a really fast and simple merge. "
>
> Richards, CEO WANdisco [14]

In some cases, two branches would grow so far apart, they had to abandon one of them altogether. Prior to DVC, branches were typically created only for releases and *not* new features. For instance, Koziarski from Ruby on Rails states: "We had branches for versions [releases]. Feature branches were very rare for us"[20]. A preliminary empirical investigation showed that few branches were created pre-DVC. Of the examined 60 projects that switched to DVC, 1.54 branches were created on average per month per project before using DVC; after switching to DVC, the average rose to 3.67. A Wilcoxon rank sum test shows that the two populations are statistically different[4]($p \ll 0.01$)

Without easy branch and merging facilities, our interviewees reported that developers would "pass around large patch sets" or "brain dump" a mega-patch that was almost impossible to review. These large patch sets contained multiple, sometimes unrelated changes, and it was impossible to "consider each on their own merits without having to swallow the whole thing" (Turnbull, XEmacs [27]). This problem was compounded for

[4] A Wilcoxon test was used rather than the standard t test due to the heavily skewed distribution of branches.

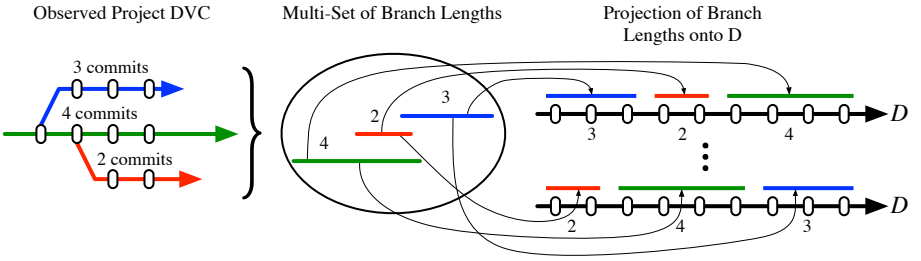
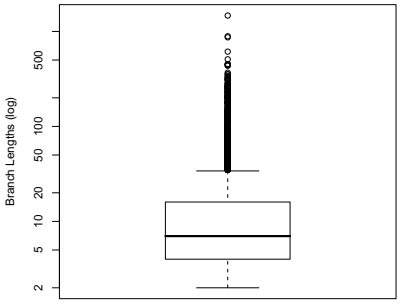
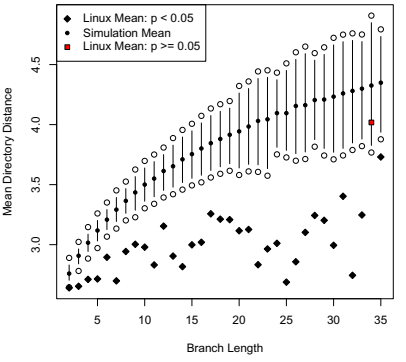


Fig. 3. Depiction of the selection of branches for the Monte Carlo simulation



(a) Distribution of branch lengths in the Linux kernel.



(b) Observed branches compared to simulated branches over D from 1,000 simulations.

Fig. 4. Linux branch lengths: observed and simulated

new developers who did not have commit access and so could not work and commit incremental work in the course of making large changes. Under CVC, developers without commit privileges, as well as core developers who refused to use "painful" (Sperber, XEmacs [21]) feature branches were effectively reduced to working in a time before version control.

> " Because we'd have these large changes that would go in all at once, it would be really difficult to find the source of problems. For example, if you wanted to find a change that was responsible for certain problems, you would often go back [in history] . . . and pretty soon you'd find one of these 'mega' patches . . . that would essentially change every file in the system and would lump together sets of unrelated changes ... [these mega changes made it] really, really difficult to track down what change was responsible for a given problem, it makes software maintenance really difficult. "
>
> Sperber, XEmacs [21]

In summary, projects continue to use a centralized repository and project maintainers have stated that the DVC branch and merging facilities was a principal motivation, so we find that the answer to RQ1 is branching, not distribution.

4.2 Cohesion

Large systems, like the Linux kernel, structure their files in a modular manner. Files that perform similar or related functions are close in the directory hierarchy [5], thus the directory structure loosely mirrors the system architecture. To determine how "cohesive" a set of changes is, we measure how far source files are from each other in the directory tree. Two files in the same directory have a distance of zero (*i.e.* the highest level of cohesion), while the distance for files in different directories is the number of directories between the two files in the hierarchy. We only include '.c' source files as Bowman [5] found that header files for the entire system often are located in one directory.

Let $d : F \times F \to \mathbb{N}_0$ denote the directory distance of two files. Each commit defines a set of modified files, or changeset. When F is the set of files in a source code repository and C is the set of commits, $f_m : C \to (2^F - \emptyset)$ returns the changeset of a commit; f_m cannot return the empty set because a changeset cannot be empty. The cohesion of a single commit is the multiset of directory distances formed from the files in its changeset. A branch is a "straight line" sequence of commits, $B = c_1, \cdots, c_n$, where c_1 is not a merge commit and c_n is either a leaf (*i.e.* HEAD) or the parent of a merge commit. Thus, one branch includes and continues through a branch commit, while each child of a merge commit starts a new branch, rather than continuing one of the merged branches. For the branch B, let B_d be the multiset of directory distances formed over the union of all its changesets:

$$B_d = \{d(f, f') : f, f' \in \bigcup_{c \in B} f_m(c)\}, \text{ for } f \neq f'. \tag{1}$$

Definition 4.1 (Branch Cohesion). *The* branch cohesion *of B is the average of the directory distances in B_d:*

$$B_c = \sum_{d \in B_d} \frac{d}{|B_d|}.$$

To determine if developers use branches to isolate cohesive changes, we need a baseline to compare the cohesion of branches because we have no *a priori* notion of what the range of good and branch cohesion values may be. Thus, we need to establish the background distribution of cohesion, as a baseline for comparison. To do so, we measure the cohesion of branches over D, the linearized history of a project (Fig. 2), which captures concurrency work as a free-for-all on a single, shared mainline. Specifically, we compare the cohesion of observed branches in the history of the Linux kernel against the cohesion of simulated branches of equivalent length over the linearized history, D, using Monte Carlo simulation. Fig. 3 depicts this simulation. We first measure the length of each branch in the observed Linux kernel history (Fig. 3 left) and extract their multiset of branch lengths (Fig. 3 middle). We then randomly tile these branch lengths (which

do not contain merge commits and sum to *precisely* the length of D) onto D to form simulated branches (Fig. 3 right). Thus, the distribution of branch lengths is exactly the same as the observed distribution of branch lengths in the Linux kernel history; specifically, this is the distribution shown in Fig. 4(a). We then compute the branch cohesion for each simulated branch. If developers generally work together on cohesive sets of files in branches then the branch cohesion for branches of length n in the observed DVC history will be higher than the cohesion for sequences of commits with length n in D. We generated 1000 tilings in our simulation.

Fig. 4(a) is a boxplot of the lengths of observed branches in the history of the Linux kernel. As Fig. 4(a) makes evident, the distribution is positively-skewed. Since 90% of the Linux kernel branches have length less than 35 commits, we truncated Fig. 4(b) at 35. Branches longer than 35 commits had fewer than 25 instances, giving too small a sample to produce meaningful results. Fig. 4(b) plots the mean branch cohesion of observed Linux kernel branches (black diamonds) against the mean of the means of the cohesion of the simulated branches (black circles). We report the mean of the means at each branch length for the 1000 tilings and provide a 95% confidence interval (the vertical lines). With the exception of branch length 34, which is not statistically significant (red square), the observed branches are more cohesive than the simulated branches at each length with $p < 0.05$.

Examining the magnitude of the differences in cohesion, we see that at branch length two (the minimum), pairs of files committed on observed branches are 0.12 directories closer together on average than pairs of files along D, the linearized history, while the difference is 1.5 directories at branch length 32 (the maximum). These differences may appear small, but note that a difference of 1 means that for *each pair of files* the distance between them is at least one directory further apart in the code base on a simulated branch than on the observed branch. This effect looms larger when one recognizes that most branches modify tens to hundreds of files.

This point is further underscored by correlating this difference to the branch length. As can be seen from Fig. 4(b), as branches become longer, the observed branches become increasingly more cohesive relative to the simulated branches (Spearman correlation: $r = .69, p \ll .001$). It is clear that developers group related changes on branches and that this grouping increases with the number of changes.

Our interviews are consonant with this result: branches are not created only for releases. In projects that have moved to DVC, branches comprise non-trivial, cohesive changes such as features or localized bug fixes and maintenance efforts. Three of our interviewees indicated that previously, such non-trivial changes would either have been avoided or created "off-line" and then committed to the VC in a single, disruptive mega-commit. Thus, we find that the answer to RQ2 is that branches are highly cohesive.

4.3 Coupling and Interruptions

Using data mined from the Linux kernel, we construct its linearized history D, as defined in Fig. 2, and quantitatively establish an upperbound on the number of integration interruptions that a developer avoids through the use of branching. By analogy to numeric intervals, $D(x, y)$ denotes the subsequence of commits between x and y in D. For the commit c, let a denote its most recent non-merge ancestor.

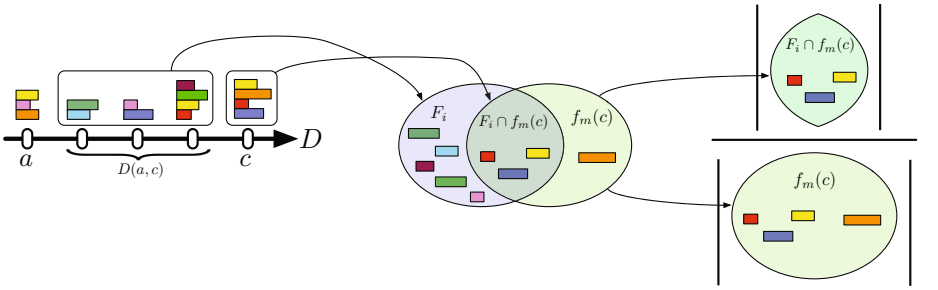


Fig. 5. Depiction of the formalisms described: The straight line indicates D, the linearized sequence of commits (ovals). The stacked colored rectangles above a commit represent its changeset. Here, c is a commit whose nearest, non-merge ancestor in DVC is a. $D(a, c)$ are the commits made by other developers in the intervening time. $F_m(c)$ is the set of files modified in c and F_i is the set of files modified in the commits in $D(a, c)$. The ratio of files in the intersection to files changed in c is the *index of similarity* δ that we vary in our definition of distraction.

Consider a developer working on a new feature on a branch. When she promotes a feature branch to master, she must not only resolve any syntactic conflict that arise, but, more generally, look for potential *semantic conflicts*, conflicts that occur when mainline changes in a way that violates the assumptions on which a feature branch rests. For instance, her branch may rely on a global variable whose range of allowed values has changed in master, because her branch is coupled to other branches promoted since her branch began. Such verification can be subtle and time-consuming. This work is inherent to concurrent development, but previously handled out-of-band by policy and procedure. To upperbound this work, we consider the work to search for semantic conflict that would occur along D where the distraction of integration work potentially intrudes into feature development work at each commit. This measures how often the integration work, ideally deferred to merge time, would instead intrude into feature development in the absence of an isolation mechanism, such as that provided by DVC.

Fig. 5 illustrates the formalisms we introduce to measure the integration interruptions that occur along D. The line at the left represents D, the linearized history. Ovals on D represent commits. Each commit c defines a changeset, a set of files that it modifies. In the figure, these modified files are the rectangles stacked above each commit. Specifically, c is a commit whose nearest, non-merge ancestor in the original DVC history is a, and $D(a, c)$ represents the commits, not including a or c, that developers made to other branches in that history in the intervening time. Definition 4.2 formalizes the set of files changed in a sequence of commits.

Definition 4.2 (Intervening Files). *The files modified in $D(a, c)$ "intervene" between c and a, its nearest, non-merge ancestor in D. These files therefore change the state of the project into which c is written. The set of* intervening files *is*

$$F_i = \bigcup_{w \in D(a,c)} f_m(w).$$

If c modifies $f \in F_i$, a syntactic or semantic conflict could occur. Semantic conflicts can be more distracting than syntactic conflicts as c's author must review each file in $f_m(c) \cap F_i$ to ensure their absence, since VC catches syntactic conflicts. For instance, one of the commits in $D(x, c)$ could have changed the semantics of a function used in c. Intuitively, the commit c is *distracted* if commits fall between it and its nearest, non-merge branch ancestor on D and one of those intervening commits changed a file that c also modified. In Fig. 2, all the commits except commits 1 and 5 are potentially distracted, depending on the set of files each commit changes. Definition 4.3 captures this intuition.

Definition 4.3 (Distraction). *The commit* $c \in D$ *is* distracted *if*

$$\frac{|F_i \cap f_m(c)|}{|f_m(c)|} > \delta, \textit{ for } \delta \in [0..1].$$

We cannot know how often files changed in both c and $D(a, c)$ will actually cause a conflict or require the developer committing c to understand a change that occurred in $D(a, c)$. We capture this uncertainty in the threshold δ, an *index of similarity*, or fraction of the size of the intersection of c's changeset and the changesets in $D(a, c)$ over the size of c's changeset. Each setting of δ represents a different assumption about how likely concurrent changes are to generate integration work in order to write the current changeset and form the commit c. At the right of Fig. 5, the fraction of the number of files in the intersection divided by the number of files in c pictorially depicts this index of similarity that we use to measure integration interruptions.

In Fig. 6, we plot the proportion of commits in the linearized history of the Linux kernel that are distracted as δ varies. At zero, we print the percentage of the time there are intervening files ($F_i \neq \emptyset$), regardless of whether they intersect with c's changeset. Even at $\delta = 1$, *i.e.* when we require $f_m(c) \subseteq F_i$, 2.8% commits are distracted, *i.e.* may encounter conflict or require review to ensure that no semantic assumption have been violated. After calculating the 95% confidence intervals, we find that a commit c modifies a file that intervenes between c and its ancestor a on D with a confidence interval of 4.47% to 4.69% of the time. This corresponds to the point in Fig. 6 with an

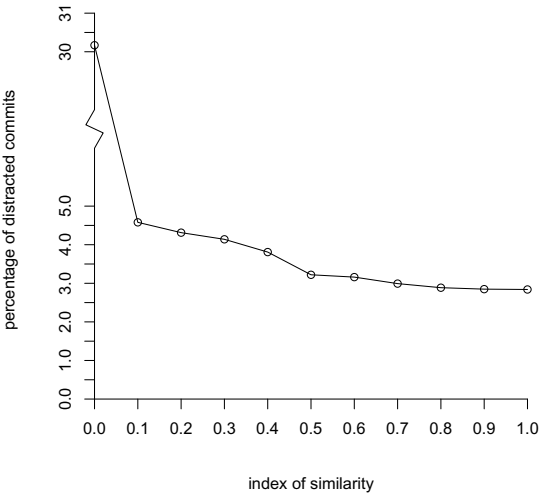


Fig. 6. Commits that require integration work as δ varies when the Linux kernel history is linearized

index of similarity of 0.1. All of the files in the changeset of a commit c are distracted (index of similarity 1.0) with a confidence interval of 2.47% to 2.93%. Thus, a non-empty overlap occurs approximately once every 22 commits and a complete overlap every 35 commits.

Clearly, using a branch reduces distractions by delaying the need to resolve conflicts until merging the branch back into its parent. But how often does the use of branching actually avoid potential distractions in practice? Quantifying exactly how much distraction is avoided depends on how likely it is for concurrent changes to a single file to generate integration work. First, there is the rate, reported above, of non-empty intersection. That is, how often concurrent edits on different branches touch the same file. Second, there is the cardinality of that intersection; how many files are edited concurrently by different branches. Finally, there is the probability that the concurrent changes to a file in different branches actually generate integration work, at the very least in the form of confirming the changes made to the file are semantically noninterfering. We have established that on average, non-empty intersections occur once in every 22 commits. To be conservative, we assume that these intersections contain only a single file and that 90% of the time the programmer must examine the out-of-branch change made to it. To answer RQ3, we therefore conclude that working on branches protects a programmer from unexpected, unwanted semantic conflicts once in every $24.\bar{4}$ commits on average, across all branches that a developer works on.

4.4 Threats to Validity

The main threat to the external validity of our cohesion and distractions results is their dependence on Linux Git history, which may not be representative. Further, Git history can be perfected via "rebasing", an operation that allows the history to be rewritten to merge, split or reorder commits [3]. Repositories hosted locally by developers are also not observable until branches are merged elsewhere.

Many projects we surveyed did not have a long enough DVC history (*i.e.* sample-size) to produce statistically significant results in all of our measures. Developers are still adjusting to DVC and may not have adopted history-preserving branching to break apart larger commits. As well, many contributions, even to DVC-using projects, are still submitted as large patches to the mailing-list, diluting, at least in the short term, the impact of DVC adoption.

D, the linearization of a DVC history that projects all branches onto a single, shared mainline overapproximates the integration interruptions faced by a developer, but we do not know by how much. Our use of directory distance as a cohesion measure does not capture the cohesion of a cross-cutting change; however, the fact that we found a significant difference in spite of understating the history-preserving nature of lightweight branching strengthens our result. Our analysis assumes that all integration interruptions waste time, which may not always be the case.

5 Related Work

Version control systems have a long and storied past. In this paper, our concern is primarily the introduction of history-preserving branching and merging, and the resulting

rich histories. The importance of preserving histories, including branches, has been well recognized [11]. The usefulness of detailed histories for comprehension [1] and for automated debugging [34] are by now well accepted. Some have even advocated very fine-grained version histories [18] for improved understanding and maintenance. Automating the acquisition of information, such as static relationships or why some code was commited, from accurate and rich VC history might improve developer productivity [15].

Branching in VCs have received a fair bit of attention [11]. Some have recommended "patterns" of workflows for disciplined use of branching [29]. Others advocate ways of branching and merging approaches [6] that mitigate the difficulties experienced with the branch and merge operations of earlier version control systems. Merging is a complex and difficult problem [19], which, if anything, will become more acute as a result of the transition to DVC and the corresponding surge in the use of branching we have shown. Bird *et al.* [4] developed a theory of the relationship between the goals embodied by the work going on in branches and the "virtual" teams that work on such branches.

Perry *et al.* [24] study parallel changes during large-scale software development. They find surprising parallelism and conclude "current tool, process and project management support for this level of parallelism is inadequate". Their conclusion anticipates the rapid transition to DVC that we chronicle in this paper.

The influential work of Viégas et al [30] uses a visualization methodology to study the historical record of edits in Wikipedia, and report interesting patterns of work (such as "edit wars"). To our knowledge, our paper is the first detailed study of the impact of DVC and its history-preserving branching and merging operations on the practice of large-scale, collaborative software engineering.

6 Conclusion and Future Work

Contrary to conventional wisdom, branching, not distribution, has driven the adoption of DVC: most projects still use a centralized repository, while branching has exploded (RQ1). These branches are used to undertake cohesive development tasks (RQ2) and are strongly coupled (RQ3). In the course of investigating these questions, we have defined two new measures: branch cohesion and distracted commits, a type of task interruption that occurs when integration work intrudes into development.

We intend to investigate how projects select branches to merge. The isolation that branches afford carries the risk that the work done on that branch may be wasted if the upstream branch evolves too quickly. We intend to investigate the impact of history-preserving branching on the use of named stable bases [2].

References

1. Atkins, D.L.: Version Sensitive Editing: Change History as a Programming Tool. In: Deng, T. (ed.) ECOOP 1998 and SCM 1998. LNCS, vol. 1439, pp. 146–157. Springer, Heidelberg (1998)
2. Berczuk, S.: Configuration Management Patterns. In: Third Annual Conference on Pattern Languages of Programs (1996)

3. Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., German, D.M., Devanbu, P.: The promises and perils of mining git. In: Proc. 6th MSR (2009)
4. Bird, C., Zimmermann, T., Teterev, A.: A Theory of Branches as Goals and Virtual Teams. In: Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering (2011)
5. Bowman, I.T., Holt, R.C., Brewster, N.V.: Linux as a case study: Its extracted software architecture. In: Proc. ICSE 1999 (1999)
6. Buffenbarger, J., Gruell, K.: A Branching/Merging Strategy for Parallel Software Development. System Config. Management (1999)
7. Cannon, B.: PEP 374: Choosing a distributed VCS for the Python project (2009), http://www.python.org/dev/peps/pep-0374
8. Clatworthy, I.: Distributed version control — why and how. In: Open Source Developers Conference, OSDC 2007 (2007)
9. Creswell, J.: Research design: Qualitative, quantitative, and mixed methods approaches, 3rd edn. Sage Publications, Inc. (2009)
10. DeMarco, T., Lister, T.: Peopleware: productive projects and teams. Dorset House Publishing Co., Inc., New York (1987)
11. Estublier, J.: Software configuration management: a roadmap. In: Proc. of the Conf. on The future of Software Engineering. ACM (2000)
12. Jacky, J.-M.F., Estublier, J., Sanlaville, R.: Tool adoption issues in a very large software company. In: Proc. of 3rd Int. Workshop on Adoption-Centric Software Engineering (2003)
13. KDE. Projects/MoveToGit - KDE TechBase (November 2009), http://techbase.kde.org/Projects/MovetoGit
14. Kerner, S.M.: Subversion 1.7 released with some git-esque merging. developer.com (2011)
15. Ko, A.J., DeLine, R., Venolia, G.: Information needs in collocated software development teams. In: Proc. of the 29th ICSE. IEEE (2007)
16. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: a study of developer work habits. In: Proc. of the 28th ICSE. ACM (2006)
17. Lindlof, T., Taylor, B.: Qualitative communication research methods. Sage (2002)
18. Magnusson, B., Asklund, U.: Fine grained version control of configurations in COOP/Orm. Software Configuration Management, 31–48 (1996)
19. Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering 28(5), 449–462 (2002)
20. Koziarski, M.: Personal interview, April 5 (2009), rubyonrails.org
21. Sperber, M.: Personal Interview, April 3 (2009), xemacs.org
22. Parnin, C., Rugaber, S.: Resumption strategies for interrupted programming tasks. In: Proc. of 17th ICPC 2009. IEEE (2009)
23. Paul, R.: DVCS adoption is soaring among open source projects. ars technica, January 7 (2009)
24. Perry, D.E., Siy, H.P., Votta, L.G.: Parallel changes in large-scale software development: an observational case study. ACM Trans. Softw. Eng. Methodol. 10(3), 308–337 (2001)
25. Rocha, L.: GNOME to migrate to git (March 2009), http://mail.gnome.org/archives/devel-announce-list/2009-March/msg00005.html
26. Sarma, A., Noroozi, Z., Van der Hoek, A.: Palantír: raising awareness among configuration management workspaces. In: Proc. of 25th ICSE (2003)
27. Turnbull, S.: Personal Interview, April 7 (2009), xemacs.org
28. van Solingen, R., Berghout, E., van Latum, F.: Interrupts: just a minute never is. IEEE Software 15(5), 97–103 (1998)

29. Vance, S.: Advanced SCM branching strategies (1998), http://www.vance.com/steve/perforce/Branching_Strategies.html
30. Viégas, F., Wattenberg, M., Dave, K.: Studying cooperation and conflict between authors with history flow visualizations. In: Proc. of the SIGCHI Conf. on Human Factors in Computing Systems. ACM (2004)
31. Weiss, R.S.: Learning From Strangers: The Art and Method of Qualitative Interview Studies. Free Press (November 1995)
32. Weißgerber, P., Neu, D., Diehl, S.: Small patches get in! In: Proc. of the 2008 Int. W. Conf. on Mining Software Repositories. ACM (2008)
33. Zacchiroli, S.: (declared) VCS usage for Debian source pacakges (February 2011), http://upsilon.cc/~zack/stuff/vcs-usage
34. Zeller, A.: Yesterday, My Program Worked. Today, It Does Not. Why? In: Wang, J., Lemoine, M. (eds.) ESEC/FSE 1999. LNCS, vol. 1687, pp. 253–267. Springer, Heidelberg (1999)

Making Software Integration Really Continuous

Mário Luís Guimarães and António Rito Silva

Department of Computer Science and Engineering
IST, Technical University of Lisbon, Lisbon, Portugal
{mario.guimaraes,rito.silva}@ist.utl.pt

Abstract. The earlier merge conflicts are detected the easier it is to resolve them. A recommended practice is for developers to frequently integrate so that they detect conflicts earlier. However, manual integrations are cumbersome and disrupt programming flow, so developers commonly defer them; besides, manual integrations do not help to detect conflicts with uncommitted code of co-workers. Consequently, conflicts grow over time thus making resolution harder at late stages.

We present a solution that continuously integrates in the background uncommitted and committed changes to support automatic detection of conflicts emerging during programming. To do so, we designed a novel merge algorithm that is O(N) complex, and implemented it inside an IDE, thus promoting a metaphor of continuous merging, similar to continuous compilation. Evidence from controlled experiments shows that our solution helps developers to become aware of and resolve conflicts earlier than when they use a mainstream version control system.

Keywords: software merging, version control, continuous integration, conflict detection, continuous merging.

1 Introduction

Programming inside teams of multiple developers generally results in merge conflicts between concurrent changes. Conflicts can be difficult to detect in object-oriented programs without adequate tool support, and result in software defects as developers work more in parallel [14].

The problem with conflicts is that the later they are detected the costlier they are to resolve [1,8]. This is especially true as time passes without developers integrating their changes with those of co-workers. Not only conflicts can grow too much such that more code needs to be reworked later, but changes become less fresh in developers' minds making it more difficult to remember what was done and where to start resolution.

Recognizing this problem, good practice recommends developers to frequently integrate concurrent work to enable early detection of conflicts [1,8]. However, there are several limitations with manual integrations. First, they are disruptive because they require developers to pause their tasks, thus breaking the flow of programming. Second, they only detect conflicts with changes already committed

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 332–346, 2012.

in the Version Control System (VCS).[1] but they do not detect conflicts with uncommitted changes in the developers' working copies of the software system, so conflicts may grow as time passes thus making their resolution harder at late stages. Besides, mainstream VCSes only detect conflicts between overlapping textual regions in two versions of the same file (direct conflicts), but not between concurrent changes to different files (indirect conflicts). Third, when integration builds fail, developers still have to spend time understanding what happened and tracing failures back to the responsible changes and their authors.

This paper contributes a solution to report structural and semantic conflicts inside the IDEs of affected developers as conflicts emerge during programming. It is supported by a novel merge algorithm that continuously integrates in the background (in real-time) both uncommitted and committed changes in a team. The result is a metaphor of *continuous merging*, much like to continuous compilation inside the IDE, that alleviates developers from the burden of manual integrations for the sake of conflict detection, thus keeping them focused on programming. In contrast to our initial paper [10], this presents our background merging algorithm, and evaluates our solution using controlled experiments, showing evidence of its usefulness compared to only using a VCS.

The following sections are summarized: Section 2 describes the problem of not detecting conflicts early, and shows the limitations of current VCSes and manual integrations. Section 3 presents our solution to early conflict detection and its background merging process. Section 4 presents an empirical evaluation that sustains our solution. Section 5 lists the related work, and Section 6 concludes.

2 Problem

Imagine three developers checking out the same working copy of an application from a VCS, and then making concurrent changes. Mike ❶ changes class Mammal to extend Animal, and checks in. Anne ❷ creates class Primate by extending Mammal, adds a feature to move primates to an absolute position, merges Mike's changes from the head of the development line in the VCS, and checks in. Meanwhile, Bob ❸ changes class Animal to move animals by some distance from where they are, merges Mike's and Anne's changes, and checks in. Note that all changes were done to different files, so the merges were clean, that is, the VCS reported no conflicts. The final code in the development line is shown in Fig. 1. The problem with the final code is that an unexpected override conflict affecting the "move(int, int)" methods was not caught by the VCS, and now, it is causing this bug: if "Animal.move(int dx, int dy)" is called on a primate, this will move to position "(dx,dy)" instead of moving distance "(dx,dy)" from its current position, as expected for animals.

Using the VCS, the earliest the conflict could be found was when Bob merged Mike's and Anne's changes. Nevertheless, the VCS told him "Go ahead, the merge is clean!", so Bob has no reason to suspect Mike's and Anne's files. Even though he had written a test for "Animal.move(int dx, int dy)", this would not

[1] E.g., Subversion (`http://subversion.apache.org`) and Git (`http://git-scm.com`).

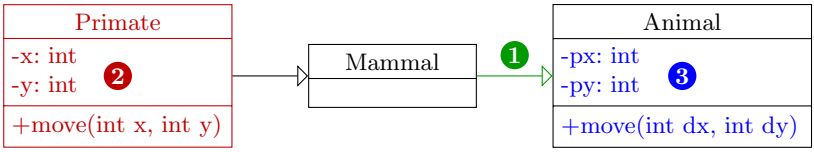


Fig. 1. The final merge at the head

check primates because Bob did not know about them when he wrote the test, so testing would not help much in this case.

The practice of frequent manual integrations has some limitations too. If Bob attempts to frequently integrate the code of his colleagues, he will probably bring code tangential to his work [1], and interrupt him too much. Because manual integration does not detect conflicts with uncommitted code in working copies, the best chance to detect the conflict is if Mike and Anne checked in often. However, they may prefer to defer check-ins until when they are finished with their tasks, thus delaying the detection of the conflict. On the other hand, this practice requires developers to check-in partial changes just for the sake of conflict detection, thus causing distraction and polluting the VCS with insignificant check-ins.

Eventually, a few days later a user approaches Bob: "Do you remember that animal move feature I asked?", "Yes!?", "It does not work for gorillas!", "How's that?", "Well, you coded it, go figure out!". Unfortunately, time passed and changes are no longer fresh in Bob's head, so he will have to work harder to investigate and resolve the bug. He will have to remember what he did before, determine the impact of the bug on other parts of the code, approach his colleagues if they are available, and decide what to do. At least he will have to remove one of the duplicated points, rename one of the "move" methods, and change where in the code there are dependencies on the removed point and the renamed method. All this requires more time and effort than if the conflict was detected earlier. This example is simple but shows that conflicts can be difficult to detect and are costly to resolve when found late.

Wouldn't it be helpful a tool that did continuous (real-time) integration in the background to automatically detect the above conflict as it emerged during programming, and reported it inside the IDE, thus exempting developers from manual integrations and from all that rework? This is what our solution does.

3 Solution

Our solution assumes that a software project comprises one or more teams of developers, each team working along a *development line* (or *branch*) [1] supported by a mainstream VCS, as shown in Fig. 2.

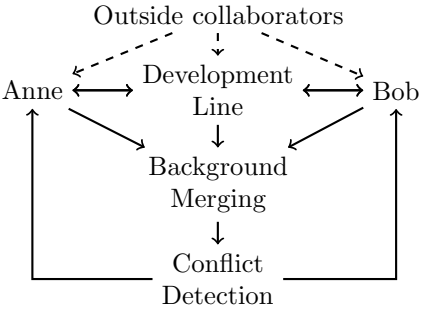


Fig. 2. The information flow inside a team

Team members follow the typical "copy-modify-merge" process: they check out working copies of the system from the development line, modify the working copies, merge other members' and outsiders' changes into the working copies, directly or via the line, and check in their working copies into the line.

Simultaneously, changes to the code are captured when files are saved in working copies or checked in into the development line, and transmitted to background merging in order to continuously update a background system, called the team's *merged system*. This system is then post-processed to automatically detect conflicts as they emerge inside the team. In addition, when members leave the team the effect of their changes is removed from the merged system.

Changes in working copies are sent to background merging automatically, or manually if the developer wants to take control — for example, a developer may decide to transmit only when changes are reasonably stable. Anyway, changes can only be sent if the working copy compiles successfully, to avoid syntactically invalid code entering the merged system. On the other hand, changes at the head of the development line are always automatically processed, so developers should guarantee that check-ins do not carry compilation errors into the development line, which is a good practice and an easy one to ensure using today's IDEs and VCSes (e.g., via pre-commit hooks).

Conflicts are reported in detail to affected members inside a view in the IDE, much like compilation errors are reported today, thus promoting a metaphor of *continuous merging*.

3.1 Tracking Changes

The working copies, the check-ins along the development line, and the merged system are abstractly modeled as trees of labeled, typed, and attributed nodes representing the physical folders, files, and program elements. The labels allow to match nodes in different trees for computing changes between consecutive trees, and the types and attributes define how source code is stored in trees. Labels, types, and attributes are specific to the language domain.

We implemented our solution for Java programming, and chose the labels as the names of folders, files, fields, and the signatures of methods. In some cases,

to disambiguate program elements in the same scope, like classes and interfaces, the labels are the concatenation of name and type. Only folder nodes do not require attributes.

Fig. 3 exemplifies how changes inside the team are tracked and merged in the background (this figure will guide us throughout Section 3). In Fig. 3a, Anne and Bob made several concurrent changes to base file F.java, which were merged in the background into the file in the merged system (background merging is described in Section 3.2). In Fig. 3b, the content of some nodes in the base file is shown as an example of how source code is mapped to nodes and attributes.

Fig. 3c shows the trees of the base file and those of Anne's and Bob's working copies of that file.[2] As developers change the code, the working copy tree evolves from the base tree as follows: unchanged nodes are shared by the two trees (e.g., Bob did not change node "e"); added nodes only exist in the working copy tree (e.g., Anne added node "a"); deleted nodes only exist in the base tree (e.g., Anne deleted node "e"); and changed nodes appear as new nodes in the working copy tree (e.g., Anne's and Bob's node "pi"). In addition, added, changed, and deleted nodes cause a change of their parent nodes (*change propagation*), as it happened with file node "F.java". The arrows represent the succession relationships between the nodes in consecutive trees (the successor points to the predecessor). Likewise, the evolution of nodes between the trees of consecutive check-ins in the development line follows the same rules.

The Evolution Tree. The evolution of the software as it changes is tracked in the evolution tree, shown in Fig. 3d for F.java. The entries in this tree are called **evolution graphs**, and they capture the succession relationships of the nodes in the same labeled position in the working copies and the check-ins along the development line, as shown by the arrows. In a graph, the dark circles represent added or changed nodes in a tree, and the white circles, called **null nodes**, represent deleted nodes in a tree. In the figure, the labels "b", "A", and "B", indicate from which trees in our example the nodes come from (e.g., graph "e" shows a node deleted by Anne that is shared between the base and Bob). A node is **older than** another if it precedes the other on the transitive closure of the succession relationships (e.g., Bob's node "e" is older than Anne's null node "e").

In a graph, the nodes that are not succeeded are called **forefront nodes** because they contain the most recent edits to the attributes. A graph is n-way if its number of forefront nodes is n (> 0). A graph is **consistent** if all forefront nodes, except null nodes, have the same type (in this case we say that the graph is "a consistent <type>" or is "of type <type>"); otherwise, it is **inconsistent** — this is the bizarre case of one developer adding a file while another developer adds a folder with the same name. For Java, inconsistent nodes only occur at folder and file level. A graph is a **null graph** if it has only null nodes at the forefront.

The evolution tree is used to update the merged system during background merging and to identify members affected by conflicts.

[2] We omit parent folders for clearness, but the discussion applies to folders too.

F.java (base)

```
class F {
 float e = 2.7f;
 public float pi;
 int q = 1;
 int m(){
  return 1;
 }
}
```

F.java (Anne)

```
final class F {
 float e = 2.7f;
 public float pi =3.14;
 int a = 1;
 int q = 2;
 int m(){
  return q;
 }
}
```

F.java (Bob)

```
public class F {
 float e = 2.7f;
 public float pi;
 int b = 2;
 int q = 3;
 int m(){
  return 1;
 }
}
```

F.java (merged system)

```
public final class F {
 float pi=3.14;
 int a = 1;
 int b = 2;
 int q = 0;
 int m(){
  return q;
 }
}
```

(a) Anne's and Bob's changes to F.java (indicated by the rectangles and the strikes), and the resulting file in the merged system.

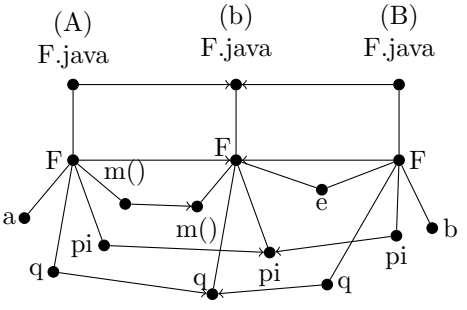
F.java:File
package=""

F:Class
visibility=""
final=false
extends=""

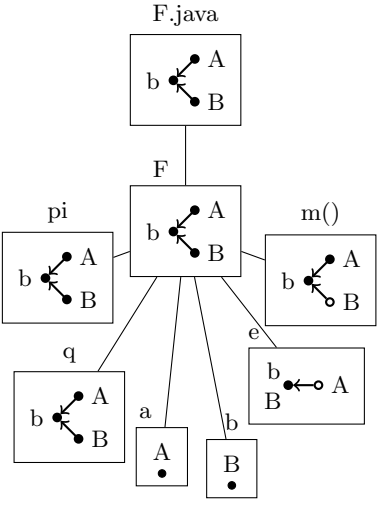
pi:Field
visibility="public"
type="float"
initval=""

m():Method
visibility=""
type="int"
body="{...}"

(b) Some nodes in F.java (base).



(c) The trees of (b)ase, (A)nne, and (B)ob.



(d) The evolution tree.

Fig. 3. Tracking changes inside the team

3.2 Background Merging

Background merging uses the evolution tree to do *automatic and incremental n-way structural merging* of the most recent changes to the software. It is:

- automatic because all structural conflicts are temporarily resolved in the merged system using *default resolutions* to not stall background merging. One example is the *deletions rule*, which ignores forefront null nodes when other forefront nodes exist, in order to favor changes over deletions;[3]
- incremental because it has to "remerge" only the folders and files that have been modified in the working copies or in the check-ins since the last call to background merging;
- and n-way because it merges the forefront nodes in each graph in one pass.

Initialization. The merged system is initialized with the tree at the development line's head, i.e., $MS = H$. Then, for each call i, MS is updated as follows:

Folder Merging. Visit bottom-up the subtree comprising the graphs in the evolution tree corresponding to the folders and files that were modified after call $i - 1$,[4] and follow these rules at each visited graph:

- If the graph is inconsistent, delete the corresponding node in MS.
- If the graph is a consistent file, proceed to "File Merging" (see below).
- If the graph is a consistent folder, create that folder in MS if it does not exist already there (e.g., it exists before call i, or it was created during call i because some child was created).
- If the graph is null and the corresponding node in MS is a file, or a folder having no children, delete that node in MS.

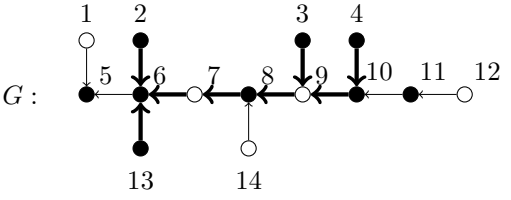
File Merging. Run these ordered steps at each visited graph G (top-down), starting with the evolution graph corresponding to the file:

1. Identify the graph G' to be merged. Let $G' = G$, then modify G' following the order of these steps:
   (a) Ignore all null nodes at the forefront whose parent nodes are null and were ignored in the preceding visited graph. This step is explained with an example. When a developer deletes a class and another developer changes one of its methods, MS should include the entire class with the method change to avoid programming language inconsistencies (e.g, the change may use fields and methods that were deleted with the class). Therefore, this step ignores all null nodes corresponding to the class and its descendents that were deleted, so the class is not incomplete in MS.

[3] We have tested this decision with twenty-one graduate students by exposing them to a "change & deletion" situation, and they all decided to preserve the change.

[4] Bottom-up traversal supports files checked out from different points in the development line, and checkouts of partial trees.

(b) For the remaining nodes, let N = {all null nodes at the forefront} and $\overline{N}$ = {all non-null nodes at the forefront}. Then, if $\overline{N}$ is not empty (you may follow the example in the figure below in which the final G' corresponds to the nodes linked by the strong arrows):
   i. Ignore all nodes in N (*deletions rule*);
   ii. Ignore all nodes succeeded by those in N that are not succeeded by those in $\overline{N}$ (nodes only succeeded by ignored null nodes do not contribute with attribute edits to the merge);
   iii. Ignore all nodes succeeded by the oldest node that directly precedes one in $\overline{N}$ (the oldest or its successors contribute to the values of all attributes of the forefront nodes in G', so we can cut the "tail").



Ignored at each step:

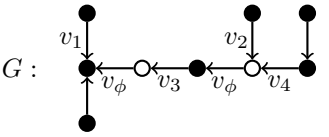
i. {1, 12, 14}
ii. {11}
iii. {5}

2. Now do the one that applies:
   – If G' is a null graph, delete the corresponding node in MS.
   – If G' has only one node at the forefront, copy that node's tree to MS.
   – If G' has several nodes at the forefront:
      (a) Copy the merge of G' to MS as per "Merging a Graph" (see below).
      (b) Repeat 1, now for each of the evolution graphs corresponding to the children of the nodes at the forefront of G'.

Applying these steps to merge Anne's and Bob's modifications to base file F.java will result in "F.java (merged system)" listed in Fig. 3a.

Merging a Graph. A consistent graph G of type T is merged as follows:

1. Set $n \leftarrow$ a new node of type T.
2. For each n_a attribute of n do:
   (a) Set $E \leftarrow$ {the values of concurrent edits of attribute a in G}. For example, given the values of the edits of a (the v_is, and the v_ϕs of null nodes) in the next figure, $E = \{v_1, v_2, v_4\}$:



   (b) If $\#E = 0$: Set $n_a \leftarrow$ the single value of a in all forefront nodes (because there are no concurrent attribute edits we maintain the former value).
   (c) If $\#E = 1$: Set $n_a \leftarrow$ the single value in E (either there is a single edit, or all concurrent edits set the same value).

(d) Otherwise ($\#E > 1$): Set $n_a \leftarrow$ the default value for a, to not stall background merging.

3. Return the merge node n.

Complexity Analysis. Performance of background merging is assessed via a cursory complexity analysis.

Time Complexity. File merging is $O(N_{elem} \times R_{max} \times A_{max})$ where N_{elem} is the number of elements visited in a file, R_{max} is the maximum number of nodes in any of those elements' graphs, and A_{max} is the maximum number of attributes in any of those elements' types. Since A_{max} is bounded by the programming language, file merging is $O(N_{elem} \times R_{max})$. Background merging updates in the worst case N_{files} and calls file merging for every file, so its time complexity is $O(N_{files} \times N_{elem} \times R_{max}) \approx O(N \times R_{max})$, where N is the total number of elements to "remerge". In practice, R_{max} is bounded by the maximum team size, which is generally small for teams in a project to be manageable, so the overall time complexity is $O(N)$. In our implementation, we merged 16 file versions (484 LOC each on average) in $4.3ms$.

Space Complexity. This is proportional to the memory needed to maintain the evolution tree, so in the worst case it is $O(N_{system} \times R_{max})$, where N_{system} is the total number of folders, files, and elements in the software system being created by the team, and R_{max} is the same as above. Like before, this can be approximated by $O(N_{system})$ in practice. Besides, the evolution tree is computed and stored in memory as needed.

3.3 Conflict Detection

Structural Conflicts. These are detected during background merging. They are temporarily resolved in the merged system using *default resolutions*, defined for the language domain. Default resolutions are necessary to not stall background merging, yet these conflicts persist in the affected working copies until the team resolves them after being informed. The types of structural conflicts are these:

- *pseudo direct conflict*: occurs when different attributes of a node are concurrently changed, or the same attribute is concurrently changed to the same value. It is a warning that reminds of a possible semantic conflict at language level (see below). In Fig. 3a, it happened with class "F" and field "pi";
- *attribute change & change conflict*: occurs when the same attribute of a node is concurrently changed to different values. The default resolution is to assign a default value to the attribute in the merged system's node according to the attribute's type. In Fig. 3a, this conflict was temporarily resolved (gray color) for the "initval" attribute of field "q" by setting it to zero (the default value of "int");

- *node change & deletion conflict*: occurs when there are concurrent changes and deletions to the same node, and the default resolution is to apply the deletions rule. In Fig. 3a, it happened with method "m()", so Anne's change prevails in the merged system;
- *inconsistent graph conflict*: occurs when an evolution graph becomes inconsistent, and the default resolution is to delete the corresponding node in the merged system. This weird case should never happen, yet we handle it.

Note that developers are always alerted to structural conflicts, and once they resolve them in their working copies the merged system is updated with their resolution in the next iteration of background merging. This is why default resolutions in the merged system are always temporary.

The remaining conflicts are detected by post-processing the merged system.

Language Conflicts. The merged system is immediately compiled after being updated. As explained before, all changes only enter background merging if they are syntactically valid, so compilation errors in the merged system can only result from invalid combinations of concurrent changes with respect to the static semantics of the programming language.

As such, post-processing listens the compilation output, processes the errors, and reports them as *language conflicts* back to the IDEs of affected members. This is a very effective solution because it avoids to re-implement complex programming language rules. One case is the *undefined constructor conflict*, which occurs when one developer adds a constructor with one argument to a class having no constructors, while another developer creates a subclass of that class.

Behavior Conflicts. These represent undesired behavior because of unexpected interactions between merged changes. They are detected by searching for *conflict patterns*, that is, logical conjunctions of facts regarding the program elements and their semantic dependencies in the merged system that identify potentially unwanted behavior.

The more specialized patterns are the more interesting ones because the conflicts they represent are hard for developers to find without tool support. This is the case of the *unexpected override conflict* in Section 2, which is found using the pattern $\exists A, B, m_1, m_2 \in G : extends*(A, B) \wedge method(A, m_1) \wedge method(B, m_2) \wedge equalSignature(m_1, m_2)$, where A is a super class of B and $extends*$ is the transitive closure of the *extends* dependency.

Note that conflicts only occur if the instantiated facts correspond to nodes changed by different members (we omitted this part in the example pattern to avoid complicating it). An advantage of using conflict patterns is that they can be easily added to support more behavior conflicts.

Test Conflicts. These are detected by running automated tests in the merged system. A test conflict is one that fails and its execution flow has reached methods changed by different members. Suppose that Anne adds a test to verify that all species have a price defined by method "getPrice()", and Bob adds class

Chimpanzee without such method because he is not aware of Anne's new feature. As such, Anne's test will fail in the merged system when retrieving the price of Chimpanzee, like this execution flow shows:

```
zoo.testing.ZooTests.setUp()✔
zoo.testing.ZooTests.testAnimalGetPrice()✔                    (Anne)
...
zoo.animals.Animal.getPrice(Ljava/lang/Class;)✔
zoo.animals.Chimpanzee.getPrice()✖                            (Bob)
```

A test conflict is detected for "testAnimalGetPrice()" because its execution reached Anne's new method (the test) and tried to call "getPrice()" on Bob's new class via reflection. Post-processing places hooks in the reflection API (via bytecode instrumentation), and checks if missing methods were deleted or never existed, which was the case for Bob, thus detecting a *missing method conflict*.

3.4 Reporting Conflicts

A conflict is reported to the members that changed the nodes affected by it. These are the nodes that were "remerged" (structural conflicts), the nodes involved in a compilation error (language conflicts), the nodes that instantiate the facts of a conflict pattern (behavior conflicts), and the nodes corresponding to the methods in failed execution flows (test conflicts). Only the members that modified these nodes will receive notifications for the conflicts affecting the nodes. The evolution tree tracks who modified which nodes, so to find these members we look for those nodes in this tree.

4 Evaluation

Our evaluation shows that developers using our solution become aware of and resolve conflicts earlier than when they use only their VCS. It was done via controlled user experiments, as described next.

The Tool. In order to evaluate, we implemented WECODE as an extension to the Eclipse IDE (`http://www.eclipse.org`). The main screen of WECODE is shown in Fig. 4. The Team view ❸ shows all members in the team and the details of their changes down to program elements. It shows yellow and red icons to respectively signal pseudo or more urgent structural conflicts on folders and files. If they wish to control change transmission, developers can manually transmit changes to background merging only at stable moments of their tasks. Developers can also update their code with other members' changes in order to early resolve conflicts while changes are vivid in their minds (a chat view, not shown, facilitates the discussion of changes and resolutions). The Team Merge view ❹ reports semantic conflicts: notifications have detailed messages

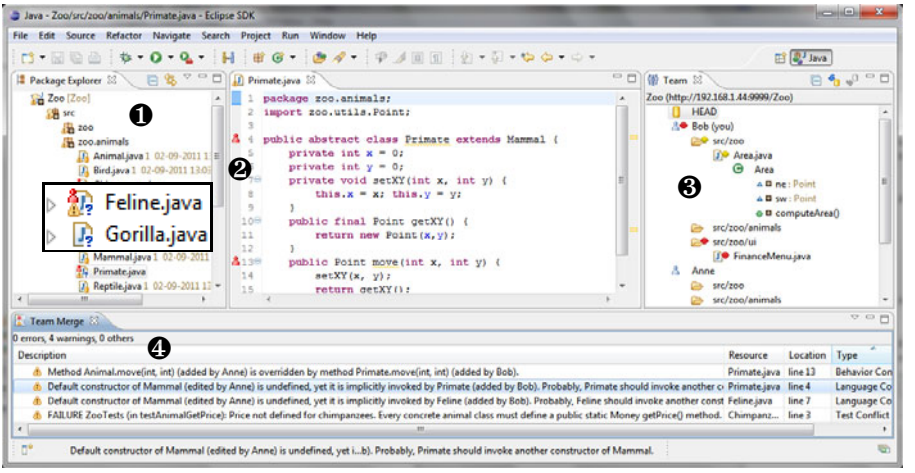


Fig. 4. Continuous merging inside the IDE

describing the conflict, the affected program elements, the affected members, and how elements were changed. This fosters conflict resolution. In addition, conflicts are signaled at affected files ❶ and program elements ❷.

Controlled Experiments. *Does our tool help developers become aware of and resolve conflicts earlier than when they use a mainstream VCS?* To answer this, we organized two groups of 7 graduate software engineering students: the WECODE and the VCS groups. Each subject teamed with a confederate, who inserted the same conflicts (those in the Team Merge view ❹) at equivalent times for all subjects, before half of their tasks were done. The application (41 classes, 1143 LOC) and the tasks were designed by the authors, and the application was sent to all subjects at least two days before their experiment. Before start, the subjects watched a tutorial video of the tools to use (WECODE and Subversion). At the end, they were asked to check in and resolve any remaining conflict.

Results. WECODE subjects became aware of all 28 conflicts (7 subjects × 4 conflicts) as they emerged during their tasks before check in, whereas VCS subjects detected no conflict before check in (these are statistically significant results by Fisher's or Pearson χ^2's tests of the corresponding 2x2 contingency table). At check in, VCS subjects only detected the language conflicts because of the compilation errors (they forgot to run the tests so they missed the test conflict). Since there were no direct conflicts VCS subjects did not pay attention to the files changed by the confederate. Only one VCS subject was observed to frequently integrate with the VCS after each task, but there was no direct conflicts so he said "there is no problem here".

Regarding the WECODE group, subjects resolved conflicts early and generally between their tasks. The average delay to start resolving each conflict type was (we removed other conflicts' resolution time from delays): language conf. (4m22s, sd=5m1s); behavior conf. (2m10s, sd=2m44), and test conf. (3m1s, sd=3m55). This was also the order that conflicts emerged, and even though the sample was small, it is interesting to observe that the delay slightly decreased as subjects got more used with the tool.

Both groups were asked to score (from 1 to 10) if they (liked / would like) to be informed about conflicts during programming, instead of only at check-in. WECODE subjects scored 9.3 (sd=0.70) and VCS subjects scored 8.00 (sd=1.85), thus showing a high desire for such feature. Asked why, they said:

> VCS: "I would not have to read many conflicts at the end when I probably had forgotten the changes I made" and "It would make development more interactive, and I would anticipate when a colleague is breaking our code"
> MERGE: "Because it informs me in useful time, thus sparing me time looking for errors later. All time wasted tracing error messages in the build would be spent doing useful things"

About our tool, the subjects said "I liked the icon in the editor informing me about the conflict and with whom" and "I liked most its simplicity of use".

Threats to Validity. Subjects did not knew they would be evaluating our tool to not influence their behavior and responses. They were randomly selected into the groups, and all had experience with the VCS. Using Subversion or another mainstream VCS would not change the results: all they do is textual merging. The code to type in every task was given to them thus eliminating the effect of different programming skills. Regarding external validity, our conflicts might be threatened regarding their occurrence in practice. Studies are needed to understand the nature of conflicts, however it is reasonable to assume that conflicts in real projects are at least as difficult to detect as those we chose. Our results indicate that continuous merging can be beneficial, still we believe that experimenting with real projects will provide further insight into our work.

5 Related Work

Our work relates to others in the areas of software merging and awareness.

Software Merging. Textual merging (Unix's diff3, and all mainstream VCSes) blocks when textual conflicts occur, like when different attributes are changed on the same line of text, and fails to match changes when the program elements are reordered inside concurrent files. Consequently, background textual merging, like done in [4], has these limitations. In contrast, our background structural merging never blocks, handles changes to different nodes and attributes transparently, and supports reorderings, so it detects more important conflicts (semantic) earlier. *Flexible structural merging* [13] merges two versions of a file

using two-dimensional matrices that decide how merging is done at node level (manual or automatic). Flexibility is achieved by configuring matrices for different collaboration scenarios. This solution does not scale for more than two versions of a file, because matrices need to be reconfigured each time the number of versions varies, which is unfeasible in practice. *Semantics-based merging* [3] has been mostly theoretical achievements using very limited languages. *Operation-based merging* [12] serializes two concurrent sequences of operations. Tools must capture all changes as operations, but the editors developers use are not operation-based. Sequences may grow too much, and redundant operations must be eliminated to avoid false conflicts. In contrast, our solution adapts to the tools developers use.

Awareness. Solutions based on *awareness* [7] report which files, types, and program elements are being changed at the moment by co-workers, which may help to detect conflicts early. These solutions may overload developers with notifications that are irrelevant to what they are doing [5,9,11], and require developers to investigate the notifications to determine if they bear any conflict. This can be difficult because of the complex semantic dependencies between program elements (e.g., polymorphism and late binding). Besides, this steals time from programming. For example, some solutions report direct conflicts even when changes are done to independent program elements in a file [2,15]. A structure-based solution like ours does not have this shortcoming. Others go beyond direct conflicts, and notify when two files, types, or program elements, connected by a path of semantic dependencies, have been concurrently changed by the developer and a co-worker [6,16,17]. Nevertheless, in these solutions, developers still have to investigate the notifications to identify where real conflicts exist.

6 Conclusion and Future Work

Early detection of conflicts is important to facilitate resolution. The recommended practice is to frequently manually integrate others' changes, but this is too much burden and disrupts the flow of programming. In contrast, we presented a solution that does really continuous integration in the background in order to automatically detect conflicts as they emerge during programming, and reports them in detail inside the IDE. An empirical evaluation demonstrated that our solution makes developers aware of conflicts that are difficult for them to find using current tools, and fosters early resolution while changes are still fresh. This support clearly contrasts with the tools developers use today.

Our research will proceed in several directions. We want to support refactorings and domains beyond programming, like collaborative model-driven engineering. We want to evaluate our solution via a longitudinal study with professional programmers in order to adjust it to real projects, and to understand how continuous merging influences their software process, for example, if new collaboration patterns emerge. In the long term, we want to measure the overall effect of continuous merging on software quality.

References

1. Berczuk, S., Appleton, B.: Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Addison-Wesley, Boston (2002)
2. Biehl, J., et al.: FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In: ACM SIGCHI Conf. on Human Factors in Computing Systems, CHI 2007, pp. 1313–1322. ACM Press, New York (2007)
3. Binkley, D., et al.: Program Integration for Languages with Procedure Calls. ACM Trans. Softw. Eng. Methodol. 4(1), 3–35 (1995)
4. Brun, Y., et al.: Proactive Detection of Collaboration Conflicts. In: 8th Joint Meet. of the Euro. Softw. Eng. Conf. and ACM SIGSOFT Symp. on the Foundations of Softw. Eng., ESEC/FSE 2011, pp. 168–178. ACM, New York (2011)
5. Damian, D., et al.: Awareness in the Wild: Why Communication Breakdowns Occur. In: Inter. Conf. on Global Softw. Eng., ICGSE 2007, pp. 81–90. IEEE Computer Society, Washington, DC (2007)
6. Dewan, P., Hegde, R.: Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In: Bannon, L., Wagner, I., Gutwin, C., Harper, R., Schmidt, K. (eds.) ECSCW 2007, pp. 159–178. Springer, London (2007)
7. Dourish, P., Bellotti, V.: Awareness and Coordination in Shared Workspaces. In: ACM Conf. on Computer Supported Cooperative Work, CSCW 1992, pp. 107–114. ACM, New York (1992)
8. Fowler, M.: http://martinfowler.com/articles/continuousIntegration.html
9. Fussell, S., et al.: Coordination, Overload and Team Performance: Effects of Team Communication Strategies. In: ACM Conf. on Computer Supported Cooperative Work, CSCW 1998, pp. 275–284. ACM, New York (1998)
10. Guimarães, M., Rito-Silva, A.: Towards Real-Time Integration. In: 3rd Inter. Workshop on Cooperative and Human Aspects of Softw. Eng., CHASE 2010, pp. 56–63. ACM, New York (2010)
11. Kim, M.: An Exploratory Study of Awareness Interests about Software Modifications. In: 4th Inter. Workshop on Cooperative and Human Aspects of Softw. Eng., CHASE 2011, pp. 80–83. ACM, New York (2011)
12. Lippe, E., van Oosterom, N.: Operation-based Merging. In: 5th ACM SIGSOFT Symp. on Softw. Dev. Environ., SDE 5, pp. 78–87. ACM, New York (1992)
13. Munson, J., Dewan, P.: A Flexible Object Merging Framework. In: ACM Conf. on Computer Supported Cooperative Work, CSCW 1994, pp. 231–242. ACM, New York (1994)
14. Perry, D., et al.: Parallel Changes in Large-Scale Software Development: An Observational Case Study. ACM Trans. Softw. Eng. Methodol. 10(3), 308–337 (2001)
15. Sarma, A., et al.: Palantír: Raising Awareness among Configuration Management Workspaces. In: 25th Inter. Conf. on Softw. Eng., ICSE 2003, pp. 444–454. IEEE Computer Society, Washington, DC (2003)
16. Sarma, A., et al.: Towards Supporting Awareness of Indirect Conflicts Across Software Configuration Management Workspaces. In: 22nd IEEE/ACM Inter. Conf. on Aut. Softw. Eng., ASE 2007, pp. 94–103. ACM, New York (2007)
17. Schümmer, T., Haake, J.: Supporting Distributed Software Development by Modes of Collaboration. In: 7th Euro. Conf. on Computer Supported Cooperative Work., ECSCW 2001, pp. 79–98. Kluwer Academic Publishers, Norwell (2001)

Extracting Widget Descriptions from GUIs

Giovanni Becce, Leonardo Mariani, Oliviero Riganelli, and Mauro Santoro

Department of Informatics, Systems and Communications
University of Milano Bicocca
Milano, Italy
{becce,mariani,riganelli,santoro}@disco.unimib.it

Abstract. Graphical User Interfaces (GUIs) are typically designed to simplify data entering, data processing and visualization of results. However, GUIs can also be exploited for other purposes. For instance, automatic tools can analyze GUIs to retrieve information about the data that can be processed by an application. This information can serve many purposes such as ease application integration, augment test case generation, and support reverse engineering techniques.

In the last years, the scientific community provided an increasing attention to the automatic extraction of information from interfaces. For instance, in the domain of Web applications, learning techniques have been used to extract information from Web forms. The knowledge about the data that can be processed by an application is not only relevant for the Web, but it is also extremely useful to support the same techniques when applied to desktop applications.

In this paper we present a technique for the automatic extraction of descriptive information about the data that can be handled by widgets in GUI-based desktop applications. The technique is grounded on mature standards and best practices about the design of GUIs, and exploits the presence of textual descriptions in the GUIs to automatically obtain descriptive data for data widgets. The early empirical results with three desktop applications show that the presented algorithm can extract data with high precision and recall, and can be used to improve generation of GUI test cases.

Keywords: program analysis, graphical user interface, testing GUI applications.

1 Introduction

A Graphical User Interface (GUI) can be a valuable source of information for understanding the features implemented by an application. For instance, a GUI typically includes a number of descriptive labels that specify the kind of data that an application processes; a GUI includes menus and buttons that are related to the features an application can execute; a GUI visualizes and processes data, such as descriptions of facts, names of places, and names of people. Unfortunately, the knowledge represented by the content of descriptive labels, menu and data is embedded into the widgets and it is hardly accessible by automatic systems.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 347–361, 2012.

In the last years, the scientific community provided an increasing attention to the extraction of information from interfaces with the objective of understanding the data and the features offered by an application under analysis. In particular, techniques for extracting data from Web interfaces, such as [13,19,10], early studied this problem limitably to Web forms with the objective of easing data integration of online databases.

Extracting information from interfaces is not only relevant when applied to Web applications, but also when applied to desktop applications. For example, a relevant limitation of automatic test case generation techniques for GUI-based desktop applications is the lack of mechanisms for the identification and generation of data values useful to produce interesting executions in the application under test [20]. Identifying the right data that can be entered in a GUI would overcome this limitation, increasing the effectiveness of test case generation. Similarly, the automatic extraction of the data and features available in a desktop application would enable the possibility to automatically check conformance with respect to requirements [17]. Many other areas could also benefit from the extraction of information from GUIs, for example reverse engineering, and tool integration.

In this paper, we present a technique for the automatic identification of descriptive information about the data that can be entered in data widgets of GUI-based desktop applications. The technique relies on well grounded and widely adopted standards and best practices about the design of GUIs, and exploits the presence of widgets that include textual descriptions to discover the right descriptors of data widgets. The early empirical results with three applications show that the presented algorithm is effective and has the potential of enabling the previously described researches. We also report early empirical results that show how generation of GUI test cases can benefit from our algorithm.

The paper is organized as follows. Section 2 describes the principles underlying the design of our technique. Section 3 presents the algorithm that we use to extract descriptors of data widgets. Section 4 presents early empirical results. Section 5 discusses related work. Finally, Section 6 provides concluding remarks.

2 Design Principles of GUIs

How to design easy-to-use and user-friendly interfaces has been a subject of studies from many years. Nowadays there are a number of mature standards, guidelines and practices that help developers designing good GUIs. Among the many standards, we recall the Java look and feel design guidelines [3], the ISO guidance and specifications [6], and the the laws of the Gestalt about the perception of the space [14]. Our technique exploits these standards and principles to correctly identify relations between widgets.

According to the taxonomy presented in [11], widgets can be classified in three groups: action widgets, static widgets, and data widgets. *Action widgets* are widgets that give access to program functions. A typical example of an action widget is a button. *Static widgets* are widgets used to increase the understandability and usability of a GUI, but not for direct interaction. A typical example

of a static widget is a label. *Data widgets* are widgets that display or accept data. A typical example of a data widget is a textarea.

In this work, we focus on static and data widgets. We further distinguish static widgets in descriptive widgets and container widgets. *Descriptive widgets* are static widgets that display textual information that help users understanding how to use action and data widgets (e.g., labels). *Container widgets* are static widgets used to group related widgets (e.g., panels and frames).

The key idea exploited in this paper is retrieving descriptions of data widgets by looking in the descriptive widgets. Thus, we designed an algorithm that can identify the descriptive widget associated with a data widget according to three basic principles about the design of GUIs: proximity, homogeneity, and closure (from the laws of the Gestalt). In the following, we describe how these principle affect the algorithm.

Proximity. The law of proximity is based on the fact that people tend to logically group together objects that are displayed close each other. This principle is almost applied to the design of every GUI. For instance, the expected content of a data widget (e.g., a person name) is normally specified with a label (e.g., with the text "name") that is placed close to the data widget. In this work we focus on the left-to-right writing convention, we thus consider that labels, and more in general descriptive widgets, are placed at the left/top of the documented widget. The algorithm presented here can be easily adapted to other writing conventions.

Fig. 1. Search space for data widgets

Our algorithm takes advantage of this convention by restricting the search space that it considers when looking for a descriptive widget associated with a data widget. In particular, when searching a descriptive widget that appropriately describes the content of a data widget the algorithm looks into the rectangular area delimited as follows: the bottom-right corner of the area coincides with the center of the data widget under consideration, and the top-left corner of the area coincides with the top-left corner of the current window. Any descriptive widget which is entirely or partially displayed in that area is considered as a candidate for being associated with the data widget under analysis.

More formally, if (x, y) are the cartesian coordinates of the center of a data widget of interest dw, and (x^*, y^*) are the cartesian coordinates of the upper left corner of a descriptive widget, only the descriptive widgets that satisfy the following relation are candidates for being associated with dw: $x \geq x^* \wedge y \geq y^*$.

Figure 1 shows an example. Any descriptive widget that intersects the dotted area is a candidate descriptor for the textarea.

Homogeneity. The principle of homogeneity says that widgets should be distributed with regular patterns and possibly grouped according to their semantic. The regular distribution of descriptive and data widgets often implies that widgets are aligned horizontally and vertically (see for example the GUI in Figure 2). When widgets are dense the likelihood of associating a wrong descriptive widget to a data widget is quite high.

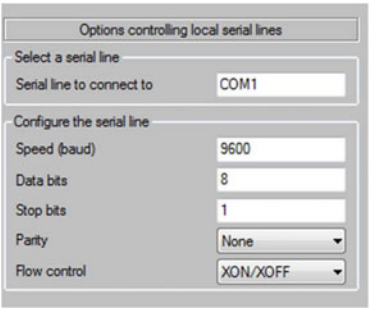


Fig. 2. Aligned widgets

We took this issue into account when defining how to compute the distance between widgets, as illustrated in Figure 3. Each widget is associated with a representative point. The distance between two widgets is defined as the distance between their representative points. We choose representative points with the purpose to favor descriptive widgets that are aligned horizontally with the data widget under consideration, compared to widgets at different vertical positions. In particular, the representative point of the data widget is always its top-left corner. The representative point of any widget placed below or at the same level of the data widget is its top-right corner (the position of a widget is the position of its center). The representative point of any widget above the data widget is its bottom-left corner. Thus, if there are descriptive widgets placed both above the data widget and aligned with the data widget, the aligned widget is favored because its representative point is closer to the data widget (nevertheless it is still possible to associate a widget placed above the data widget with the data widget, if the aligned widget is far enough). This strategy is able to well handle the many situations where many widgets in a same window are aligned, like the case shown in Figure 2.

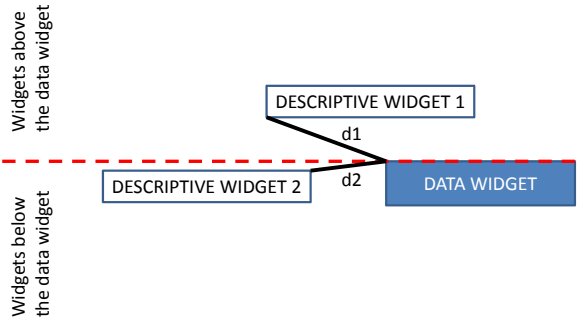


Fig. 3. Distance between widgets

Closure. The principle of closure says that persons tend to see complete figures even when part of the information is missing. This principle is typically exploited in the design of windows that contain many widgets. In fact, it is common practice to use container widgets for separating into multiple groups the widgets in a same window. Groups include semantically correlated widgets. For instance, two containers can separate the widgets designed for entering personal information from the widgets designed for entering credit card data, in a window dedicated to the handling of a payment process.

Our algorithm takes into account this practice implementing the possibility to limit the search space of a data widget to the widgets included in its same container.

3 Extraction of Widget Descriptions

In this section we present the algorithm for guessing associations between descriptive widgets and data widgets. The behavior of the algorithm is influenced by multiple parameters, which are empirically investigated in Section 4.

The types of widgets supported by the algorithm are specified in Table 1 column `Widgets`. Note that the column includes not only data widgets, but also container widgets. Container widgets are included because the label used to describe a container can be often used to describe the data widgets in the container as well.

The widgets listed in column `Widgets` cover the majority of data widgets that are used in practice. Each type of widget, specified in column `Widget Type`, can be associated with a different set of descriptors. Column `Descriptor Widgets` indicate the descriptors that the algorithm considers for each type of widget. For instance, our algorithm uses Labels, CheckBoxes and RadioButtons as possible descriptors for TextField; while it uses only Labels as descriptors for ComboBoxes. These associations are defined according to common practices in design of GUIs[1].

[1] note that for widgets like checkboxes the items that can be checked are reported at the right of small rectangles, but the items that should be checked are anyway described with a label placed in the area at the top-left of the checkboxes.

Table 1. Association between widgets and their descriptors

Widget Type	Widgets	Descriptor Widgets
Text	TextField, FormattedTextField, PasswordField, TextArea, EditorPane, TextPane	Label, CheckBox, RadioButton
Multichoice	CheckBox, RadioButton, ToggleButton, ComboBox, List	Label
Container	Panel, ScrollPane, TabbedPane, SplitPane	Label

Associations can be discovered statically (i.e., by analyzing the source code) or dynamically (i.e., by analyzing the windows of the application at run-time). Since static analysis techniques can only be applied if specific development strategies are adopted, such as the use of Rapid Application Development environments [15], our algorithm discovers associations dynamically. In particular, it extracts the data necessary for the analysis from the widgets displayed by the application at run-time.

Algorithm 1 reports the pseudocode of the main algorithm, while Algorithm 2 reports the pseudocode of the `isCandidate()` auxiliary function. The pseudocode is a simplified version of the implemented algorithm that does not consider performance optimizations.

Algorithm 1 takes as input a data widget, the set of widgets in the same window and four parameters, and returns the descriptor widget that passes the selection criteria implemented by the algorithm and is closest to the input data widget. Algorithm 2 implements all the checks that a widget has to pass to be considered as candidate descriptor for another widget. These checks include the ones inherited from the Proximity principle (see lines 6-8 in Algorithm 2); the Homogeneity principle (see computation of the distance at line 16 in Algorithm 1); the Closure principle, which is mapped into the local search strategy described afterward; and associations in Table 1, which are exploited for the check at line 2 in Algorithm 2. Table 2 specifies the values that can be assigned to parameters taken as input by Algorithm 1. We now describe in detail the role of the parameters.

Since a GUI can include noisy descriptor widgets, that is descriptor widgets with no information that are incorrectly used to layout the widgets in a window (e.g., empty labels invisible to users), the algorithm includes a noise reduction step. Noise reduction can be done at two different times: before starting the analysis of a window (opt_{noise} =Begin), or while analyzing a window (opt_{noise} =Incremental). When opt_{noise} =Begin, the algorithm removes every useless descriptor widget from the set considered by the analysis and then proceeds normally with the updated set (see lines 6-8 in Algorithm 1). When opt_{noise} =Incremental, the algorithm removes useless descriptor widgets incrementally while considering them (see lines 10-12 in Algorithm 2).

The algorithm can look for descriptor widgets globally in the current window (opt_{search} =Global) or within the current container only, following the closure principle (opt_{search} =Local). If opt_{search} =Local, Algorithm 2 discards every widget that is not in the same container than in the data widget under consideration (see lines 14-16 in Algorithm 2). If opt_{search} =Local, when no descriptor is

Algorithm 1. guessDescription()

Require: $dw = (x, y, width, height)$ a data-widget with upper left corner at (x, y), width $width$ and height $height$
Require: $W = \{w_1, \ldots, w_n\}$ a window with n widgets, where $dw \in W$
Require: opt_{noise}, opt_{search}, $opt_{visible}$, $opt_{hierarchical}$
Ensure: returns either $w \in W$ descriptive widget associated with dw or $\varnothing$

```
1:
2: pos_dw = ((x+width)/2, (y+height)/2)
3: min = MAXINT
4: bestWidget = ∅
5:
6: if opt_noise = Begin then
7:    W = removeNoisyWidgets(W)
8: end if
9:
10: for each i=1 to |W| do
11:    if not isCandidate(dw, W, w_i, opt_noise, opt_search, opt_visible) then
12:       continue //skip to next widget
13:    end if
14:
15:    //select the closest descriptive widget
16:    dist = computeDistance(w_i, dw)
17:    if dist < min then
18:       min = dist
19:       bestWidget = w_i
20:    end if
21: end for
22:
23: if min=MAXINT then
24:    if opt_search = Local and opt_hierarchical then
25:       return guessDescription(container(dw), W, opt_noise, opt_search, opt_visible,
          opt_hierarchical)
26:    else
27:       return ∅
28:    end if
29: else
30:    return bestWidget
31: end if
```

found for a data widget in a container, the algorithm can associate the descriptor of the container to the data widget ($opt_{hierarchical}$ =Yes), instead of using no descriptor ($opt_{hierarchical}$ =No). This case is covered by the recursive call at line 25 in Algorithm 1. If opt_{search} =Global, the algorithm can be further tuned to ignore widgets that are not visible to users ($opt_{visible}$ =VisibleOnly) or to also consider widgets invisible to users ($opt_{visible}$ =All). The check is implemented from line 18 to line 20 in Algorithm 2. A typical case influenced by this parameter is the analysis of a window with a ScrollPane that includes many elements, but only some of them are visualized at time.

Algorithm 2. isCandidate()

Require: $dw = (x, y, width, height)$ a data-widget with upper left corner at (x, y), width $width$ and height $height$
Require: $W = \{w_1, \ldots, w_n\}$ a window with n widgets, where $dw \in W$
Require: $w_i = (x_i, y_i, width_i, height_i) \in W$
Require: opt_{noise}, opt_{search}, $opt_{visible}$
Ensure: returns True if w is a candidate descriptor for dw, False otherwise

```
1:
2: if not compatible(type(w), type(dw)) then
3:     return False //skip widgets that cannot be associated with dw according to Table 1
4: end if
5:
6: if x < x_i or y < y_i then
7:     return False //skip widgets that are outside the interesting area of dw
8: end if
9:
10: if opt_noise = Incremental and noisy(w_i) then
11:     return False //incrementally ignore noisy widgets
12: end if
13:
14: if opt_search = Local and container(dw) ≠ container(w_i) then
15:     return False //ignore descriptive widgets in other containers
16: end if
17:
18: if opt_search = Global and opt_visible = VisibleOnly and not visible(w_i) then
19:     return False //skip widgets that are not visible
20: end if
21: return True
```

4 Empirical Evaluation

The empirical evaluation presented in this section investigates the quality of the results produced by the algorithm presented in this paper, with particular emphasis on the tradeoffs between the different configurations. We also analyze the performance of the algorithm, and we report early results about the benefits introduced in the AutoBlackTest GUI testing technique by the presented algorithm.

Case Studies. In order to evaluate the technique presented in this paper we looked for applications from different domains with GUIs of increasing size and structure. Table 3 summarizes the applications we selected from Sourceforge.

JPass [4] is a personal password manager. PDFSaM [5] is an application for splitting and merging PDF files. jAOLT [2] is a desktop client for eBay. Column `Windows Number` indicates the number of analyzed windows. We measured the size of the analyzed windows reporting the average and maximum number of widgets per window (columns `Widgets avg` and `max` respectively). We counted

Table 2. Parameters

Parameters	**Values**	
opt_{noise}	Begin	Incremental
opt_{search}	Global	Local
Global Search Parameters	**Values**	
$opt_{visible}$	All	VisibleOnly
Local Search Parameters	**Values**	
$opt_{hierarchical}$	Yes	No

Table 3. Case Studies

Application	**Windows**	**Widgets**		**Containers**	
	Number	**avg**	**max**	**avg**	**max**
JPass	4	20.75	32	1.5	4
PDFSaM	7	23.63	32	3.13	5
jAOLT	12	52.58	169	4.42	12

both visible and invisible widgets. To approximatively derive a measure of the structure of the GUI of an application, we measured the average and maximum number of container classes per window (columns `Containers` `avg` and `max` respectively), assuming that more containers per window intuitively suggests that developers provided greater effort into suitably grouping widgets according to their semantics.

Table 4. Configurations

Configuration Name	opt_{search}	$opt_{visible}$	$opt_{hierarchical}$	opt_{noise}
GLOBAL(All) + Incremental	Global	All	-	Incremental
GLOBAL(All) + Begin	Global	All	-	Begin
GLOBAL(VisibleOnly) + Incremental	Global	VisibleOnly	-	Incremental
LOCAL(No) + Incremental	Local	-	No	Incremental
LOCAL(Yes) + Incremental	Local	-	Yes	Incremental

Empirical Process. In the validation, we studied the configurations reported in Table 4. Note that the option $opt_{noise} = Begin$ is studied only for the Global search strategy. We made this choice because it was clear already from the initial experiments that in the practice the Begin and Incremental strategies produce results with the same quality, but Incremental is faster. We thus kept $opt_{noise} = Incremental$ for the rest of the experiments.

We measure the quality of the results produced by the algorithm using the standard metrics of precision, recall and F-measure. Precision indicates the fraction of correct associations extracted by the algorithm with respect to the overall number of extracted associations. Recall indicates the fraction of correct associations extracted by the algorithm with respect to the overall number of associations that could be extracted from the applications. F-measure is an index that combines and balances precision and recall. Formally,

$$precision = \frac{CA}{WA + CA}, recall = \frac{CA}{TA}, F\text{-}measure = \frac{2 * precision * recall}{precision + recall} \quad (1)$$

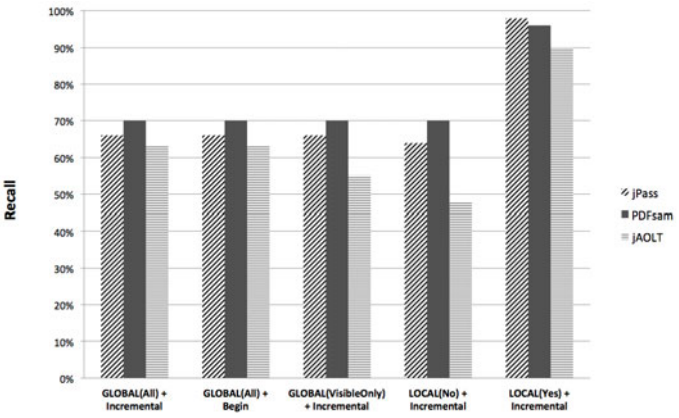


Fig. 4. Recall

where *CA* is the number of correct associations extracted by the algorithm, *WA* is the number of wrong associations extracted by the algorithm and *TA* is the total number of correct associations that the algorithm should have retrieved. The value of *CA*, *WA* and *TA* are measured by checking one by one each association retrieved by the algorithm and every widget in every window of the case studies. To mitigate the risk of computing imprecise data we repeated the counting multiple times.

We evaluated the performance of the technique by measuring the total time required for analyzing the GUI of the applications.

We finally integrated the algorithm in the AutoBlackTest test case generation technique [12]. AutoBlackTest generates GUI test cases randomly choosing concrete input values from a pre-defined set of values. The integration of the algorithm presented in this paper augmented AutoBlackTest with the capability of selecting test inputs according to the kind of data widgets that must be filled in. We measure the benefit of the augmented approach measuring code coverage.

Effectiveness. Figure 4 shows the results about recall. We can notice that every configuration based on a Global search performed similarly. The restriction of the search to visible widgets only causes a small reduction of recall, which means that the option causes the lost of some relevant associations. The local search with $opt_{hierarchical}$ =No surprisingly behaves worst than any global search. On the contrary, when $opt_{hierarchical}$ =Yes the recall raises to values between 90% and 100%. These results suggest that in the practice the labels associated with containers are frequently used to also describe the data expected by data widgets.

Figure 5 shows data about precision. We can notice that every configuration worked well, which means that regardless the selected configuration the algorithm seldom extract wrong associations. We can also notice that the local search worked slightly better than the global search. Intuitively this confirms

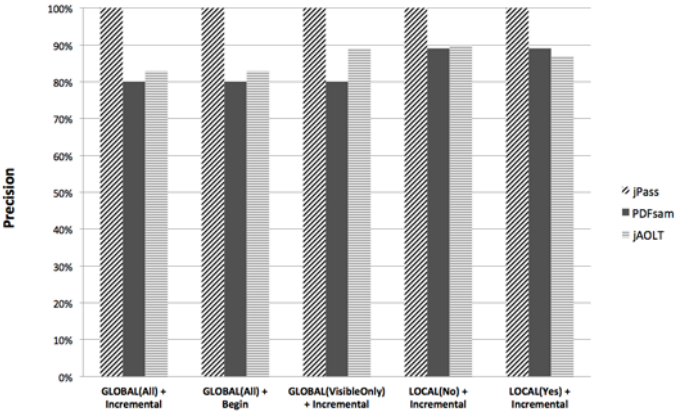


Fig. 5. Precision

that the closure principle is applied in the practice, it is thus better to restrict the search within containers, otherwise the risk of extracting wrong associations increases.

Figure 6 shows data about F-measure. We can notice that global search and local search with $opt_{hierarchical}$ =No perform similarly. On the contrary the last configuration outperformed the others producing the best compromise in terms of precision and recall (note that F-measure varies between 89% and 99%). It is interesting to notice that local search with $opt_{hierarchical}$ =No is worse than global search, while the local search with $opt_{hierarchical}$ =Yes is better than global search, regardless the amount of containers and structure in the interface. Considering the presence of containers is thus important only if considering also the labels associated with containers, even for interfaces with little structure.

Performance. To evaluate the performance of the configurations we measured the amount of time required to complete the analysis of the windows implemented in the three case studies that we selected. Figure 7 visualizes the performance of each configuration.

We can notice that the configuration that generated the best performance is also the slowest one, while the other configurations have similar performance. Since the time difference is small both in absolute and relative terms, unless performance is of crucial importance, the LOCAL(Yes) + Incremental is the configuration that should be selected. On the contrary, if performance is a crucial aspect, the fastest configuration that still provides good results is GLOBAL(All) + Incremental.

Augmenting AutoBlackTest. The technique presented in this paper can be used to augment the capabilities of many techniques in many domains. Here we

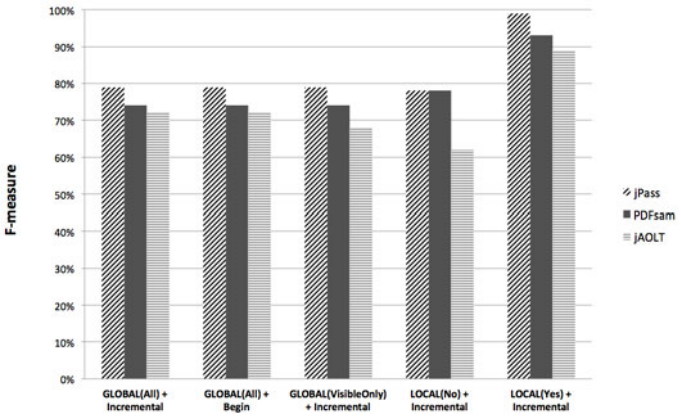


Fig. 6. F-measure

report early empirical data about the improvement that this technique produced in AutoBlackTest [12]. The study focuses on two applications that we already tested with AutoBlackTest and that we now tested gain with the augmented version of AutoBlackTest: PDFSam [5] and Buddi [1].

AutoBlackTest automatically generates GUI test cases that contain simple concrete values selected from a predefined data pool with many generic strings and numbers. This clearly limits the testing capability of AutoBlackTest. We extended AutoBlackTest with both the capability of associating descriptions to data widgets, according to the algorithm presented in this paper, and with the capability of using a datapool with concrete values organized according their kind. For instance, the datapool distinguishes dates, quantities, person names, and city names. The datapool is manually populated with concrete values following the boundary testing principle, that is it includes legal values, boundary values, illegal values and special values. As a result, the augmented AutoBlackTest can generate test cases that make a better use of data widgets by selecting proper concrete values from the datapool, exploiting the semantic information associated with the widgets.

The non-extended version of AutoBlackTest generated GUI test cases that cover 64% and 59% of the code in PDFSam and Buddi, respectively. The extended version of AutoBlackTest increased code coverage to 70%(+6%) and 64%(+5%), respectively. Considering that the computations implemented by these applications make a limited use of the data entered in data widgets (they mostly store and retrieve values from a database), the obtained increment is encouraging. In the future, we will study stronger ways of integrating this algorithm with test case generation techniques.

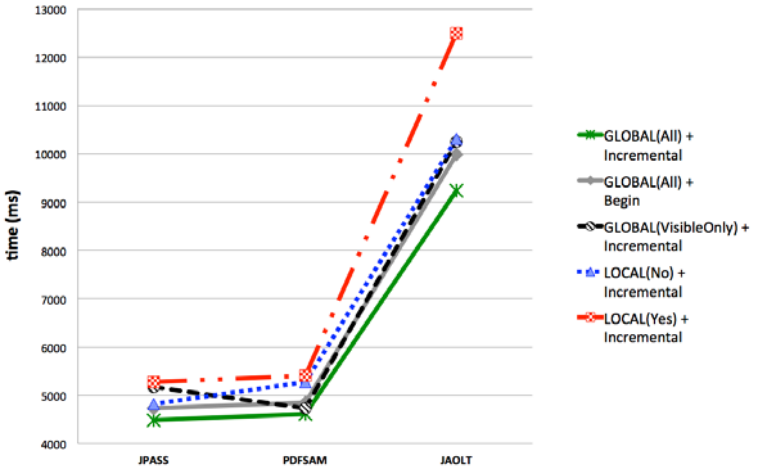


Fig. 7. Performance

5 Related Work

Research specifically targeting the extraction of information from GUIs appeared in [13,19,10]. All these works addressed the integration of Web data sources. The contribution most relevant to our work is the one by Nguyen et al. [13], which uses Naive Bayes and Decision Trees classifiers for automatically associating labels with data widgets in Web forms. Our contribution complements this work according to multiple aspects. First, the work by Nguyen et al. targets Web applications while we target desktop applications, which are designed partially following different principles. Second, the technique by Nguyen et al. learns how to retrieve associations from a training set. Unfortunately, training sets are notably hard to retrieve, especially for desktop applications. In addition, learning from a training set works properly only if it is large enough and well represents the GUIs that need to be analyzed. On the contrary, our approach does not rely on any training set, but it is defined according to standards and best practices about the design of GUIs. Thus, even if the effectiveness of our solution is in principle correlated to the quality of the interface under analysis, our algorithm is always applicable, and even with imperfect interfaces like the ones we analyzed, it provided high quality results.

Test data generation is an area that is gaining increasing attention and can be well targeted by our algorithm. The generation of test data for GUI test cases is a particularly hard problem where little automation is available. Nowadays the generation of test data for GUI test cases is a difficult and laborious process, in which test designers have to manually produce the inputs for data widgets. Automated GUI testing techniques either ignore generation of test data [16] or rely on fixed datapools of values [12,20]. As a consequence many behaviors cannot be automatically tested.

In this paper we early investigated the use of our algorithm to improve generation of test data for GUI test cases. We integrated the algorithm in AutoBlackTest, but the algorithm can be potentially integrated in any other GUI test case generation technique. To the best of our knowledge the work presented in [9] is the only one that infers the values that can be entered in data widgets with the objective of using this information in the scope of testing, even if with a different purpose. In fact they used this mapping to assure that GUI test scripts could be reused after GUI modifications.

Other people addressed generation of test data for GUI test cases from specifications, such as augmented use case descriptions [8,7] and enriched UML Activity Diagrams [18], but these descriptions are seldom available in the practice.

6 Conclusions

The GUI is a useful source of information that software analysis techniques should better exploit to increase their effectiveness.

In this paper we addressed the issue of automatically extracting the associations between descriptive and data widgets. We presented an algorithm that bases the extraction strategy on standards and common principles in the design of GUIs. Early empirical data collected with three case studies suggest that the algorithm can extract associations with high precision and recall.

The algorithm can be useful in many domains. In this paper we investigated the integration of the algorithm in the AutoBlackTest technique. Early results show that AutoBlackTest augmented with this algorithm produces GUI test cases that achieve greater coverage than the non-augmented version.

References

1. Buddi, http://buddi.digitalcave.ca/
2. jAOLT, http://code.google.com/p/jaolt/
3. Java look and feel design guidelines, http://java.sun.com/products/jlf/ed2/book/
4. JPass, http://metis.freebase.hu/jpass.html
5. PDFSAM, http://sourceforge.net/projects/pdfsam/
6. ISO 9241-12:1998 Ergonomic requirements for office work with visual display terminals (VDTs) - Part 12: Presentation of information (1998)
7. Bertolino, A., Gnesi, S.: Use case-based testing of product lines. SIGSOFT Softw. Eng. Notes 28, 355–358 (2003)
8. Fröhlich, P., Link, J.: Automated Test Case Generation from Dynamic Models. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 472–491. Springer, Heidelberg (2000)
9. Fu, C., Grechanik, M., Xie, Q.: Inferring types of references to gui objects in test scripts. In: Proceedings of the International Conference on Software Testing Verification and Validation (2009)
10. He, B., Chang, K.C.-C.: Statistical schema matching across web query interfaces. In: Proceedings of the International Conference on Management of Data (2003)

11. Lo, R., Webby, R., Jeffery, R.: Sizing and estimating the coding and unit testing effort for gui systems. In: Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results (1996)
12. Mariani, L., Pezzè, M., Riganelli, O., Santoro, M.: Autoblacktest: a tool for automatic black-box testing. In: Proceeding of the International Conference on Software Engineering (2011)
13. Nguyen, H., Nguyen, T., Freire, J.: Learning to extract form labels. In: Proceedings of the VLDB Endowment, 1 (August 2008)
14. Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., Carey, T.: Human-Computer Interaction. Addison Wesley (1994)
15. Sánchez Ramón, O., Sánchez Cuadrado, J., García Molina, J.: Model-driven reverse engineering of legacy graphical user interfaces. In: Proceedings of the International Conference on Automated Software Engineering (2010)
16. Shehady, R.K., Siewiorek, D.P.: A method to automate user interface testing using variable finite state machines. In: Proceedings of the International Symposium on Fault-Tolerant Computing (1997)
17. Tichy, W.F., Koerner, S.J.: Text to software: developing tools to close the gaps in software engineering. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (2010)
18. Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., Kazmeier, J.: Automation of gui testing using a model-driven approach. In: Proceedings of the 2006 International Workshop on Automation of Software Test (2006)
19. Wu, W., Yu, C., Doan, A., Meng, W.: An interactive clustering-based approach to integrating source query interfaces on the deep Web. In: Proceedings of the International Conference on Management of Data (2004)
20. Yuan, X., Memon, A.M.: Generating event sequence-based test cases using GUI run-time state feedback. IEEE Transactions on Software Engineering 36(1), 81–95 (2010)

Language-Theoretic Abstraction Refinement

Zhenyue Long[1,2,3,⋆], Georgel Calin[4], Rupak Majumdar[1], and Roland Meyer[4]

[1] Max Planck Institute for Software Systems, Germany
[2] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[3] Graduate University, Chinese Academy of Sciences
[4] Department of Computer Science, University of Kaiserslautern

Abstract. We give a language-theoretic counterexample-guided abstraction refinement (CEGAR) algorithm for the safety verification of recursive multi-threaded programs. First, we reduce safety verification to the (undecidable) language emptiness problem for the intersection of context-free languages. Initially, our CEGAR procedure overapproximates the intersection by a context-free language. If the overapproximation is empty, we declare the system safe. Otherwise, we compute a bounded language from the overapproximation and check emptiness for the intersection of the context free languages and the bounded language (which is decidable). If the intersection is non-empty, we report a bug. If empty, we refine the overapproximation by removing the bounded language and try again. The key idea of the CEGAR loop is the language-theoretic view: different strategies to get regular overapproximations and bounded approximations of the intersection give different implementations. We give concrete algorithms to approximate context-free languages using regular languages and to generate bounded languages representing a family of counterexamples. We have implemented our algorithms and provide an experimental comparison on various choices for the regular overapproximation and the bounded underapproximation.

1 Introduction

Counterexample-guided abstraction refinement (CEGAR) has become a widely applied paradigm for automated verification of systems [2, 5, 14]. While CEGAR has had a lot of successes in the analysis of single-threaded programs (most notably, device drivers), its application to *recursive multi-threaded* programs has been relatively unexplored.

We present a uniform language-theoretic view of abstraction refinement for the safety verification of recursive multi-threaded programs. First, with known encodings [3, 8], we reduce the safety verification problem to checking if the intersection of a set of context-free languages (CFLs) is empty. This is a well-known undecidable problem (and equivalent to safety verification of recursive

⋆ This work is supported by the National Natural Science Foundation of China (Grant No.60833001) and by a fellowship at the Max-Planck Institute for Software Systems.

J, de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 362–376, 2012.

multi-threaded programs, which is also undecidable [26]). Then we give a purely language-theoretic abstraction refinement algorithm for checking emptiness of such an intersection.

To illustrate the idea of our CEGAR loop, consider CFLs L_1 and L_2 for which we would like to check whether $L_1 \cap L_2 = \emptyset$. In our algorithm, we have to specify the following steps: (a) how to abstract the intersection of two CFLs and check that the abstraction is empty? (b) in case the abstraction is not empty, how to refine it by eliminating spurios counterexamples?

Abstraction and Checking. To abstract the context-free intersection $L_1 \cap L_2$ we rely on *regular* overapproximations of the component languages. Once we have approximated one of the languages, say L_1, by a regular language R_1 such that $L_1 \subseteq R_1$, we can check emptiness of $R_1 \cap L_2$. Because of the overapproximation, $R_1 \cap L_2 = \emptyset$ entails $L_1 \cap L_2 = \emptyset$.

Counterexample Analysis and Refinement. Suppose $A = R_1 \cap L_2 \neq \emptyset$. In the analysis and refinement step, we check if the counterexample to emptiness is genuine, and try to eliminate any string in the intersection A that is not in $L_1 \cap L_2$. A naive approach takes an arbitrary word in A (a potential counterexample) and checks if it is in $L_1 \cap L_2$. We generalize this heuristic to produce candidate counterexamples B such that we can effectively check if $L_1 \cap L_2 \cap B = \emptyset$. The advantage is that we consider (and rule out) the potentially infinite set B in one step. Refinement simply removes B from A.

Our abstraction refinement algorithm runs these two steps in a loop with one of the following outcomes. Either the abstract intersection is eventually found to be empty (the system is safe), or the counterexample analysis finds a bug (the system is unsafe), or (because the problem is undecidable) the algorithm loops forever. Unlike other abstraction-refinement algorithms [2, 5, 14], it is not the case that the abstraction always overapproximates the original program. The proof of soundness shows that refinement steps never remove buggy behaviors from the abstraction.

We give concrete constructions for the abstraction and counterexample analysis steps. To find a regular overapproximation to a context-free language, we adapt the construction of a downward closure of a CFL [28] and combine it with a graph-theoretic heuristic from [7]. We show that our construction produces regular approximations that lie between the original CFL and its downward closure (and is tighter w.r.t. set inclusion than the construction in [7]).

For the counterexample analysis, we use *bounded languages* [10] to represent an infinite set of counterexamples. Bounded languages are regular sets of the form $w_1^* w_2^* \ldots w_k^*$, for words $w_1, \ldots, w_k$. Given a bounded language B, checking $L_1 \cap L_2 \cap B = \emptyset$ is known to be decidable and NP-complete[8, 10]. What is an appropriate bounded language? We experimented with two algorithms. First, using a construction from [9, 20], we constructed, from a CFL L, a bounded language B such that $L \cap B$ has the same Parikh image as L. Unfortunately, this construction did not scale well in the implementation. Instead, we relied on a simple heuristic based on pumping derivation trees.

We have implemented our algorithm, and we have tried our implementation for the safety verification of several recursive multi-threaded programs. Our first class of examples models variants of a bluetooth driver [18] (also studied in [22, 25, 27]). Our second class contains example programs written in Erlang [1]. Erlang programs communicate via message passing, and are naturally written in a functional, recursive style. (In our experiments, we assume a rendezvous communication rather than asynchronous communication.) Most of the correctness properties we considered could be proved using the regular approximation. When there was a bug, the bounded language based procedure could find it.

Related Work. Analysis techniques for recursive multi-threaded programs can be categorized in four main classes. First, *context-bounded* reachability techniques [19, 23], and their generalizations using reachability modulo bounded languages [8, 9], provide a systematic way to underapproximate the reachable state space, and have proved useful in finding bugs. Second, for specific structural restrictions on the communication, one can get decidability results [4, 16, 17, 22, 25]. Third, there are some techniques to abstract the behaviors of these programs. For example, [3] explores language-based approximations by abstracting queue contents with their Parikh images. Finally, although abstraction-refinement has been studied for *non-recursive* multi-threaded programs before [6, 11–13], its systematic study for recursive multi-threaded programs has been little investigated.

Our constructions for regular overapproximations for context-free languages are inspired by language-theoretic constructions for the downward closure of context-free languages [28], together with approximation techniques originating in speech processing [7, 21]. We use bounded languages to represent families of counterexamples using ideas from [8, 9].

2 From Safety Verification to Language Emptiness

We recall the reduction from the safety verification of recursive multi-threaded programs to the emptiness problem for the intersection of context free languages.

2.1 Preliminaries

We assume familiarity with language theory (see, e.g. [15]) and only briefly recall the basic notions on regular and context free languages that we shall need in our development. A *context free grammar* (CFG) is a tuple $\langle N, \Sigma, \mathcal{P}, S\rangle$ where N is a finite and non-empty set of *non-terminals*, Σ is an alphabet of *terminals*, $\mathcal{P} \subseteq N \times (N \cup \Sigma)^*$ is the set of *production rules*, and $S \in N$ is the *start non-terminal*. We typically write $X \to w$ to denote a production $(X, w) \in \mathcal{P}$ and use $X \to w_1 | \ldots | w_k$ for $\{(X, w_i) \mid 1 \leq i \leq k\} \subseteq \mathcal{P}$.

Given two strings $u, v \in (N \cup \Sigma)^*$, we write $u \Rightarrow v$ for the fact that v *is derived from* u by an application of a production rule. Technically, $u \Rightarrow v$ if there are a non-terminal $X \in N$, a production rule $X \to w$ in $\mathcal{P}$, and strings $y, z \in (N \cup \Sigma)^*$ so that $u = yXz$ and $v = ywz$. The reflexive and transitive closure of $\Rightarrow$ is written $\Rightarrow^*$.

The *language* $L(G)$ *of a grammar* G is the set of terminal words that can be derived from the start symbol: $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$. A language $L \subseteq \Sigma^*$ is *context-free* if there is a context-free grammar G such that $L = L(G)$. We denote the class of all context-free languages by CFL. A context-free grammar is *regular* if all its productions are in $N \times (\Sigma^* N \cup \{\epsilon\})$. A language is *regular* if $L = L(G)$ for some regular grammar. The class of *all regular languages* is REG.

Example 1. Consider the grammars $G_1 = \langle\{S_1\}, \{a, b\}, \{S_1 \to abS_1b \mid \epsilon\}, S_1\rangle$ and $G_2 = \langle\{S_2\}, \{a, b\}, \{S_2 \to aS_2b \mid baS_2b \mid \epsilon\}, S_2\rangle$ that define the languages $L(G_1) = \{(ab)^n b^n \mid n \in \mathbb{N}\}$ and $L(G_2) = \{(a + ba)^n b^n \mid n \in \mathbb{N}\}$. Both languages are context free, and it can be shown that they are not regular.

The following results are well-known (see, e.g., [15]).

Theorem 1. *Let* L, L' *be context-free languages and* R *a regular language.*

1. *It is undecidable whether* $L \cap L' = \emptyset$.
2. *It is decidable whether* $L \cap R = \emptyset$.

2.2 From Programs to Context Free Languages

We encode the behaviour of recursive multi-threaded programs by an intersection of context-free languages. We consider both asynchronous communication via a shared memory and synchronous communication via rendezvous-style message exchange. The shared memory is modelled by a finite set of shared variables that range over finite data domains. This covers programs with finite data abstractions. For rendevous-style communication, we use actions $m!$ to let a thread broadcast a message m taken from a finite set M. To receive the message, all other threads have to execute a corresponding receive action $m?$ in lock step.

Encoding Programs. We illustrate the encoding of shared memory concurrency by means of an example, rendevous synchronisation immediately translates into language intersection.

Algorithm 1. Shared Memory "Toy" Program

initially $x = 0$ and $y = 0$

T1:	T2:
$x := \neg y$	$y := \neg x$
if $(*)$ **then**	**if** $(*)$ **then**
call T1	call T2
$x := \neg y$	$y := \neg x$

assert "$x = 0 \vee y = 0$"

Table 1. Context-free grammars G_{T1} encoding routine $\mathsf{T1}$ (left) and G_X encoding variable x (right). G_{T2} and G_Y are similar. $\mathsf{A_X}$ is described in the text below.

$$\begin{aligned}
\mathsf{ST1} &\to \mathsf{T1}.(r,x,1)\\
\mathsf{T1} &\to \mathsf{A_X}.\mathsf{IF}\\
\mathsf{IF} &\to \mathsf{T1}.\mathsf{FI} \mid \mathsf{FI}\\
\mathsf{FI} &\to \mathsf{A_X}
\end{aligned}
\qquad
\begin{aligned}
\mathsf{X}_{=0} &\to (r,x,0).\mathsf{X}_{=0} \mid (w,x,1).\mathsf{X}_{=1}\\
&\mid (w,x,0).\mathsf{X}_{=0} \mid \mathsf{L_X}.\mathsf{X}_{=0} \mid \epsilon\\
\mathsf{X}_{=1} &\to (r,x,1).\mathsf{X}_{=1} \mid (w,x,0).\mathsf{X}_{=0}\\
&\mid (w,x,1).\mathsf{X}_{=1} \mid \mathsf{L_X}.\mathsf{X}_{=1} \mid \epsilon
\end{aligned}$$

The program given by Algorithm 1 above consists of two threads that execute the procedures $\mathsf{T1}$ and $\mathsf{T2}$, respectively. They communicate through shared Boolean variables x, y that take values in $\{0,1\}$. The first thread assigns the negation of y to x. Based on a non-deterministic choice, it then calls itself recursively and finally repeats the assignment. Thread $\mathsf{T2}$ is symmetric, substituting y for x. We assume that each statement is executed atomically and concurrency is represented by interleaving. We are interested in the safety property that the assertion $x = 0 \vee y = 0$ holds upon termination.

We model the program's behaviour by an intersection of four context-free languages as defined by the grammars in Table 1. We explain the behaviour of G_{T1}. From its start location $\mathsf{ST1}$ (say the main routine) the thread calls procedure $\mathsf{T1}$. In $\mathsf{T1}$ it first assigns $\neg y$ to x. This is encoded by the non-terminal $\mathsf{A_X}$ that we discuss below in more detail. When the assignment has been executed, the thread proceeds with the *if* statement. If the non-determinism decides to call $\mathsf{T1}$ again, the thread proceeds with symbol $\mathsf{T1}$ and pushes the *endif* location onto the stack, indicated by $\mathsf{T1.FI}$. Otherwise, a second $\mathsf{A_X}$ statement terminates this $\mathsf{T1}$ call. When the overall execution terminates, the thread runs a final $(r,x,1)$ action which checks the assertion violation. We discuss it below.

Since we model the variables x and y by separate grammars, we split the assignment $x := \neg y$ into two actions, each modifying a single variable. First, a read $(r,y,0)$ or $(r,y,1)$ determines the value of y. A following write $(w,x,1)$ or $(w,x,0)$ finishes the assignment. This yields

$$\begin{aligned}
\mathsf{A_X} &\to \mathsf{L_{T1}}.(r,y,0).(w,x,1).\mathsf{L_{T1}} \mid \mathsf{L_{T1}}.(r,y,1).(w,x,0).\mathsf{L_{T1}}\\
\mathsf{L_{T1}} &\to (r,x,0).\mathsf{L_{T1}} \mid (r,x,1).\mathsf{L_{T1}} \mid (w,y,0).\mathsf{L_{T1}} \mid (w,y,1).\mathsf{L_{T1}} \mid \epsilon.
\end{aligned}$$

Here, $\mathsf{L_{T1}}$ lets the first thread loop on the actions of the second. Similarly for G_X, writes and reads of y are synchronized via $\mathsf{L_X}$.

From Safety to Emptiness. The safety verification problem takes as input a multi-threaded program, a tuple of locations (one for each thread), and an assertion, and asks if the assertion holds when the threads are simultaneously at these locations. With the above encoding, there is an execution of the program reaching the locations specified in the safety verification problem so that the assertion fails iff there is a word common to the languages of all context-free grammars [26].

In the above example, the assertion is violated if both threads finished their execution and the variables had values $x = y = 1$. Note that the latter condition matches the execution of the reads $(r, x, 1)$ and $(r, y, 1)$ in the grammars G_{T1} and G_{T2}. Indeed, program safety is confirmed by

$$L(G_{\mathsf{T1}}) \cap L(G_{\mathsf{T2}}) \cap L(G_X) \cap L(G_Y) = \emptyset.$$

To check it note that $L(G_{\mathsf{T1}}) = \{L(\mathsf{A_X})^n.L(\mathsf{A_X})^n.(r, x, 1) \mid n \in \mathbb{N}\}$. The simple intuition to why the assertion holds is as follows. The first assignment changes the variables' valuation from $x = y = 0$ to either $x = 1, y = 0$ or $x = 0, y = 1$. This valuation is never altered throughout the rest of the execution.

3 The Abstraction-Refinement Procedure

We now give an abstraction-refinement algorithm to check whether the intersection $L_1 \cap L_2$ of two CFLs L_1 and L_2 is empty.

Algorithm 2. LCegar: test emptiness of context free language intersection

Input: Context-free languages L_1 and L_2
Output: "empty" or "non-empty"

1: $A_1 := \mathsf{mkreg}(L_1) \cap L_2$, $A_2 := \mathsf{mkreg}(L_2) \cap L_1$, $B_1 := \emptyset$; $B_2 := \emptyset$
2: **loop**
3: **Invariant:** $w \in B_1 \cup B_2 \;\Rightarrow\; w \notin L_1 \cap L_2$
4: **if** $A_1 = \emptyset$ or $A_2 = \emptyset$ **then**
5: **return** "empty"
6: **else**
7: $B_1 := \mathsf{gencx}(A_1)$ and $B_2 := \mathsf{gencx}(A_2)$
8: **if** $L_1 \cap L_2 \cap B_1 \neq \emptyset$ or $L_1 \cap L_2 \cap B_2 \neq \emptyset$ **then**
9: **return** "non-empty"
10: **else**
11: $A_1 := A_1 \setminus (B_1 \cup B_2)$, $A_2 := A_2 \setminus (B_1 \cup B_2)$

3.1 CEGAR Loop

The abstraction refinement procedure we present in Algorithm 2 takes as input two CFLs L_1 and L_2 and correctly returns "empty" or "non-empty" upon termination, reflecting whether the intersection $L_1 \cap L_2$ is empty or not. Termination, however, is not guaranteed due to the undecidability of the problem.

The key idea in our approach is to *abstract and refine the intersection* $L_1 \cap L_2$. The algorithm takes two parameters that define this approximation. For the initial abstraction, $\mathsf{mkreg}\colon \text{CFL} \to \text{REG}$ computes regular overapproximations of the context-free languages of interest, i.e., we require $L \subseteq \mathsf{mkreg}(L)$ for any CFL L. Function $\mathsf{gencx}\colon \text{CFL} \to \text{REG}$ serves in the refinement step. It isolates a regular language $\mathsf{gencx}(L)$ from the current overapproximation L of $L_1 \cap L_2$. Intuitively, $\mathsf{gencx}(L)$ contains candidate words that may lie in the intersection.

To check whether the candidates are spurious, the problem $L_1 \cap L_2 \cap \mathsf{gencx}(L) = \emptyset$ is required to be decidable for all CFLs L, L_1, L_2.

We now explain the algorithm in detail. Initially (Line 1), it overapproximates $L_1 \cap L_2$ by $A_1 = \mathsf{mkreg}(L_1) \cap L_2$ and $A_2 = L_1 \cap \mathsf{mkreg}(L_2)$ using the regular overapproximations $\mathsf{mkreg}(L_i)$. It should be noted that the A_i are CFLs. If either approximation is empty, the algorithm stops and returns "empty". Otherwise, it computes (regular) approximations $B_i = \mathsf{gencx}(A_i)$ from the (non-empty) overapproximations (Line 7). If the intersection $L_1 \cap L_2$ can be proved non-empty with words in B_1 or B_2 (Line 8), the algorithm returns "non-empty". Otherwise, we refine the approximations A_i by removing the B_i and run the loop again. By the assumptions on mkreg and gencx, each step of the algorithm is effective.

The choice of mkreg determines the granularity of the initial abstraction, and thus how quickly one can prove emptiness (in case the intersection is empty). The choice of gencx influences how fast counterexamples are found (and whether they are found at all).

Theorem 2 (Soundness). *If on input L_1, L_2 Algorithm 2 outputs "empty" then $L_1 \cap L_2 = \emptyset$, and if it outputs "non-empty" then $L_1 \cap L_2 \neq \emptyset$.*

Proof. The key to the proof is the invariant on Line 3. It states that only words outside the intersection $L_1 \cap L_2$ are removed from the A_i. Therefore $A_i \supseteq L_1 \cap L_2$ always holds. For the first iteration, the invariant is trivial. For an arbitrary iteration, $B_1 \cup B_2$ is removed from (e.g.) A_1 only when $L_1 \cap L_2 \cap B_i = \emptyset$ for $i = 1, 2$. Thus any strings removed from A_1 do not belong to $L_1 \cap L_2$ and the invariant holds.

If, e.g., $A_1 = \emptyset$ we conclude that $L_1 \cap L_2 = \emptyset$ since A_1 is throughout the program an overapproximation of the intersection. Thus the "empty" output is sound. On the other hand, any strings in $L_1 \cap L_2 \cap B_i$ are also in $L_1 \cap L_2$, thus the check on Line 8 ensures the "non-empty" output is sound. □

Example 2. Consider languages $L(G_1)$ and $L(G_2)$ from Example 1. Suppose we approximate them by regular sets $(ab)^*b^*$ and $(a + ba)^*b^*$, respectively. The intersections $(ab)^*b^* \cap L(G_2)$ and $L(G_1) \cap (a + ba)^*b^*$ are both non-empty. Suppose gencx returns $(ab)^*b^*$ for the first intersection. We can compute that $L(G_1) \cap L(G_2) \cap (ab)^*b^* = \emptyset$. In the next iteration, the new approximation $A_1 = \{(ab)^n b^n | n \in \mathbb{N}\} \setminus (ab)^*b^* = \emptyset$, and we conclude that $L(G_1) \cap L(G_2) = \emptyset$.

Since checking emptiness of the intersection is undecidable, the algorithm can run forever. But what happens if $L_1 \cap L_2 \neq \emptyset$ holds? In this case the algorithm is guaranteed to terminate and return "non-empty" if gencx satisfies the following additional enumeration property.

Definition 1 (Enumerator). *A sequence $(L_i)_{i \in \mathbb{N}}$ of languages* **enumerates** *a language L if for every $w \in L$ there is an index $i \in \mathbb{N}$ so that $w \in L_i$.*

Let $(L_{1,i})_{i \in \mathbb{N}}$, $(L_{2,i})_{i \in \mathbb{N}}$ be the sequences of languages generated by $\mathsf{gencx}(A_1)$ and $\mathsf{gencx}(A_2)$ when running Algorithm 1 on the input languages L_1 and L_2.

Proposition 1 (Semidecider). *If* $(L_{1,i} \cup L_{2,i})_{i \in \mathbb{N}}$ *enumerates* $L_1 \cap L_2 \neq \emptyset$ *then Algorithm 2 terminates with output "non-empty" on input* L_1, L_2.

3.2 The Regular Approximator mkreg

We now describe our implementation of mkreg, the regular overapproximation of a context-free language.

Step I: Intuition: Downward Closures. Given strings $u, v \in \Sigma^*$, we define $u \preceq v$ if u is a (not necessarily contiguous) substring of v. For example, $abd \preceq aabccd$. For a language L, we define the downward closure of L, denoted by $L{\downarrow}$, as $L{\downarrow} = \{u \in \Sigma^* \mid \exists v \in L. u \preceq v\}$. It is known that $L{\downarrow}$ is effectively regular for a CFL L [28], and clearly $L \subseteq L{\downarrow}$. Thus, the downward closure $-{\downarrow}$ is an immediate candidate for mkreg.

Step II: A Modification to the Downward Closure. Unfortunately, the downward closure is usually too coarse and leads to many spurious counterexamples. The problem occurs already when the original language is regular; for example, the downward closure of $(ab)^*$ is $(a+b)^*$. Further, since our downward closure construction is doubly exponential in the worst case, the obtained approximation can get too big to apply further operations in non-trivial examples. We encountered both these shortcomings in our experiments.

As a first modification, we "tightened" the downward closure algorithm to provide a regular approximation that lies (w.r.t. set inclusion) between the CFL and its downward closure. We observe that for $t \in \Sigma$, the downward closure $L(t){\downarrow} = \{t, \epsilon\}$ can drop the t. In the inductive construction of the downward closure, this introduces too many new (sub)words. We show how to avoid dropping some letters in the downward closure computation.

Our modification, called pseudo-downward closure (PDC), constructs a finer regular overapproximation $L{\Downarrow}$ of a CFL L. The idea is to preserve contiguous subwords. We proceed by iterating over all the grammar non-terminals. More precisely, we set $L(t){\Downarrow} = \{t\}$ for a letter $t \in \Sigma$. To ease the definition, let $-{\Downarrow}$ distribute over concatenation ($L(u.v){\Downarrow} = L(u){\Downarrow}.L(v){\Downarrow}$) and set

$$L(X){\Downarrow} = \begin{cases} \Sigma(X)^* & \text{if } X \Rightarrow^* uXvXw \\ (\bigcup L(u){\Downarrow})^*.(\bigcup_{X \Rightarrow M, X \not\triangleleft M} L(M){\Downarrow}).(\bigcup L(v){\Downarrow})^* & \text{if } X \Rightarrow^* uXv \\ \bigcup_{X \Rightarrow M, X \not\triangleleft M} L(M){\Downarrow} & \text{otherwise.} \end{cases}$$

$\Sigma(X) \subseteq \Sigma$ denotes the set of all letters that appear in some word derived from X. The unions $\bigcup L(u){\Downarrow}$ and $\bigcup L(v){\Downarrow}$ range over the *shortest words* so that X reproduces itself via a derivation $X \Rightarrow^* uXv$. $X \not\triangleleft M$ means there is no derivation $M \Rightarrow^* uXv$. Finally, we require the first case to have precedence over the second.

Proposition 2 below states that $-{\Downarrow}$ is also a candidate for mkreg, and a better approximation than $-{\downarrow}$. Its proof is a induction on the structure of the grammar.

Proposition 2. *Given a CFG* $G = (N, \Sigma, \mathcal{P}, S)$, $L(G) \subseteq L(G){\Downarrow} \subseteq L(G){\downarrow}$.

Step III: Refinement by Cycle Breaking. We can further tighten our PDC construction as follows. Consider the grammar G defined by

$$A \to aAbAc \mid t \tag{1}$$

where lower-case letters denote terminals. Each word in the language $L(G)$ is either t, or starts with an a, has a b in the middle, and ends with a c. However, $L(G)\Downarrow = (a+b+c+t)^*$, and the PDC construction loses the order of the letters in the original language.

We augment the PDC construction with the following insight [7]. Given a CFG G in Chomsky normal form, we can construct a regular over-approximation of $L(G)$ by replacing each rule $A \to BC$ with a rule $A \to R_B C$, where R_B generates a regular over-approximation of $L(B)$. After the replacement, the grammar will be right regular and the new language will over-approximate $L(G)$. If there is no production $B \Rightarrow^* uAv$, we can inductively compute an over-approximation R_B of B. In the case when $B \Rightarrow^* uAv$ we use the PDC construction to compute an overapproximation of B.

The intuition behind the construction is similar to the *cycle-breaking* heuristic to approximate CFLs by regular languages used in speech processing [7]. Our construction below computes a regular approximation that is guaranteed to be as tight as the construction in [7].

To clarify the intuition, we construct a directed graph from the CFG as follows. The nodes of the graph are the non-terminals of the grammar. For each rule $A \to BC$ in the grammar, we have an edge in the graph from A to B labeled l (for left) and an edge from A to C labeled r (for right). A simple cycle in this graph is called mono-chromatic if it only contains edges marked l or r, and duo-chromatic otherwise. Our construction "breaks" every duo-chromatic cycle in the graph as follows. It picks an l-edge (A, B) in the cycle, and replaces the rule $A \to BC$ from which the (A, B) edge was constructed with $A \to R_B C$. The language of the new nonterminal R_B is $L(B)\Downarrow$. We denote this approximation by $L(G)\downdownarrows$. To give an example, if we turn the grammar in Equation (1) into Chomsky normal form, we obtain $L(A)\downdownarrows = t + a(a+b+c+t)^* b (a+b+c+t)^* c$.

Proposition 3. *Given a CFG G, we have* $L(G) \subseteq L(G)\downdownarrows \subseteq L(G)\Downarrow$.

The proof is again a direct application of structural induction.

3.3 The Counterexample Generator gencx

We now give an algorithm to generate families of counterexamples in case A_1 and A_2 are non-empty. A naive idea is to lexicographically enumerate words in A_1 and A_2 and to check if one of them is in $L_1 \cap L_2$. Since the length of a path leading to the unsafe location is finite, this approach guarantees termination (the sequence is clearly an enumerator for $L_1 \cap L_2$). However, enumeration-based approaches do not scale, even if the language is finite. For example, depending on the choice of the w_i's, a language of the following type has upto 2^k words:

$$(w_1 + \epsilon) \cdot (w_2 + \epsilon) \cdots (w_k + \epsilon) \tag{2}$$

Instead, we use *elementary bounded languages* (EBLs) [10] to represent families of counterexamples. EBLs are regular languages of the form $w_1^* w_2^* \cdots w_k^*$ for some fixed words $w_1, \cdots, w_k \in \Sigma^*$. For CFLs L_1 and L_2 and an EBL B, checking if $L_1 \cap L_2 \cap B = \emptyset$ is NP-complete [8]. In case of Language (2), the bounded language $w_1^* \ldots w_k^*$ contains all the words in (2) (and more). In general, an EBL captures an infinite number of potential counterexamples.

Which EBL should one choose? We first implemented an algorithm from [9] which computes, given a CFL L, an EBL B such that for every word $w \in L$, there is a permutation of w in $L \cap B$. (That is, the commutative images of $L \cap B$ and L coincide. Recall that the commutative image of a word w is an integer vector that represents the number of times each alphabet letter occurs in w.) Intuitively, the EBL B captures "many" behaviors of L. Unfortunately, our experiments indicated that this construction does not scale well. Therefore, our implementation makes use of a simple heuristic. The idea is to pump derivation trees as follows.

Starting with a CFG $G = \langle N, \Sigma, \mathcal{P}, S \rangle$ for L, we first construct an initialized *partial derivation tree* up to a fixed depth. A partial derivation tree is a tree whose nodes are labeled with symbols from $N \cup \Sigma$ with the following property. Each leaf is labeled with a symbol from $N \cup \Sigma$. Each internal node is labeled with a non-terminal from N, and if an internal node is labeled with $T \in N$ and has k children labeled with $A_1, \ldots, A_k \in N \cup \Sigma$, then $T \to A_1 \ldots A_k$ is a production in $\mathcal{P}$. The partial derivation tree is *initialized* if its root is labeled with S. A partial derivation tree corresponds to a (partial) derivation of the grammar G, and conversely, each (partial) derivation of the grammar defines a partial derivation tree. The *yield* of a partial derivation tree is the word of symbols at the leaf nodes. More formally, for a leaf l labeled by $\sigma \in N \cup \Sigma$ we set $\mathit{yield}(l) = \sigma$. For an internal node n with k children $n_1, \ldots, n_k$, we define $\mathit{yield}(n) = \mathit{yield}(n_1) \cdot \mathit{yield}(n_2) \cdot \ldots \cdot \mathit{yield}(n_k)$.

A partial derivation tree t is *pumpable* if its root is labeled by a non-terminal A and some (not necessarily immediate) child of t again carries label A. For the sake of clarity, we denote the (not necessarily unique) descendant node labeled by A by t_A. Words x, z are then called *pump-words* for the pumpable tree t with descendant t_A if the derivation corresponding to t can be written as $A \Rightarrow^* xAz \Rightarrow^* xyz$ for some $y \in (N \cup \Sigma)^*$. Here, the non-terminal A in the derivation labels node t_A.

The EBL is computed in two steps. We first construct an initialized partial derivation tree up to some fixed depth, and then traverse this tree with a depth-first search. The corresponding procedure traverse in Algorithm 3 is called with the root of the initialized partial derivation tree, an empty list of words, and an empty initial word ε. It returns a (possibly empty) word w and a list of words $[w_1, \ldots, w_k]$. From these words, we construct the required elementary bounded language $h(w_1)^* \ldots h(w_k)^* h(w)^*$ using the following homomorphism h. Since the words are defined over $N \cup \Sigma$ but we look for a language over Σ, homomorphism h replaces non-terminals $A \in N$ by words $w_A \in \Sigma^*$ that can be derived from A. Terminals are left unchanged, $h(\sigma) = \sigma$ for $\sigma \in \Sigma$.

Algorithm 3. traverse

Input: partial derivation tree t, word list l, current word w
Output: pair of word $w \in (N \cup \Sigma)^*$ and list of words l'

```
1: match t with
2: | leaf labeled with σ ∈ N ∪ Σ : return ⟨w · σ, l⟩
3: | internal node labeled with A :
4:     if t is pumpable with descendant t_A and pump words x, z
5:     then let ⟨w_t, l_t⟩ = traverse(t_A, [ ], ε)
6:       return ⟨ε, l@[w, x]@l_t@[w_t, z]⟩
7:     otherwise for the children t_1, ..., t_k of t,
8:       let ⟨w_1, l_1⟩ = traverse(t_1, l, w)
9:       ...
10:      let ⟨w_k, l_k⟩ = traverse(t_k, l_{k−1}, w_{k−1})
11:      return ⟨w_k, l_k⟩
```

The procedure recursively traverses the partial derivation tree, collecting the yield of the tree in the word w. However, when it sees a "cycle" in the derivation tree (i.e., a subtree that can be pumped), it pumps its pump-words. Pumping moves the partial yield w constructed so far to the list of words (Line 6). Therefore, a word w that is returned actually gives the partial yield of the nodes visited after the last pumping situation. This explains why $h(w)^*$ is added to the end of the EBL and why w_t is placed behind l_t in Line 6. As a special case, if the partial derivation tree consists of a single node containing only the start symbol, the algorithm returns w^* for some word w in the language.

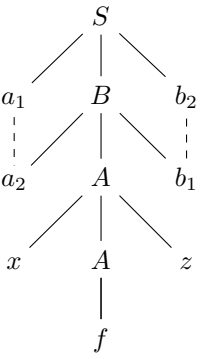


Fig. 1. Traverse Sample

As clarification, applied to the partial derivation tree depicted in Figure 1, the algorithm returns $\langle b_1 \cdot b_2, [a_1 \cdot a_2, x, f, z]\rangle$ with x and z being pump words for the given derivation tree and a_1a_2 as well as b_1b_2 being partial yields w.

4 Experiments

We have implemented the language-theoretic CEGAR algorithm and tested our implementation on the shared memory example in Section 2, a set of Erlang examples from [1], and the bluetooth driver examples from [22, 25, 27]. We manually convert programs to the input format of our tool (a description of context free grammars). We compare the results of the PDC and cycle-breaking algorithms for mkreg and we use the simplified EBL generation algorithm for gencx. To check emptiness of $L_1 \cap L_2 \cap B$ (for CFLs L_1 and L_2 and EBL B), we use the algorithm of [8] and reduce the check to a satisfiability problem in Presburger arithmetic. We use Yices to check satisfiability of this formula.

All experiments were carried on an Intel Core2 Q9400 PC with 8GB memory, running 64-bit Linux (Ubuntu 11.10 x86_64). Tables 2 and 3 describe our results.

4.1 Recursive Multi-threaded Programs

We applied our algorithm to a set of recursive multi-threaded programs: the shared memory program in Section 2 and some selected Erlang programs. We chose these examples since they provide a good test suite for checking both our method's bug finding and proving capabilities. For each test we used both the *pseudo downward closure* (PDC) and *cycle breaking* (CB) overapproximations.

Toy Example (from Section 2). For the shared memory program in Section 2 we produced 4 CFLs, for $\mathsf{T1}$, $\mathsf{T2}$, x and y, with a total of 118 production rules. It took our implementation about 16 seconds to report the system's safety by adopting the PDC approach and about 26 seconds by the CB approach. Note that this example does not fall into the subclass considered in [4].

Peterson Mutual Exclusion Protocol (from [29], made recursive). In this example, two processes try to acquire a lock. The one which receives it can enter the critical section to perform operations on shared variables, then frees the lock. The code is written in functional style with tail-recursive calls in each component. The checked property is that at any time, at most one process is in the critical section. The model comprised 4 CFLs with 242 production rules. LCegar reported the system's safety in 6.8 seconds by adopting the PDC closure, and in 0.2 seconds by using the CB closure.

Resource Allocator (from [1], pp. 81, 111). A resource allocator (RA) manages a number of resources and handles "alloc" and "free" messages sent from clients. When it receives an "alloc" message, the RA checks if there is a free resources in the system. If yes, it replies to the client with the resource id, and also marks that the resource has been allocated; otherwise it replies that no free resources are available. When it receives a "free" message with a resource id from a client, the RA checks if the resource is actually held by the client. If yes, the resource is freed, otherwise an error is reported to the client. The safety property checked was that the server would not allocate more resources to clients than the current free resources in the system. We produced 2 CFLs for the server and resource with a total of 100 production rules from the original program. Though simple, LCegar did not terminate by adopting the PDC over-approximation. It took LCegar 0.5 seconds to report the system's safety by using the CB approach.

In the modified version of the resource allocator (MoRA), the situation that a client can exit normally or abnormally is handled. When allocating a resource id to a client, MoRA makes a link between server and client, and traps the exit signal of the client. Once the client leaves without freeing the resource, the MoRA will detect the event, free the resource on server side, and unlink the client. The property to verify was identical to the one for the RA. This more complicated example needed 5 CFLs with 239 production rules. It took LCegar 31 seconds to report safety by using the CB closure and PDC did not terminate.

Table 2. Experimental results for recursive multi-threaded (Erlang) programs

	SharedMem		Mutex		RA		Modified RA		TNA	
	PDC	CB	PDC	CB	PDC	CB	PDC	CB	PDC	CB
CFL	4		4		2		5		3	
Terminal	8		16		20		22		17	
Non-Ter	22		27		22		32		23	
Rule	118		242		100		239		93	
Time	15.7s	26.0s	6.8s	0.2s	N/A	0.5s	N/A	31.2s	0.8s	0.3s

Table 3. Experimental results for Bluetooth drivers

		Version 1		Version 2		Version 3 (2A1S)		Version 3 (1A2S)	
		PDC	CB	PDC	CB	PDC	CB	PDC	CB
w/o Heuri	CFL	7		9		9		8	
	Terminal	17		26		25		22	
	Non-Ter	29		47		47		39	
	Rule	362		839		846		585	
	Time	19s	18s	109m57s	96m48s	81m7s	77m21s	3m50s	3m55s
w/ Heuri	CFL	7		9		x		8	
	Terminal	17		26		x		22	
	Non-Ter	29		47		x		39	
	Rule	285		591		x		408	
	Time	unknown		56s	50s	x	x	7s	7s

Telephone Number Analyzer (from [1], p. 109). The telephone number analyzer (TNA) running on the server side handles "lookup" or "add_number" requests from the telephone ends. When it receives a request, it performs the corresponding action in a try-catch block. The example tries to show that the try-catch block can guard against the inadvertent programming errors. However, it also mentions that a malicious program can crash the server by sending an incorrect process id. The reason is that TNA does not check if the id in the message content is the same as its sender's. We modeled the program with 3 CFLs and a total of 93 production rules to spot the same bug in 0.8s (PDC) and 0.3s (CB).

4.2 Bluetooth Drivers

We considered the bluetooth driver [18] and its variations studied before through various methods [4, 24, 27]. Once LCegar terminates, it reports safety or a path leading to the buggy location of the system. Note that in contrast to bounded context-switch approaches, LCegar looks for error traces without an a priori context-switching bound, and thus can prove correctness of the protocol as well. However, our language-theoretic computations take much more time than bounded context-switching.

Table 3 shows the experimental results for the bluetooth drivers. For each program, we use the two kinds of overapproximations discussed.

The 2nd version and the erroneous 3rd version (1A2S) prove to be the most difficult to handle for our full fledged method. For this reason we considered an unsound heuristic restricting context switches at basic block boundaries. This heuristic sped up the tool and found the bugs. However, in the first version, our method reported "unknown" since the heuristic method is an underapproximation, and therefore, cannot report the system's correctness when no bug is found. If the method does not find a bug, we therefore run the original (sound) version of the algorithm. For example, we used the non-optimized algorithm for the 3rd version of the driver with 2 adders and 1 stopper (2A1S) which is safe. In this case, it took LCegar 77 minutes and 21 seconds (CB) to verify correctness.

4.3 Conclusion

While the run times of our implementation are somewhat disappointing, we believe our implementation demonstrates the potential of language-based techniques in the verification of recursive multi-threaded programs.

Also, although the run times could probably be improved by providing better symbolic implementations of the language-theoretic operations used, considering more succinct encodings of programs is a must.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. Prentice Hall (1996)
2. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL 2002: Principles of Programming Languages, pp. 1–3. ACM (2002)
3. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. International Journal on Foundations of Computer Science 14(4), 551–582 (2003)
4. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying Concurrent Message-Passing C Programs with Recursive Calls. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
6. Clarke, E.M., Talupur, M., Veith, H.: Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008)
7. Eğecioğlu, Ö.: Strongly Regular Grammars and Regular Approximation of Context-Free Languages. In: Diekert, V., Nowotka, D. (eds.) DLT 2009. LNCS, vol. 5583, pp. 207–220. Springer, Heidelberg (2009)
8. Esparza, J., Ganty, P.: Complexity of pattern-based verification for multithreaded programs. In: POPL 2011: Principles of Programming Languages, pp. 499–510. ACM (2011)
9. Ganty, P., Majumdar, R., Monmege, B.: Bounded Underapproximations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 600–614. Springer, Heidelberg (2010)

10. Ginsburg, S., Spanier, E.H.: Bounded Algol-like languages. Transactions of the American Mathematical Society 113(2), 333–368 (1964)
11. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL 2011: Principles of Programming Languages, pp. 331–344. ACM (2011)
12. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: PLDI 2004: Programming Language Design and Implementation, pp. 1–13. ACM (2004)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-Modular Abstraction Refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002: Principles of Programming Languages, pp. 58–70. ACM (2002)
15. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
16. Kahlon, V.: Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In: LICS 2009: Logic in Computer Science, pp. 27–36. IEEE Computer Society (2009)
17. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: POPL 2003: Principles of Programming Languages, pp. 303–314. ACM (2007)
18. Kidd, N.: Bluetooth protocol, http://pages.cs.wisc.edu/~kidd/bluetooth/
19. Lal, A., Reps, T.: Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
20. Latteux, M., Leguy, J.: Une propriété de la famille GRE. In: FCT 1979, pp. 255–261. Akademie-Verlag (1979)
21. Mohri, M., Nederhof, M.-J.: Regular approximation of context-free grammars through transformation. In: Robustness in Language and Speech Technology, vol. 9, pp. 251–261. Kluwer Academic Publishers (2000)
22. Patin, G., Sighireanu, M., Touili, T.: SPADE: Verification of Multithreaded Dynamic and Recursive Programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 254–257. Springer, Heidelberg (2007)
23. Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
24. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI 2004: Programming Language Design and Implementation, pp. 14–24. ACM (2004)
25. Ben Rajeb, N., Nasraoui, B., Robbana, R., Touili, T.: Verifying multithreaded recursive programs with integer variables. Electr. Notes Theor. Comput. Sci. 239, 143–154 (2009)
26. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM TOPLAS 22(2), 416–430 (2000)
27. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic Context-Bounded Analysis of Multithreaded Java Programs. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 270–287. Springer, Heidelberg (2008)
28. van Leeuwen, J.: Effective constructions in well-partially-ordered free monoids. Discrete Mathematics 21(3), 237–252 (1978)
29. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. 3(12), 115–116 (1981)

Learning from Vacuously Satisfiable Scenario-Based Specifications*

Dalal Alrajeh[1], Jeff Kramer[1], Alessandra Russo[1], and Sebastian Uchitel[1,2]

[1] Department of Computing, Imperial College London, UK
[2] Departamento de Computaciòn, FCEyN, UBA

Abstract. Scenarios and use cases are popular means for supporting requirements elicitation and elaboration. They provide examples of how the system-to-be and its environment can interact. However, such descriptions, when large, are cumbersome to reason about, particularly when they include conditional features such as scenario triggers and use case preconditions. One problem is that they are susceptible to being satisfied vacuously: a system that does not exhibit a scenario's trigger or a use case's precondition, need not provide the behaviour described by the scenario or use case. Vacuously satisfiable scenarios often indicate that the specification is partial and provide an opportunity for further elicitation. They may also indicate conflicting boundary conditions. In this paper we propose a systematic, semi-automated approach for detecting vacuously satisfiable scenarios (using model checking) and computing the scenarios needed to avoid vacuity (using machine learning).

1 Introduction

Scenarios, use cases and story boards are popular means for supporting requirements engineering activities. They illustrate examples of how the software-to-be and its environment should and should not interact. They are commonly used as an intuitive, semi-formal language for describing behaviour at a functional level.

A common form for providing examples of behaviour is through conditional statements. Use cases [1] support existential conditional statements such as "once an appropriate user ID and passwords has been obtained, a homeowner *can* access the surveillance cameras placed throughout the house from any remote location via the internet" [21]. Live Sequence Charts [14] support universal conditional statements such as "the controller should probe the thermometer for a temperature value every 100 milliseconds, and if the result is more that 60 degrees, it *should* deactivate the heater and send a warning to the console". Some languages support both existential and universal conditional scenarios [24].

Conditional scenarios with different modalities are useful. They provide support for "what-if" elaboration of requirements specifications [1], and the progressive shift from existential statements, in the form of examples and use-cases, to universal statements in the form of declarative properties. Each conditional scenario

* We acknowledge financial support for this work from ERC project PBM - FIMBSE (No. 204853).

J, de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 377–393, 2012.

constitutes only a partial description of the system's intended behaviour. Hence, typically many of them are used in conjunction along with other behaviour descriptions such as system goals [10]. The emergent behaviour of such rich descriptions can be complex to reason about, hindering validation, and resulting frequently in specifications that are incomplete or contradictory.

One particular issue that conditional scenarios have is that they are liable to being satisfied vacuously; a system can be constructed so that it satisfies the conditional scenarios by never satisfying the condition. For instance, a system in which the homeowner is never given a user password vacuously satisfies the use case described above. This problem, commonly referred to as *antecedent failure* [8] in temporal specifications, is often an indication that the specification is partial and hence provides an opportunity for elicitation; it is clear that the stakeholder's intention is that "the system should provide the user with an id and password", and if it does, then the user can access the installed surveillance cameras. In addition, vacuously satisfiable specifications can have pernicious effects, concealing conflicting behaviour which is important to explore. For example, consider two scenarios extracted from the mine pump example in [16]: "once the methane sensors detect that the methane level is critical, *then* the pump controller must send a signal to the pump to be switched off" and "once the water sensors detect that the water level is above the high-threshold, *then* the pump controller must send a signal to the pump to be switched on". These scenarios are consistent as a system in which water sensors never detect high water and methane levels vacuously satisfies both scenarios. However, if these two levels were to occur, then the scenarios provide contradictory information of what the controller must do.

In this paper we describe an approach that not only detects vacuously satisfiable conditional scenarios but also provides automated support for learning new scenarios that ensure the conditions, i.e. triggers, are satisfied. More specifically, the approach takes as input a set of scenarios formalised as triggered existential and universal scenarios [24] and consists of two main phases. The first involves (*i*) synthesising a Modal Transition System from the scenarios, representing all possible implementations that satisfy them and (*ii*) performing a vacuity check, using a model checker, against a scenario's trigger. If the vacuity check is positive, the model checker produces examples of how the system-to-be could satisfy the trigger, i.e. non-vacuity witnesses [13]. In the second phase, (*iii*) an engineer classifies the examples as either positive or negative, i.e. ones that should be accepted or not in the final implementation, and then (*iv*), together with the given scenarios, inputs them into an inductive logic programming learning tool to compute new triggered scenarios which, if added to the existing scenarios, guarantee that they are no longer vacuously satisfiable. This process is repeated for each given triggered scenario, producing in the end a scenario-based specification that is not vacuously satisfiable. Figure 1 outlines the proposed framework.

Although the integrated use of model checking and ILP has been previously applied to other software engineering tasks, such as goal operationalisation [2] and zeno behaviour elimination [3], the current application introduces a

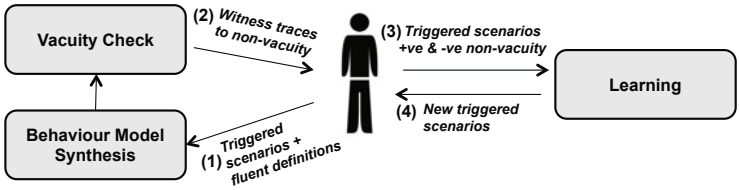


Fig. 1. Overview of the proposed framework

number of new technical challenges not present previously: the need to model and reason about partial behaviour, branching time and alphabet scoping of learned expressions. We elaborate further on these issues in Section 6.

The rest of this paper is organised as follows. We describe a motivating example in Section 2 and the necessary background in Section 3. Section 4 presents the main approach. Section 5 illustrates the results obtained by applying the approach to two case studies. We discuss related work in Section 6 and conclude in Section 7.

2 Motivating Example

Consider a simplified version of the mobile phone system described in [17]. The system is composed of six participants: a user, cover, display screen, speaker, chip and the environment. A phone user can open and close the phone cover, switch the phone on and off, answer and end calls and talk. The chip can detect incoming calls from the environment and the cover opening and closing. It can also initialise the phone settings and send requests to display the caller ID on the screen and to the speaker to start and stop ringing.

Suppose the engineer elicits the two scenarios shown in Figure 2 using a universal and existential triggered scenario notation, respectively. The universal scenario *Receive* informally states that "once an incoming call is detected (*incomeCall*), the phone rings (*startRing*) and the caller id is displayed on the screen (*displayCaller* and *setDisplay*) subsequently". The existential scenario *Phone* specifies the requirement "once an incoming call is detected (*incomeCall*) and the user opens the cover (*open* followed by *coverOpened*), the user may talk (*talk*)". Both scenarios are composed of two parts; a trigger (shown in a hexagon) and a triggered sequence (shown in a box; solid in universal and dashed in existential).

One problem with the specified scenarios is that although they describe what the system must or can do when the system exhibits the triggers, they do not state what it is required to do otherwise. For instance, they do not say when the system can exhibit an incoming call nor what the system can do between the occurrence of an incoming call and the user opening the phone cover. Because this specification is only partial, any implementation of the system in which an

incoming call is never allowed to occur is a valid implementation of the *Receive* scenario (See Figure 2.a). We refer to triggered scenarios which may result in a system that never exhibits the trigger as *vacuously satisfiable scenarios*.

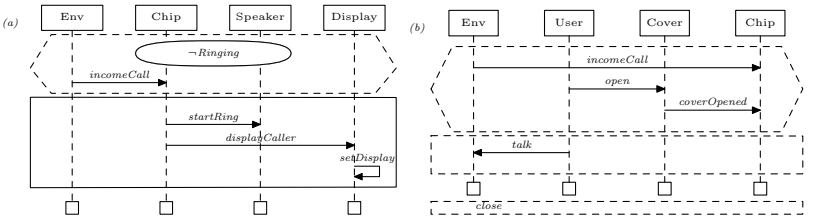


Fig. 2. Mobile phone system scenarios for (a) *Receive* and (b) *Phone*

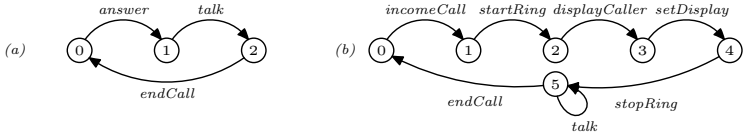


Fig. 3. Implementation that (a) vacuously satisfies *Receive* and *Phone* scenarios, and (b) satisfies the *Receive* scenario non-vacuously but *Phone* vacuously

Feedback about vacuously satisfiable scenarios may help engineers in recognising further behaviour which should be required or proscribed by any derived implementation. By informing an engineer about possible implementations in which an incoming call never occurs, the engineer could provide further examples of what the system behaviour may, must or cannot include. For instance, an engineer could provide a trace showing that incoming calls occurs after the phone is switched on and initialised, i.e. *switchOn*, *initialise*, *incomeCall*, or a negative trace where an incoming call occurs after the phone starts ringing, i.e. *startRing*, *incomeCall*. From such traces, it can be inferred that an incoming call may be triggered when the phone is initialised, or not ringing as shown in Figure 4.

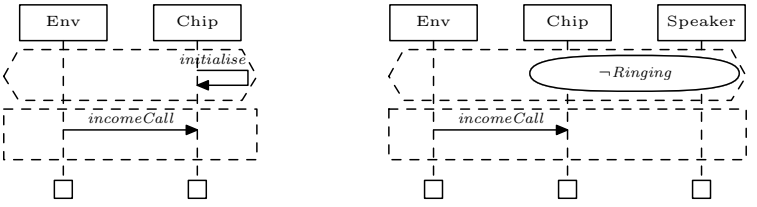


Fig. 4. New triggered scenarios to avoid vacuously satisfying the *Receive* scenario

In this paper, we show how model checking and inductive logic programming provide automated support for detecting vacuously satisfiable scenarios and the computing new scenarios that avoids vacuity, such as those shown in Figure 4.

3 Background

3.1 Triggered Scenarios

Triggered scenarios are sequence charts that represent interactions between the system's agents. Graphically, a triggered scenario comprises several vertical lines labelled by names representing agents' lifeline. Time is assumed to flow downward. Annotated arrows between these lines correspond to synchronous messages which represent instantaneous events on which both objects synchronise.

A triggered scenario consists of three parts; a trigger that is surrounded by a dashed hexagon, a main chart that is surrounded by a rectangular frame and a scope. The trigger is the condition that activates the main chart. It can include event messages as well as properties (depicted in rounded boxes). A property may be associated with one or more agent instances. It is a boolean combination of propositional atoms and their negations, expressed in Fluents Linear Temporal Logic (discussed later), that are expected to be true or false at that point in the system. A main chart can only contain messages. Event messages and properties are associated with ordered locations along the agents' lifelines. A universal Triggered Scenario (uTS) forces the occurrence of the main chart (depicted in a solid rectangular frame) after every occurrence of the trigger. An existential Triggered Scenario (eTS) asserts that it is possible to perform the main chart after every occurrence of the trigger but not necessarily, i.e. alternative behaviour after the trigger is allowed. The purpose of the scope is to restrict the occurrence of certain messages. Events appearing in a triggered scenario are by default within its scope. Further events can be included in the scope by adding them to the *restricts* set depicted in a dotted frame below the scenario's main chart. We refer to events in the scope as *observed* events. Any non-observed event can occur interleaved without restriction.

Triggered scenarios are interpreted over execution trees. An uTS (resp. eTS) is satisfied in an execution tree if at any node of the tree where the trigger is satisfied, *every* (resp. *at least one*) outgoing branch satisfies the main chart.

3.2 Fluent Linear Temporal Logic

Fluent Linear Temporal Logic (FLTL) is a linear temporal logic of fluents [12]. A fluent is a propositional atom defined by a set I_f of initiating events, a set T_f of terminating events and an initial truth value either *true* ($\mathbf{tt}$) or *false* ($\mathbf{ff}$). Given a set of event labels *Act*, we write $f = \langle I_f, T_f, Init \rangle$ as a shorthand for a fluent definition, where $I_f \subseteq Act$, $T_f \subseteq Act$, $I_f \cap T_f = \emptyset$ and $Init \in \{\mathbf{tt}, \mathbf{ff}\}$. We use $\dot{a}$ as a shorthand for a fluent defined as $\langle a, Act \backslash \{a\}, \mathbf{ff} \rangle$.

Returning to our running example, the fluents *Opened*, *Ringing* and *Calling*, meaning the cover is open, the phone is ringing and there is an incoming call, can be respectively defined in FLTL as follows.

```
Opened =<coverOpened, coverClosed, ff>
Ringing =<startRing, stopRing, ff>
Calling =<incomeCall, endCall, ff>
```

Given a set of fluents F, FLTL formulae are constructed using standard boolean connectives and temporal operators X (next), U (strong until), F (eventually) and G (always). The satisfaction of FLTL formulae is defined with respect to traces, i.e. sequences of events over a given alphabet *Act*. Given a trace $\sigma = a_1, a_2, ...$ over *Act* and fluent definitions D, a fluent is said to be true in σ at position i with respect to D if and only if,

- f is defined initially true and $\forall j \in \mathcal{N}.\ ((0 < j \leq i) \rightarrow a_j \notin T_f)$;
- $(\exists j \in \mathcal{N}.\ (j \leq i) \wedge (a_j \in I_f)) \wedge (\forall k \in \mathcal{N}.((j < k \leq i) \rightarrow a_k \notin T_f))$.

In other words, a fluent f holds if and only if it is initially true or an initiating event for f has occurred and no terminating event has occurred since.

3.3 Modal Transition Systems

A Modal Transition System (MTS) is used to formalise a partial model of the system's behaviour [19]. It extends Labelled Transition Systems (LTSs), a widely used formalism for describing and reasoning about system behaviour, by distinguishing between transitions that are required, proscribed and unknown, i.e. transitions for which it is not possible, based on current available knowledge, to guarantee that they will be admissible or prohibited.

Definition 1 (MTS and LTS). *A Modal Transition System is a tuple* $M = (Q, Act, \Delta^r, \Delta^p, q_0)$ *where* Q *is a finite set of states, Act is a set of event labels, called the alphabet,* $\Delta^r \subseteq Q \times Act \times Q$ *is a required transition relation and* $\Delta^p \subseteq Q \times Act \times Q$ *is a possible transition relation where* $\Delta^r \subseteq \Delta^p$ *and* q_0 *is the initial state. A transition that is possible but not required is called a maybe transition. An MTS where all possible transitions are required is called a Labelled Transition System, written* (Q, Act, Δ, q_0).

An MTS M is said to have a required transition on a, denoted $q \xrightarrow{a}_r q'$, if $(q, a, q') \in \Delta^r$. Similarly, M is said to have a maybe transition on a, denoted $q \xrightarrow{a}_m q'$, if $(q, a, q') \in \Delta^p - \Delta^r$. Figure 5 shows an example MTS for the mobile phone system, with the alphabet *Act*= {*open*, *close*, *incomeCall*, *coverOpened*, *coverClosed*, *setDisplay*, *displayCaller*, *startRing*, *answer*, *talk*}, where maybe transitions are denoted with a question mark following the label. Figure 3 shows two LTSs for the same system. Note that the numbered nodes are used for reference and do not designate a particular state.

A trace $\sigma = a_1, a_2, ...$, where $a_i \in Act$, is said to be required in an MTS M if there exists in M a sequence of states such that $q_0 \xrightarrow{a_1}_r q_1 \xrightarrow{a_2}_r q_2 ....$ It is said to be possible if there exists in M a sequence of states such that $q_0 \xrightarrow{a_1}_p q_1 \xrightarrow{a_2}_p q_2 ...$, with at least one transition relation that is in $\Delta^p - \Delta^r$.

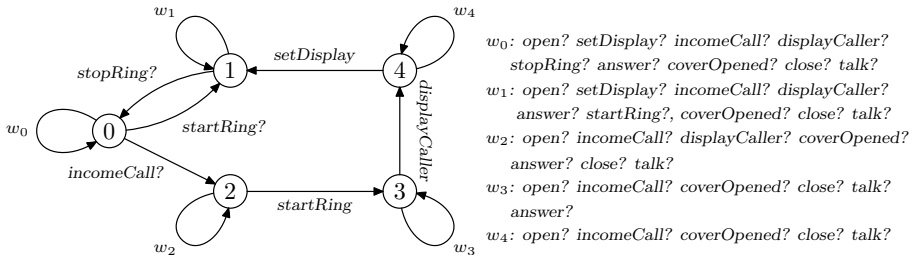


Fig. 5. An MTS synthesised from *Receive* scenario

Given two MTSs N and M, N is said to refine M if N preserves all of the required and proscribed transitions of M [19]. An LTS that refines an MTS M, i.e. an implementation, is a complete description of the system up to the alphabet of M. For example, the LTS shown in Figure 3.b is an implementation of the MTS given in Figure 5. *Merging* MTSs is the process of combining what is known from each MTS. In other words, it is the construction of a new MTS that includes all the required behaviour from each MTS but none of the prohibited ones. An MTS can be synthesised automatically from a safety property ϕ expressed in FLTL [25] and triggered scenarios *TS* [24] that characterises all implementations satisfying ϕ under a 3-valued interpretation on FLTL and *TS*, respectively.

4 Approach

As illustrated in Figure 1, the approach comprises two main phases. The first takes as input a set of fluent definitions and universal and existential triggered scenarios and uses model checking to verify if any of the existing triggered scenarios are vacuously satisfiable by some system implementations. If this is the case, the model checker provides non-vacuity witnesses. In the second phase, after an engineer classifies the non-vacuity witnesses into positive and negative examples, these are used to compute new triggered scenarios that ensure that the existing scenario is satisfied non-vacuously.

4.1 Checking Vacuity of Triggered Scenarios

We first define the term *vacuously satisfiable triggered scenario*, and then discuss the MTS construction, vacuity checks and witness generation.

Definition 2 (Vacuously Satisfiable Triggered Scenario). *Let S be a triggered scenario with trigger P, main chart C and scope Θ. Let M be an MTS that characterises all LTSs that satisfy S. The scenario S is said to be vacuously satisfiable in M, if there exists at least one LTS implementation I of M such that for all traces in I, restricted to the scope Θ, the trigger P is never satisfied.*

For instance, the triggered scenario *Receive*, shown in Figure 2.a, is vacuously satisfiable since there exists at least one implementation (e.g. the LTS in Figure 3.a) of the MTS synthesised from the scenario (shown in Figure 5) where the trigger is never satisfied.

The first step in detecting vacuity involves automatically synthesising an MTS that characterises all LTSs that satisfy the given set of triggered scenarios. The synthesis is done on a per triggered scenario basis, following the technique described in [24]. Once constructed, the generated MTSs are then merged. If the merge is successful, then the resulting MTS describes all implementations that satisfy all triggered scenarios. If it is unsuccessful then this indicates that the scenarios are inconsistent and hence do not have an implementation.

The second step comprises performing a vacuity check on the MTS resulting from the merge against a property that informally says: *it is always the case that a scenario's trigger does not hold*. The property can be expressed formally in FLTL and automatically constructed from the scenario's trigger. We refer to this property as the *negated trigger property* of a triggered scenario. For instance, the negated trigger property for the uTS *Receive* in Figure 2 is

$$\mathsf{G}\neg(\neg \mathit{Ringing} \wedge \dot{\mathit{incomeCall}}) \tag{1}$$

Model checking an MTS against a property is akin to checking the property against every LTS implementation that it describes. The result can be one of three values: all, none, and some, or more formally, true, false or undefined.

When checking for vacuity, if the result of model checking an MTS against a negated trigger property is true, then every trace in every implementation of the MTS satisfies the property, i.e. the trigger of the scenario under analysis never occurs. This entails that any implementation that satisfies the available specification vacuously satisfies the triggered scenario. This is an undesirable situation as it is not possible to extend the specification to avoid vacuity, and hence it must be revised. If the verification returns false, every implementation of the MTS has a trace that violates the property, i.e. in which the trigger occurs. Hence the triggered scenario is not vacuously satisfiable, so the specification need not be augmented for this particular scenario.

If the result of the verification is undefined, this means that there are some implementations that satisfy the concerned scenario vacuously and others that satisfy it non-vacuously. The purpose of the second phase of this approach is to automatically learn triggered scenarios that will prune out all implementations of the MTS that vacuously satisfy the concerned triggered scenario. However, for such learning to occur, examples of how the system-to-be may trigger the scenario under analysis are needed. In cases where the result is either false or undefined, a counterexample is given. In the former case, the counterexample is a trace that violates the property and can be exhibited by all LTS implementations. In the latter case, the counterexample is a trace that violates the property and can be exhibited by at least one LTS implementation. Our interest lies in

the latter case where the model checker provides an example of how some implementations can achieve the scenario's trigger. This trace, leading to the trigger, is taken as a non-vacuity witness for that triggered scenario.

Returning to our running example, verifying the MTS generated from the scenario *Receive* and *Phone* against the property (1) using the MTSA model checker [11] gives the following violation:

```
Trace to property violation in Never_Trigger_Receive:
     incomeCall       Calling
No. MobilePhone+ does not satisfy Never_Trigger_Receive
```

The trace produced by the MTSA is the shortest trace in an implementation of the MTS that violates the negated trigger property. In particular, the above means that there exists some implementations of the mobile phone system in which the phone is not ringing and there is an incoming call, i.e. where the trigger of scenario *Receive* is reachable (e.g. Figure 3.b).

Once the model checker detects a non-vacuity witness, this is shown to the engineer for validation. The engineer might indicate that the trace is *positive*, i.e. should be required in all implementations, or *negative*, i.e. should be proscribed in all implementation. In the former case, the trace is given to the learning phase. In the latter case, the engineer is expected to produce at least one positive non-vacuity witness which satisfies the trigger. Positive witnesses can be automatically generated from the model checker.

4.2 Learning Triggered Scenarios

The input to this phase is a set of triggered scenarios, fluent definitions and positive and negative non-vacuity witnesses. The output is a set of triggered scenarios, called a *required*, that ensure that a trigger is required by at least one positive witness trace in every implementation of the system.

Definition 3 (Required Scenarios). *Let TS be a set of triggered scenario, D a set of fluent definitions and $\Sigma^+ \cup \Sigma^-$ a set of positive and negative traces consistent with TS. Then a set of triggered scenarios S is said to be required of TS with respect to traces in $\Sigma^+ \cup \Sigma^-$ if the MTS synthesised from $TS \cup S$ requires each trace in Σ^+ but none of the traces in Σ^-.*

To compute new triggered scenarios, we use an Inductive Logic Programming (ILP) approach described in [22]. ILP is a machine learning technique for computing a new solution H that explains a given set E of examples with respect to a given (partial) background knowledge B [20]. Within the context of our problem, the background comprises the given set of triggered scenarios and fluent definitions whereas the positive and negative traces constitute the examples. A solution is a set of required scenarios requiring the positive non-vacuity witnesses, but none of the negative ones.

To perform the learning task, the input is encoded into Prolog. We have defined a sound translation (based on an extension of that given in [4]) that maps triggered scenarios and fluent definitions into an Event Calculus (EC) [15] program.

The program makes use of new predicates such as *required*, *maybe*, *reachable*, *trigger_satisfied*. The encoding of a triggered scenario TS results in a number of Prolog rules, one for each event appearing in the main chart of TS. Each rule defines the predicate *trigger_satisfied(e,T_m, S)*, where e is the event appearing in the main chart, T_m is the time variable associated with the location m at which the trigger is satisfied, and S is a trace variable[1]. The body of this rule contains $happens(e, T\_l, S)$ atoms for each event e at location $l < m$ in TS, and a conjunction of literals $(not)\ holds\_at(f_i, T\_l, S)$ for each fluent $(\neg)f_i$ that appears in a property $(\neg)f_1 \wedge \ldots \wedge (\neg)f_n$ at location $l < m$ in TS[2]. The order of the time variables in EC respects the location ordering in TS. Constraints over the type of triggering rule, i.e. existential or universal, are also defined according to the semantics in [24]. The scope of any scenario to be learned is constrained to be a subset of the events appearing in the positive non-vacuity witness. The main charts are encoded to ensure the soundness of the resulting program and consistency of learned hypotheses.

The solution search space is governed by a language bias which defines the syntactic structure of plausible solutions. To learn triggered scenarios, we define a language bias to capture rules with triggers as conditions and triggered events as its consequents. The language bias is also set to compute sequences of events leading to the main chart of the given scenarios so that all computed scenarios are consistent with the existing ones. For every positive example, the learning tries to construct a solution H which explains why a certain sequence of events must be required either existentially or universally, within a given scope. It then performs a generalisation step in which it tries to weaken the conditions to cover required occurrences of the triggered sequence in other traces. This generalisation can be controlled by providing several positive and negative non-vacuity witnesses.

The learning succeeds in computing a solution if there is at least one event occurrence in a positive non-vacuity witness that is not required by an existing triggered scenario. If several possible required scenarios exists, these will be given as output and it is the engineer's task to select the appropriate ones from those available. The number of triggered scenarios learned can be influenced by a number of factors including the number of events in the scope of the scenario to be learned, their occurrences in the example traces and the number of given negative traces. All produced scenarios are guaranteed to be consistent with the existing specification and the traces provided, as stated in Theorem 1 below.

Theorem 1. *[Soundness of Learning] Let TS be a set of triggered scenarios, D a set of fluent definitions and $\Sigma^+ \cup \Sigma^-$ a set of positive and negative traces consistent with TS. Let $\Pi = B \cup E$ be the EC encoding of TS, D and $\Sigma^+ \cup \Sigma^-$ into background knowledge B and examples E. If H is a solution to E with respect to B, then the set of learned triggered scenarios T, where T is the triggered scenarios corresponding to H, are required of TS w.r.t. traces in $\Sigma^+ \cup \Sigma^-$.*

[1] In Prolog, variables (resp. constants) start with a capital (resp. lowercase) letter.

[2] The notation $(\neg)\phi$ is a shorthand for ϕ or $\neg\phi$. A similar interpretation is used for $(not)\phi$.

The proof is by contradiction. In brief, it assumes that a trace $\sigma+ = e_1, \ldots, e_n$ is not required in the MTS synthesised from $TS \cup T$. Then it goes to show that this results in a program Π that does not contain any rule that requires the occurrence of some event e_i in the trace σ^+. Given that this leads to a contradiction as H is a set of rules that requires the occurrence of each event in σ^+, then $\sigma+ = e_1, \ldots, e_n$ is shown to be a required trace. The proof for σ^- is done in a similar fashion. As a corollary of the above theorem, when the traces are examples of positive and negative non-vacuity traces to triggers in TS, the learned triggered scenarios will guarantee that each positive non-vacuity trace is required in every implementation but none of the negative non-vacuity traces.

The choice of which learned scenarios to include may have an impact on later iterations. For instance, selecting a universal scenario over an existential one might imply that an incoming call is the only observed event when there is no incoming call detected and the phone is not ringing, for a given scope. It is obvious that selecting such an interpretation would prevent the occurrence of any behaviour other than that which is depicted in the main chart of the learned scenario within that scope. Therefore, we found that it is often preferable to select existential scenarios over universal at early stages of the elaboration process.

In our running example, the ILP tool computed two alternative required extensions as solutions; an existential and a universal. The learned existential triggered scenario (shown in Figure 6) states that whenever the user is not engaged in a call and the phone is not ringing then it is possible to accept an incoming call. The scope is restricted to the event appearing in the scenario. The universal contained the same trigger, main chart and scope.

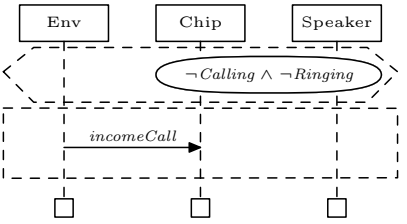


Fig. 6. A learned existential triggered scenario *IncomingCalls*

Once the engineer has made a selection, the learned scenario is added to the initial set. Then the newly synthesised MTS (which is a refinement of the original) is verified against the negated trigger property of another triggered scenario. If the model checker returns false for all negated trigger properties of the available scenario then this marks the end of the elaboration task with respect to the concerned trigger. If however, the model checkers returns undefined, then the process is repeated again with respect to the new non-vacuity witness trace.

Note that the encoding of the specification and the computation are hidden from the engineer. In fact, the engineer only needs to provide the learning system with the triggered scenarios, witnesses and fluent definitions and it will automatically propose a set of required scenarios with respect to the witnesses.

5 Case Studies

We report on the results obtained from two case studies, the Philips television set configuration from [23] and the air traffic control system in [9]. These were chosen because they have been used as case studies in much of the literature for which an elaborated scenario specification exists.

For the Philips configuration set, the specification contained existential triggered scenario from [23]. For the air traffic control system, it included a set of universal live sequence charts from [9]. All available scenarios were produced by third parties. We extracted a subset of the scenarios that constituted the main behaviour requirements provided, i.e. sunny day, normal behaviour. The aim of the case studies was (i) to investigate the capability of the approach in identifying the partiality of the given specification (ii) to verify that the learned triggered scenarios resulted in implementations that non-vacuously satisfied the given scenarios and finally (iii) to ensure that the learned scenarios were relevant to the domain at hand. The latter was achieved by comparing the learned scenarios with the available specification.

5.1 Philips Television Set Configuration

This case study is on a protocol used in a product family of Philips television sets. It include multiple tuners and video output devices that can be configured by a user. The protocol is concerned with controlling the signal path to avoid visual artefacts appearing on video outputs when a tuner is changing frequency.

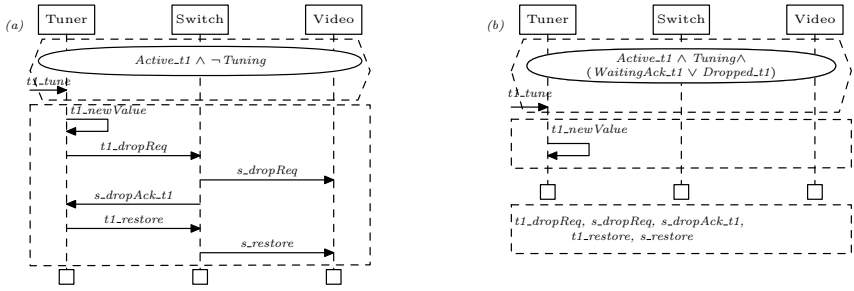


Fig. 7. Triggered scenarios: (a) *Tuning_t1_Active_t1* (b) *NestedTuning_t1_Active_t1*

We discuss here the results obtained by applying our approach to two existential triggered scenarios; *Tuning_t1_Active_t1* and *NestedTuning_t1_Active_t1* shown in Figure 7. The fluents appearing in the triggers are defined as follows.

```
Active_t1 =<set_Active_t1, set_Active_t2, tt>
Tuning_t1 =<t1_tune, {s_restore, set_Active_t1, set_Active_t2}, ff>
WaitingAck_t1 =<t1_dropReq, s_dropAck_t1, ff>
Dropped_t1=<s_dropAck_t1, t1_restore, ff>
```

A vacuity check was performed for the scenario *Tuning_t1_Active_t1* first by checking the system MTS resulting from the merge of the scenarios' MTSs against the following negated trigger property.

$$\mathsf{G}\neg((Active\_t1 \wedge \neg Tuning\_t1) \ \wedge \ (\ \mathsf{X}\ t1\dot{\_}tune)) \tag{2}$$

The model checker produced the shortest non-vacuity witness, i.e. *t1_tune*. Based on the description given in [23], we provided the system with a negative non-vacuity trace where a *t1_tune* events occurs when tuner 2 is active instead. From these traces, the learning produced two plausible triggered scenarios for the event *t1_tune*, one existential and one universal, requiring *t1_tune* event to happen when tuner 1 is active and not tuning. Choosing the universal scenario implies that a *t1_tune* must be observed every time the trigger is satisfied. We selected an existential interpretation to allow exploration of other behaviour (see Figure 8.a). The approach was also used to check for the vacuous satisfiability of *NestedTuning_t1_Active_t1* (Figure 7.b). Our application resulted in a single existential triggered scenario shown in Figure 8.b. The learned scenarios were added to the initial specification. Verifying the new MTS against the negated trigger properties showed that both triggered scenarios *Tuning_t1_Active_t1* and *NestedTuning_t1_Active_t1* were non-vacuously satisfied in all implementations.

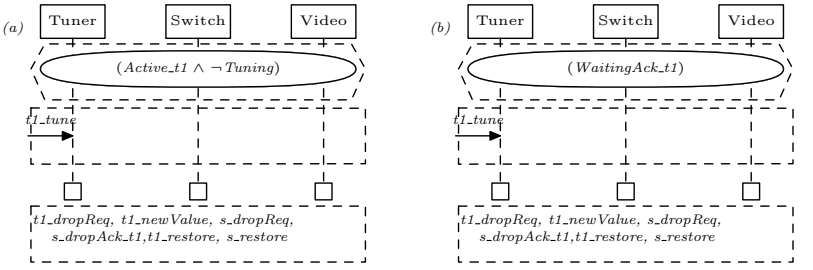


Fig. 8. Learned existential triggered scenarios (a) *TuneAllowed_t1_Active_NotTuning* (b) *TuneAllowed_WaitingAck_t1*

During the analysis phase, the approach also helped in detecting negative non-vacuity witnesses. For example, the analysis showed that the specification permitted behaviour in which the tuner sent a request to drop the signal without the user requesting a tune signal, and another in which a nested tune occurred outside the 'storing regions', i.e. when *Waiting_Ack_t1* and *Dropped_t1* are both false. With the identification of positive non-vacuity traces the learning ensured

that the learned scenarios did not require the occurrence of such events under such conditions. The final set of scenarios produced using our proposed method were validated against those generated by running the existing protocol in [23].

5.2 Air Traffic Control System

The Center-TRACON Automation System (CTAS) is a system for controlling and managing air traffic flow at major terminal areas to reduce travel delays and improve safety. The communication between the CTAS components is managed by the Communication Manager (CM) component which stores all interactions in a database and sends any required information to the requesting components. Among the CTAS requirements is that every client using weather data should be notified of any weather update. The scenarios in Figure 9 are universal triggered scenarios reproduced from [9] regarding the successful and failed update of new weather information.

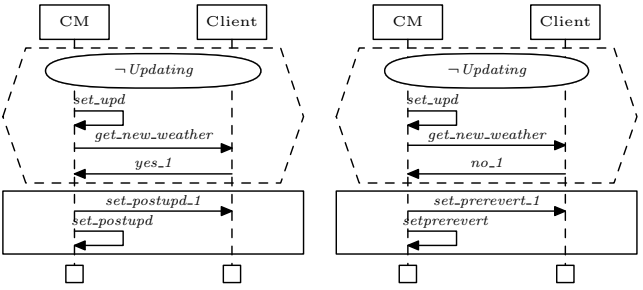


Fig. 9. Successful and failed update universal triggered scenarios

An application of our approach to this problem resulted in a total of six universal triggered scenarios and a single existential one. The set of uTSs computed were in fact the same triggered scenarios given in [9]. An excerpt is shown in Figure 10. Our approach also computed the existential scenario depicted in Figure 10 for setting the weather cycle status to "pre-updating" which was not present in the specification but is necessary to start the update process.

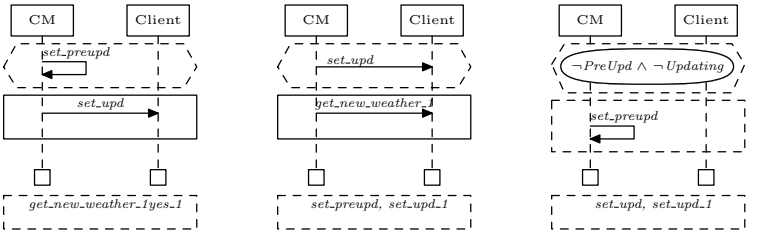


Fig. 10. CTAS learned universal and existential triggered scenarios

6 Discussion and Related Work

Although this paper discusses learning from vacuously satisfiable scenarios, the approach can be generalised for other forms of conditional scenarios (e.g. LSCs) and conditional statements (e.g. goals and requirements [10]). The exact definition of the learning task could be customised to the specific problem at hand.

There has been much research on providing automated support for elaborating scenario-based specifications [1,26]. However, much of the existing work is either informal, deals with message sequence charts or does not address the problem of vacuity introduced by conditional scenarios.

To the best of our knowledge, there is no prior work on applying learning algorithms to compute triggered scenarios. However, using model checking to detect vacuously satisfiable specifications has been the subject of several research efforts e.g. [7,18,13]. The work in [13] for instance presents a technique for detecting vacuity in temporal properties expressed in $\mathcal{X}$CTL. They use a multi-valued model checking algorithm to determine which subformulas in a given expression are vacuously satisfied in a model. Our approach is similar in that we use model checking algorithms to detect non-vacuity, and to produce a non-vacuity witness. However, in addition to the type of specifications used, our work differs in that it computes possible ways to avoid non-vacuity.

In our previous work, we combined the use of model checking and ILP to provide automated support for *different* software engineering tasks. In [2], a complete set of operational requirements in the form of preconditions and trigger conditions are iteratively learned from goal models. In [3], model checking and ILP are used to infer the missing conditions required to guarantee that a discrete time goal-based specification only admits non-zeno behaviours. Although we use here the same techniques, i.e. model checking and ILP, as in [2,3] to solve different software engineering tasks, their application to the problem of detecting vacuity and learning new triggered scenarios has posed three new main challenges: partial behaviour models, branching time and scoping of learned expressions. These points are elaborated below.

The problem addressed in this paper requires the ability to reason about universal and existential statements (both [2] and [3] deals only with universal statements). This means that traditional 2-valued semantic domains for these specifications are inadequate and a partial behaviour formalism such as MTS is required. As a consequence, the logic programming language is extended with new predicates (e.g. *required* and *maybe*). In addition, the use of triggered existential scenarios introduces statements that have a branching time semantics (in both [2,3] learning is only defined over properties with linear time). For this, the logic programming language has been extended to formalise hypothetical paths that branch from particular positions in a trace. The scenario language used in this paper supports scoping each scenario with an alphabet. (both [2,3] consider statements to have the same scope). This entails that the learning procedure must not only consider the scope of given scenarios (i.e. axioms for ensuring that the satisfiability notion with respect to a given scope are required) but also learned scenarios must include the scope for which they are intended. In [4] we

have presented preliminary work on the application of ILP in the context of MTS models. The focus there is on learning safety properties to requires some possible transitions from given traces. In this paper we build on the formalisation of MTSs in the logic programs and extend it to represent statements with a branching semantics and scoping which are not considered in [4]. Finally, note the work in [6] addresses the problem of learning operational requirements as in [2,5] but without the use of model checking.

7 Conclusion and Future Work

This paper presents a novel tool-supported approach for the elaboration of partial, conditional scenario-based specifications. In particular, we show how model checking can be used for identifying vacuously satisfiable triggered scenarios and how inductive logic programming can support the computation of new triggered scenarios needed to avoid such vacuity.

As part of this work and future work, we intend to investigate alternative methods for learning scopes of triggered scenarios. We also aim to extend the approach to resolve inconsistencies in the specification by providing support for detecting which parts of the specifications are the cause of inconsistency (building upon results in [13]) and learning possible revisions to the triggered scenarios necessary to resolve inconsistencies.

References

1. Alexander, I., Maiden, N.: Scenarios, stories, use cases: through the systems development life-cycle. Wiley (2004)
2. Alrajeh, D., Kramer, J., Russo, A., Uchitel, S.: Learning operational requirements from goal models. In: Proc. of 31st ICSE, pp. 265–275 (2009)
3. Alrajeh, D., Kramer, J., Russo, A., Uchitel, S.: Deriving non-zeno behaviour models from goal models using ILP. J. of FAC 22(3-4), 217–241 (2010)
4. Alrajeh, D., Kramer, J., Russo, A., Uchitel, S.: An inductive approach for modal transition system refinement. In: Tech. Comm. of 27th ICLP, pp. 106–116 (2011)
5. Alrajeh, D., Ray, O., Russo, A., Uchitel, S.: Extracting requirements from scenarios with ILP. In: Proc. of 16th Intl. Conf. on ILP, pp. 63–77 (2006)
6. Alrajeh, D., Ray, O., Russo, A., Uchitel, S.: Using abduction and induction for operational requirements elaboration. J. of Applied Log. 7(3), 275–288 (2009)
7. Armoni, R., Fix, L., Flaisher, A., Grumberg, O., Piterman, N., Tiemeyer, A., Vardi, M.Y.: Enhanced Vacuity Detection in Linear Temporal Logic. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 368–380. Springer, Heidelberg (2003)
8. Beatty, D.L., Bryant, R.E.: Formally verifying a microprocessor using a simulation methodology. In: Proc. of 31st DAC, pp. 596–602 (1994)
9. Bontemps, Y.: Relating Inter-Agent and Intra-Agent Specifications: The Case of Live Sequence Charts. PhD thesis, Faculties Universitaires Notre-Dame de la Paix, Namur Institut dInformatique, Belgium (2005)
10. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Comp. Program. 20(1), 3–50 (1993)

11. D'Ippolito, N., Fischbein, D., Chechik, M., Uchitel, S.: MTSA: The Modal Transition System Analyser. In: Proc. of 23rd Intl. Conf. on ASE, pp. 475–476 (2008)
12. Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems. In: Proc. 11th ACM SIGSOFT FSE, pp. 257–266 (2003)
13. Gurfinkel, A., Chechik, M.: Extending Extended Vacuity. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 306–321. Springer, Heidelberg (2004)
14. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine. Springer-Verlag New York, Inc. (2003)
15. Kowalski, R.A., Sergot, M.: A logic-based calculus of events. New Generation Comp. 4(1), 67–95 (1986)
16. Kramer, J., Magee, J., Sloman, M.: Conic: An integrated approach to distributed computer control systems. In: IEE Proc., Part E 130 (1983)
17. Kugler, H.-J., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal Logic for Scenario-Based Specifications. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 445–460. Springer, Heidelberg (2005)
18. Kupferman, O.: Sanity checks in formal verification. In: Conc. Theory, pp. 37–51 (2006)
19. Larsen, K.G., Thomsen, B.: A modal process logic. In: Proc. of 3rd Annual Symp. on Log. in Comp. Science, pp. 203 –210 (1988)
20. Muggleton, S.H.: Inverse Entailment and Progol. New Generation Comp., Special Issue on ILP 13(3-4), 245–286 (1995)
21. Pressman, R.S.: Software Engineering: A Practitioner's Approach, 7th edn. McGraw-Hill Higher Education (2010)
22. Ray, O.: Nonmonotonic abductive inductive learning. J. of Applied Log. 7(3), 329–340 (2009)
23. Sibay, G.: The Philips television set case study, http://sourceforge.net/projects/mtsa/files/mtsa/CaseStudies/
24. Sibay, G., Uchitel, S., Braberman, V.: Existential live sequence charts revisited. In: Proc. of 30th ICSE, pp. 41–50 (2008)
25. Uchitel, S., Brunet, G., Chechik, M.: Behaviour model synthesis from properties and scenarios. In: Proc. of 29th Intl. Conf. on Softw. Eng., pp. 34–43 (2007)
26. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proc. of the 22nd ICSE, pp. 314–323 (2000)

Explanations for Regular Expressions

Martin Erwig and Rahul Gopinath

School of EECS,
Oregon State University

Abstract. Regular expressions are widely used, but they are inherently hard to understand and (re)use, which is primarily due to the lack of abstraction mechanisms that causes regular expressions to grow large very quickly. The problems with understandability and usability are further compounded by the viscosity, redundancy, and terseness of the notation. As a consequence, many different regular expressions for the same problem are floating around, many of them erroneous, making it quite difficult to find and use the right regular expression for a particular problem. Due to the ubiquitous use of regular expressions, the lack of understandability and usability becomes a serious software engineering problem.

In this paper we present a range of independent, complementary representations that can serve as explanations of regular expressions. We provide methods to compute those representations, and we describe how these methods and the constructed explanations can be employed in a variety of usage scenarios. In addition to aiding understanding, some of the representations can also help identify faults in regular expressions. Our evaluation shows that our methods are widely applicable and can thus have a significant impact in improving the practice of software engineering.

1 Introduction

Regular expressions offer a limited but powerful metalanguage to describe all kinds of formats, protocols, and other small textual languages. Regular expressions arose in the context of formal language theory, and a primary use has been as part of scanners in compilers. However, nowadays their applications extend far beyond those areas. For example, regular expressions have been used in editors for structured text modification [19]. They are used in network protocol analysis [24] and for specifying events in a distributed system [18]. Regular expressions are used for virus detection using signature scanning [15], in mining the web [14], and as alternatives types for XML data [13]. Moreover, there are many uses of regular expressions outside of computer science, for example, in sociology (for characterizing events that led to placing of a child in foster care) [21] or biology (for finding DNA sequences) [16]. In addition to specific applications, many generic types of information, such as phone numbers or dates, are often presented and exchanged in specific formats that are described using regular expressions.

Despite their popularity and simplicity of syntax, regular expressions are not without problems. There are three major problems with regular expressions that can make their use quite difficult.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 394–408, 2012.

- *Complexity*. Regular expressions are often hard to understand because of their terse syntax and their sheer size.
- *Errors*. Many regular expressions in repositories and on the web contain faults. Moreover, these faults are often quite subtle and hard to detect.
- *Version Proliferation*. Because many different versions of regular expressions are stored in repositories for one particular purpose, it is in practice quite difficult to find or select the right one for a specific task.

A major source of these problems is the lack of abstraction in regular expressions, leaving the users no mechanism for reuse of repeated subexpressions. By obscuring the meaning, this also contributes to the explosion of regular expressions that are just variations of others, often with errors.

It is easy to see that lack of abstraction causes regular expressions that do anything non-trivial to grow quite large. We have analyzed all the 2799 regular expressions in the online regular expression repository at `regexplib.com` and found that 800 were at least 100 characters long. Some complex expressions actually exceeded 4000 characters, and the most complex expressions had more than ten nested levels. It is very hard for users to look through what essentially are long sequences of punctuation and understand what such an expression does. That this is a real problem for programmers can be seen, for example, from the fact that the popular website `stackoverflow.com` has over 2000 questions related to regular expressions. (Of those, 50 were asking for variations of dates and 100 questions are about how to validate emails with regular expressions.)

As an example consider the following regular expression. It is far from obvious which strings it is supposed to match. It is even more difficult to tell whether it does so correctly.

```
<\s*[aA]\s+[hH][rR][eE][fF]=f\s*>\s* <\s*[iI][mM][gG]\s+[sS][rR][cC]
=f\s*>[^<>]*<\s*/[iI][mM][gG]\s*>\s*<\s*/[aA]\s*>
```

This complexity makes it very hard to understand non-trivial regular expressions and almost impossible to verify them. Moreover, even simple modifications become extremely prone to errors.

To address these problems and offer help to users of regular expressions, we have developed an ensemble of methods to explain regular expressions. These methods reveal the structure of regular expressions, identify the differences between represented data and format, and can provide a semantics-based annotation of intent.[1] In addition to aiding users in their understanding, our explanations can also be employed to search repositories more effectively and to identify bugs in regular expressions.

The rest of the paper is structured as follows. In Section 2 we discuss shortcomings of regular expressions to reveal the opportunities for an explanation notation. Specifically, we illustrate the problems with complexity and understandability mentioned above. Based on this analysis we then develop in Section 3 a set of simple, but effective explanation structures for regular expressions and demonstrate how these can be computed. We present an evaluation of our work in Section 4 with examples taken from a public repository. Finally, we discuss related work in Section 5 and present conclusions in Section 6.

[1] A first explanation of the above example is given in Figure 1, and a more comprehensive explanation follows later in Figure 2.

2 Deficiencies of Regular Expressions

There are many different forms of regular expressions and many extensions to the basic regular expressions as defined in [12]. Perl regular expressions provide the largest extension set to the above. However, the extensions offered by Perl go far beyond regular languages and are very hard to reason about. Therefore, we have chosen the POSIX regular expression extensions [20] as the basis of our work along with short forms for character classes, which provide a good coverage of the commonly used regular expression syntax while still avoiding the complexity of Perl regular expression syntax.[2]

Some principal weaknesses of the regular expression notation have been described by Blackwell in [1]. Using the cognitive dimensions framework [2] he identifies, for example, ill-chosen symbols and terse notation as reasons for the low understandability of regular expressions. Criticism from a practical perspective comes, for example, from Jamie Zawinsky, co-founder of Netscape and a well-known, experienced programmer. He is attributed with the quote "Some people, when confronted with a problem, think: 'I know, I'll use regular expressions.' Now they have two problems." [11].

In the following we point out some of the major shortcomings of regular expressions. This will provide input for what explanations of regular expression should accomplish.

(1) Lack of Abstraction. A major deficiency of regular expressions is their lack of an abstraction mechanism, which causes, in particular, the following problems.

- *Scalability*. Regular expressions grow quite large very quickly. The sheer size of some expressions make them almost incomprehensible. The lack of a naming mechanism forces users to employ copy-paste to represent the same subexpression at different places, which impacts the scalability of regular expressions severely.
- *Lack of structure.* Even verbatim repetitions of subexpressions cannot be factored out through naming, but have to be copied. Such repetitions are very hard to detect, but knowledge about such shared structure is an important part of the meaning of a regular expression. Therefore, regular expressions fail to inform users on a high level about what their commonalities and variabilities are.
- *Redundancy*. The repetition of subexpression does not only obscure the structure and increase the size, it also creates redundancies in the representations, which can lead to update anomalies when changing regular expressions. This is the source of many faults in regular expressions and has a huge impact on their maintainability.
- *Unclear intent*. Both of the previously mentioned problems make it very hard to infer the intended purpose of a regular expression from its raw syntax. Without such knowledge it is impossible to judge a regular expression for correctness, which also makes it hard to decide whether or not to use a particular regular expression. Moreover, the difficulty of grasping a regular expression's intent makes it extremely hard to select among a variety of regular expressions in a repository and find the right one for a particular purpose.

All these problems directly contribute to a lack of understandability of regular expressions and thus underline the need for explanations. The problem of unclear intent also points to another major shortcoming of regular expressions.

[2] For simplicity we do not consider Unicode.

(2) Inability to Exploit Domain Knowledge. Abstract conceptual domains are structured through metaphorical mappings from domains grounded directly in experience [3]. For example, the abstract domain of time gets its relational structure from the more concrete domain of space. Children learning arithmetic for the first time commonly rely on mapping the abstract domain of numbers to their digits [7]. The abstract concepts are easier to pick up if users are provided with a mapping to a less abstract or more familiar domain. One of the difficulties with regular expressions is that it is a formal notation without a close mapping [2] to the domain that the user wants to work with, which makes it difficult for new users to pick up the notation [1]. Moreover, there is no clear mental model for the behavior of the expression evaluator.

(3) Role Expressiveness. A role expressive notational system must provide distinctive visual cues to its users as to the function of its components [2]. Plain regular expressions have very few beacons to help the user identify the portions of regular expressions that match the relevant sections of input string. The users typically work around this by using subexpressions that correspond to major sections in the input, and by using indentation to indicate roles of subexpressions visually.

(4) Error Proneness. Regular expressions are extremely prone to errors due to the fact that there is no clear demarcation between the portions of regular expression that are shared between the alternatives, and those portions that are not, and thus have a higher contribution towards the variations that can be matched.

In the next section we will address the problems of scalability, unclear structure, and redundancy through explanation representations that highlight their compositional structure and can identify common formats.

3 Explanation Representations and Their Computation

The design of our explanation representations is based on the premise that in order to understand any particular regular expression it is important to identify its structure and the purpose of its components. Moreover, we exploit the structures of the domain that a particular regular expression covers by providing semantics-based explanations that can carry many syntactic constraints in a succinct, high-level manner. Two other explanation structures that are obtained through a generalization process are briefly described in the Appendix.

3.1 Structural Analysis and Decomposition

Large regular expressions are composed of smaller subexpressions, which are often very similar to each other. By automatically identifying and abstracting common subexpressions, we can create a representation that directly reveals commonalities. Our method is a variation of a rather standard algorithm for factoring common subexpressions. We illustrate it with the example given in the introduction. We begin by grouping maximally contiguous sequences of characters that do not contain bracketed expressions such as

`(···)`, `[···]` or `|` into bracketed expressions. This step is not necessary if the expression does not contain `|`. It preserves the meaning of the regular expression because brackets by themselves do not change the meaning of a regular expression.

For example, here are the first several subsequences identified in that way for the introductory example (each separated by white space).

```
<\s* [aA] \s+ [hH] [rR] [eE] [fF] =f\s*>\s*<\s* [iI] [mM] [gG]
\s+ [sS] [rR] [cC] =f\s*> [^<>]* ...
```

This step essentially amounts to a tokenization of the regular expression character stream.

The next step is to introduce names for sequences (not containing space specifications) that are recognized as common entities that appear more than once in the expression.

In the extraction of subexpressions we make use of a number of heuristics for generating names for the subexpressions since the choice of names has a big influence on the understandability of the decomposed expression [4, 8]. For example, a plain name represents the regular expression that matches the name, such as *img* = `img`. Since many applications of regular expressions involve the description of keywords whose capitalization does not matter, we also use the convention that underlined names represent regular expressions that match any capitalization of that name, such as $\underline{a}$ = `[aA]` or $\underline{img}$ = `[iI][mM][gG]`.

In the example, common expressions are expressions that represent upper- or lowercase characters, such as `[aA]`, which will therefore be replaced by their names. After this replacement, the token sequence for our example looks as follows.

```
<\s* a \s+ h r e f =f\s*>\s*<\s* i m g \s+ s r c =f\s*> [^<>]* ...
```

This grouping process can now be iterated until no more groups can be identified for naming. In the example we identify several new names as follows.

```
<\s* a \s+ href =f\s*>\s*<\s* img \s+ src =f\s*> [^<>]* ...
```

The advantage of the described naming conventions is that they provide an implicit naming mechanism that simplifies regular expressions without introducing any overhead.

Another naming strategy is to generate names from equivalent POSIX character class names, such as *alpha* = `[a-zA-Z]`, *alphas* = *alpha*`*`, *digit* = $[0-9]$, and *digits* = *digit*`*`. In the case of sequence of one or more matches, we capitalize the word, that is, *Alphas* = *alpha*`+` and *Digits* = *digit*`+`. We also replace sequences that are identified as matching a continuous sequence of numbers with a range notation. For example, `[0 .. 12]` = `[0-9]|1[0-2]` and `[00 .. 99]` = `[0-9][0-9]`.

Finally, we also use the convention that a blank matches `\s*`, that is, a sequence of zero or more spaces, and that a boxed space ␣ matches `\s+`, that is, a sequence of one or more spaces. Redundant brackets that do not enclose more than one group are also removed in this step. Applying some of these simplifications, we finally obtain for our example expression the decomposition shown in Figure 1.

```
< a␣href=f > < img␣src=f >[^<>]*< /img > < /a >
```

Fig. 1. Decomposition for the embedded image regular expression

This representation makes it much clearer what the original regular expression does, namely matching an image tag embedded in a link that matches the same file name `f`. This is a common pattern frequently used in programs and scripts that scan web pages for pictures and download the image files.

Our decomposition method can also identify parameterized names for subexpressions that are variations of one another. Although, this technique can further reduce redundancy, it also makes explanations more complex since it requires understanding of parameterized names (which are basically functions) and their instantiation (or application). Since it is not clear at this point whether the gained abstraction warrants the costs in terms of complexity, we ignore this feature in this paper.

Although the overall structure of regular expressions is revealed through decomposition, the approach described so far is only partially satisfying. In particular, the different functions of format and data expressions are not yet distinguished, and opportunities for semantic grouping have not been exploited yet. We will address these two issues next.

3.2 Format Analysis

Most non-trivial regular expressions use punctuation and other symbols as separators for different parts of the expression. For example, dates and phone numbers are often presented using dashes, or parts of URLs are given by names separated by periods. We call such a fixed (sub)sequence of characters that is part of any string described by a regular expression a *format*. Since all strings contain the format, it does not add anything to the represented information. Correspondingly, we call the variable part of a regular expression its *data part*.

By definition the individual strings of a format must be maximal, that is, in the original regular expression there will always be an occurrence of a (potentially empty) data string between two format strings. The situation at the beginning and end of a format is flexible, that is, a format might or might not start or end the regular expression.

A format can thus be represented as an alternating sequence of format strings interspersed with a wildcard symbol "•" that stands for data strings. For example, possible formats for dates are `•-•-•` and `•/•/•`, and the format for our embedded image tag expression is the pattern `<•=f><•=f>•</•></•>`. The wildcard symbol matches any number of characters so long as they are not part of the format. This notion of a format can be slightly generalized to accommodate alternative punctuation styles, as in the case for dates. Thus a regular expression can have, in general, a set of alternative formats. Note that because there is no overlap between the format and data strings, identifying the format also helps with the structuring of the data.

The computation of formats can be described using a simple recursive definition of a function *format*, which computes for a given regular expression e a set of formats. The function is defined by case analysis of the structure of e. A single constant c is itself

a format, and the format for a concatenation of two expressions $e \cdot e'$ is obtained by concatenating the formats from both expressions (successive $\bullet$ symbols are merged into one). We cannot derive a format in the above defined sense from a repetition expression since it cannot yield, even in principle, a constant string.[3] The formats of an alternation $e|e'$ are obtained by merging the formats of both alternatives, which, if they are "similar enough", preserves both formats, or results in no identification of a format (that is, $\bullet$) otherwise.

$$\begin{aligned} format(c) &= \{c\} \\ format(e \cdot e') &= \{f \cdot f' \mid f \in format(e) \wedge f' \in format(e')\} \\ format(e|e') &= \cup_{f \in format(e), f' \in format(e')} align(f, f') \\ format(e^*) &= \bullet \end{aligned}$$

Two formats are *similar* when they can be aligned in a way so that the positions of the wild-cards and the fixed strings align in both expressions. If this is the case, the function *align* will return both formats as result, otherwise it will return a wildcard. For example, the formats $f = \bullet\text{-}\bullet\text{-}\bullet$ and $f' = \bullet/\bullet/\bullet$ are similar in this sense, and we have $align(f, f') = \{\bullet/\bullet/\bullet, \bullet\text{-}\bullet\text{-}\bullet\}$.

3.3 User-Directed Intent Analysis

Our analysis of repositories revealed that many regular expressions have subexpressions in common. Sometimes they share exact copies of the same subexpression, while sometimes the expressions are very similar.

This indicates that there is a huge need, and also an opportunity, for the reuse of regular expressions. Specifically, the commonality can be exploited in two ways. First, if someone wants to create a new regular expression for a specific application, it is likely that some parts of that expression exist already as a subexpression in a repository. The problem is how to find them. If the names used in the decompositions produced as part of our explanations were descriptive enough, then these domain-specific names could be used to search or browse repositories. This will work increasingly well over the middle and long run, when explanations with descriptive names have found their way into existing repositories.

The purpose of user-directed intent analysis is to provide explanations with more descriptive names, which aids the understanding of the regular expression itself, but can also be exploited for searching repositories as described. As an example, consider the following regular expression for dates.

```
(((0[13578]|[13578]|1[02])/([1-9]|[1-2][0-9]|3[01]))|((0[469]|[469]|11)
/([1-9]|[1-2][0-9]|30))|((2|02)-([1-9]|[1-2][0-9])))/[0-9]{4}
```

[3] Although we have considered the notion of *repetition formats*, we did not include them since they would have complicated the format identification process considerably. On the other hand, our analysis presented in Section 4 seems to indicate that the lack of repetition formats can limit the applicability of format analysis.

The fact that it describes dates is not inherent in the regular expression itself. However, once we are given this piece of information, we try to match its subexpressions to annotated date (sub)expressions in the repository. In this case, we find that `0[13578]|[13578]|1[02]` is used to describe months that have 31 days. At this point, this is just a guess, which can be communicated to the user. Once the user confirms the interpretation, this information can be exploited for an explanation.

Specifically, we can apply decomposition with a name that reflects this information. Moreover, we can suggest potential improvements to the expression. For example, in this case we can suggest to replace the first two alternatives `0[13578]|[13578]` by an optional prefix, that is, by `0?[13578]`. If we continue the interpretation, we can identify two other components, months with 30 days and February, the month with 29 days. Accepting all suggested interpretations, decomposition will thus produce the following explanation.

((*month-with-31-days*/*31-days*)| (*month-with-30-days*/*30-days*)| (*february*/*29-days*))/*year*
where
month-with-31-days = `0?[13578]|1[02]`
month-with-30-days = `0?[469]|11`
february = `2|02`
31-days = `[1 ... 31]`
30-days = `[1 ... 30]`
29-days = `[1 ... 29]`
year = `[0-9]{4}`

From our analysis of online repositories and discussion sites of regular expressions we found that the majority of regular expressions that are actively used and that users ask about are taken from 30 or so categories, which shows that with limited annotation effort one can have a big impact by providing improved regular expression decomposition using intention analysis.

3.4 Combined Explanations

The different explanation representations explored so far serve different purposes and each help explain a different aspect of a regular expression. Ultimately, all the different representations should work together to provide maximal explanatory benefit. To some degree this cooperation between the explanations is a question of GUI design, which is beyond the scope of the current paper.

We therefore employ here the simple strategy of listing the different explanations together with the regular expression to be explained. We add one additional representational feature, namely the color coding of identified subexpressions. This enhances the explanation since it allows us to link subexpressions and their definitions to their occurrence in the original regular expression. As an example, consider the explanation of the embedded image regular expression shown in Figure 2.

For coloring nested definitions, we employ a simple visualization rule with preference to the containing color. This leaves us with non-nested colors. This rule can be seen in action in Figure 3 where the occurrence of the subexpression *29-days* as part of the expressions representing *30-days* and *31-days* is not color coded in the original regular expression.

▶REGULAR EXPRESSION

```
<\s*[aA]\s+[hH][rR][eE][fF]=f\s*>\s*<\s*[iI][mM][gG]\s+[sS][rR][cC]=f\s*>
[^<>]*<\s*/[iI][mM][gG]\s*>\s*<\s*/[aA]\s*>
```

▶STRUCTURE

```
< a␣href=f > < img␣src=f >[^<>]*< /img > < /a >
```

▶FORMAT(S)

```
<•=f><•=f>•</•></•>
```

Fig. 2. Combined explanation for the embedded image regular expression

4 Evaluation

We have evaluated our proposed regular expression explanations in two different ways. First, we analyzed the explanation notation using the cognitive dimensions framework [2]. The results are presented in Section 4.1. Then in Section 4.2, we show the result of analyzing the applicability of our approach using the `regexplib.com` repository.

4.1 Evaluating the Explanation Notation Using Cognitive Dimensions

Cognitive Dimensions [2] provide a common vocabulary and framework for evaluating notations and environments. The explanation notation was designed to avoid the problems that the original regular expression notation causes in terms of usability and understandability. The cognitive dimensions provide a systematic way of judging that effort. Here we discuss a few of the cognitive dimensions that immediately affect the explanation notation.

Viscosity. The concept of *viscosity* measures the resistance to change that often results from redundancy in the notation. High viscosity means that the notation is not supportive of changes and maintenance.

While regular expressions score very high in terms of viscosity, our explanation structures for regular expressions have low viscosity since the abstraction provided by naming and decomposition allows redundancy to be safely eliminated. Moreover, the automatic decomposition identifies and removes duplicates in regular expressions. For example, consider an extension of the regular expression for dates, shown in Figure 3, that uses either / or - as a separation character. Using raw regular expressions, we need to duplicate the entire expression, and add it as an alternative at the top level. In contrast, the explanation representation can safely factor out the commonality and avoid the redundancy.

▶REGULAR EXPRESSION

```
((((0[13578]|[13578]|1[02])/([1-9]|[0-2][0-9]|3[01]))|((0[469]|[469]|11)/
([1-9]|[0-2][0-9]|30))|((2|02)/([1-9]|[0-2][0-9])))/[0-9]{4}
```

▶STRUCTURE

((*month-with-31-days*/*31-days*)|(*month-with-30-days*/*30-days*)|(*february*/*29-days*))/*year*
where
month-with-31-days = 0?[13578]|1[02]
month-with-30-days = 0?[469]|11
february = 2|02
31-days = [1 .. 31]
30-days = [1 .. 30]
29-days = [1 .. 29]
year = [0-9]{4}

▶FORMAT(S)

•/•/•

Fig. 3. Combined explanation for the date expression

Closeness of Mapping. Any notation talks about objects in some domain. How closely the notation is aligned to the domain is measured in the dimension called *closeness of mapping*. Usually, the closer the notation to the domain is, the better since the notation then better reflects the objects and structures it talks about.

In principle, regular expressions are a notation for a rather abstract domain of strings, so considering closeness of mapping might not seem very fruitful. However, since in many cases regular expressions are employed to represent strings from a specific domain, the question actually *does* matter. Our notation achieves domain closeness in two ways. First, intent analysis provides, in consultation with the user, names for subexpressions that are taken from the domain and thus establish a close mapping that supports the explanatory value of the regular expression's decomposition. While this obviously can be of help to other users of the expression, this might also benefit the user who adds the names in the decomposition in gaining a better understanding by moving the expression's explanation closer to the domain. Second, explanations employ a specific notation for numeric ranges that is widely used and can thus be assumed to be easily understood. Again, Figure 3 provides an example where the ranges and their names for 30 days, 31 days, and 29 days are closer to the intended list of numbers than the corresponding original parts of the regular expressions.

Role Expressiveness. This dimension tries to measure how obvious the role of a component is in the solution as a whole. In our explanation notation the formats produced by format analysis separate the fixed strings from the data part, which directly points to the roles of the format strings. The format also identifies more clearly a sequence of match-

Table 1. Regular expression in different domains in the regexplib.com repository

Type	Selected	Type	Selected
Email	38	URI	74
Numbers	107	Strings	91
Dates and Times	134	Address and Phone	104
Markup or code	63	Miscellaneous	173

ings, which supports a procedural view of the underlying expression. Moreover, roles of subexpressions are exposed by structure analysis and decomposition techniques.

4.2 Applicability of Explanations

We have evaluated the potential reach and benefits of our methods through the following research questions.

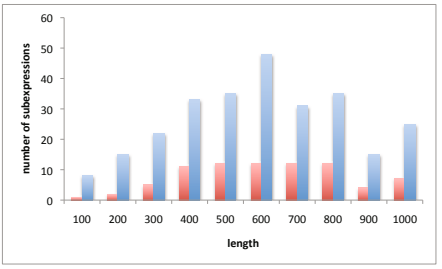
RQ1: Is the method of systematic hierarchical decomposition applicable to a significant number of regular expressions?
RQ2: How effective is our method of computing formats?
RQ3: Can we accurately identify the intended meanings of subexpressions in a regular expression given the context of the regular expression?
RQ4: How well does error detection work? In particular, can we identify inclusion errors for the regular expressions in real-world expressions?

The fourth research question came up during our work on RQ3. We noticed subtle differences in similar regular expressions and suspected that these were responsible for inaccuracies. We have developed a method of regular expression generalization that generates candidates of compatible regular expressions that can then be inspected for inclusion/exclusion errors, that is, strings that are incorrectly accepted/rejected by a regular expression.

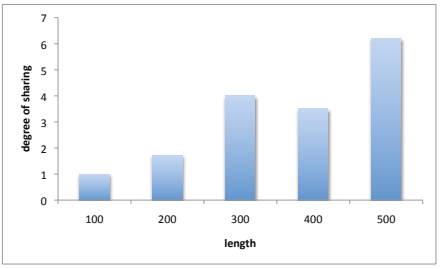
Setup. The publicly available online regular expression repository at `regexplib.com` was used for all our evaluations. This repository contains a total of 2799 user-supplied regular expressions for a variety of matching tasks. For our evaluations we manually eliminated eight syntactically invalid regular expressions as well as 18 expressions that were longer than 1000 characters to simplify our inspection and manual annotation work, leaving a total of 2773 regular expressions for analysis. Of these, 800 have been assigned to different domains. The total numbers of regular expressions in each domain are given in Table 1.

Test Results. For the evaluation of the first research question, we identified the subexpressions using our decomposition algorithm. For each expression we recorded the number of its (non-trivial)[4] subexpressions and averaged these numbers over regular expressions of similar length. That is, average was taken over intervals of 100 characters each. These are given by the long bars in Figure 4 on the left. The short bars show the averaged number of common subexpressions (that is, subexpression that occurred at least twice).

[4] A basic subexpressions was identified as the longest sequence of tokens that does not contain another group or subexpression.



(a) Frequency of subexpressions



(b) Degree of sharing

Fig. 4. Subexpression and Sharing Statistics. There was almost no sharing in expressions that are longer than 500 characters.

Considering one category in more detail, of the 134 regular expressions in the date-time category, 103 contained subexpressions. The maximum number of subexpressions was 11, and on average each expression contained about 3.25 subexpressions. The maximum number of nesting levels was 10, and on average expressions had 4 levels of nesting. The numbers in the other categories are very similar. We also determined the degree of sharing, that is, the number of times each identified subexpression occurred in the regular expression. The average degree of sharing, again averaged over expressions within a certain length range, is shown in Figure 4 on the right.

The most repeated single subexpression occurred 21 times. The total number of repeated subexpressions per regular expression was on average 3.9, the regular expression that contained the most repetitions had 42 repeated subexpressions. These numbers show that (1) there is quite a variety in number and repetitions of subexpressions, and (2) that decomposition has much structure to reveal and is widely applicable.

Since the evaluations of the second, third, and fourth research questions all required manual verification, we had to limit the scope of the considered regular expressions. For the second research question we used a randomized sample of 100 regular expressions selected from the complete repository. Format analysis was able to identify formats for 55 of those. The results are also shown in Table 2. We would have expected a larger number of formats. We suspect the rather low number is due to our ignoring repetition formats. A follow-up study should investigate how many more formats can be identified by using repetition formats.

Table 2. Analysis Applicability

Property	Found (Total)
Formats	55 (100)
Intentions	440 (1513)
Inclusion errors	39 (280)

For the evaluation of the third research question, we chose all of the 134 regular expressions from the date-time category. We applied our algorithm to identify the intent of each of the 1513 subexpressions that were contained in the 134 expressions. We could identify 440 subexpressions as having a specific intent. These results are also shown in Table 2. We believe that this number is quite encouraging and demonstrates that intent analysis can be widely applicable.

With regard to the fourth research question, we chose a randomized sample of 100 regular expressions from the date-time category. These regular expressions contained

280 subexpressions that we could identify from the context. We could detect 39 inclusion errors in these samples, but no exclusion errors. These results are summarized in Table 2. We believe that finding 39 (potential) faults in 134 regular expressions as a by-product of an explanation is an interesting aspect. Some of these are false positives since the intent provided by our analysis might be overly restrictive and not what the creator of the regular expression under consideration had in mind. Still, warnings about even only potential faults make users think carefully about what regular expression they are looking for, and they can thus further the understanding of the domain and its regular expressions.

4.3 Threats to Validity

The limited sample size may skew our results if the samples are not representative. Another threat to the validity is that the website `regexplib.com` may not be representative. A different threat to our research findings could be in our estimation of effectiveness of our techniques using cognitive dimensions framework rather than a user study.

5 Related Work

The problems of regular expressions have prompted a variety of responses, ranging from tools to support the work with regular expressions to alternative language proposals.

Prominent among tools are debuggers, such as the Perl built-in regular expression debugger and *regex buddy* (see `regexbuddy.com`). The Perl debugger creates a listing of all actions that the Perl matching engine takes. However, this listing can become quite large even for fairly small expressions. Regex buddy provides a visual tool that highlights the current match. There are also many tools that show subexpressions in a regular expression. The best known is rework [23], which shows the regular expression as a tree of subexpressions. However, it does not support naming or abstraction of common subexpressions. Several tools, such as Graphrex [6] and RegExpert [5], allow the visualization of regular expressions as a DF. All these approaches share the same limitations with respect to providing high-level explanations. Specifically, while these approaches help users understand why a particular string did or did not match, they do not provide any explanations for what the general structure of the regular expression is and what kind of strings the regular expression will match in general.

In the following we discuss a few alternatives that have been proposed to regular expressions.

Topes provides a system that allows a user to specify a format graphically without learning an arcane notation such as regular expressions [22]. The system internally converts specifications to augmented context-free grammars, and a parser provides a graded response to the validity of any string with respect to an expected format. The system also allows programmers to define "soft" constraints, which are often, but not necessarily always true. These constraints help in generating graded responses to possibly valid data that do not conform to the format.

Blackwell proposes a visual language to enter regular expressions [1]. It also provides a facility to learn regular expressions from given data (programming by example).

The system does provide multiple graphical notations for representing regular expressions. However, it is not clear how well this notation will scale when used for more complex regular expressions.

Lightweight structured text processing is an approach toward specifying the structure of text documents by providing a pattern language for text constraints. Used interactively, the structure of text is defined using multiple relationships [17]. The text constraint language uses a novel representation of selected text as collections of rectangles or region intervals. It uses an algebra over sets of regions where operators take region sets as arguments and generate region sets as result.

While most of these (and other) approaches arguably provide significant improvements over regular expressions, the fact that regular expressions are a de facto standard means that these tools will be used in only specific cases and that they do not obviate the need for a general explanation mechanism.

We have previously investigated the notion of explainability as a design criterion for languages in [9]. This was based on a visual language for expressing strategies in game theory. A major guiding principle for the design of the visual notation was the *traceabilty* of results. A different, but related form of tracing was also used in the explanation language for probabilistic reasoning problems [10].

Since regular expressions already exist, we have to design our explanation structures as extensions to the existing notation. This is what we have done in this paper. In particular, we have focused on structures that help to overcome the most serious problems of regular expressions—the lack of abstraction and structuring mechanisms. In future work we will investigate how the notion of traceability in the context of string matching can be integrated into our set of explanation structures.

6 Conclusions

We have identified several representations that can serve as explanations for regular expressions together with algorithms to automatically (or semi-automatically in the case of intention analysis) produce these representations for given regular expressions.

By comparing raw regular expressions with the annotated versions that contain decomposition structures, formats, and intentional interpretations, it is obvious—even without a user study—that our methods improve the understanding of regular expressions. The use of the developed explanation structures is not limited to explain individual regular expression. They can also help with finding regular expressions in repositories and identifying errors in regular expressions. This demonstrates that explanation structures are not simply "comments" that one might look at if needed (although even that alone would be a worthwhile use), but that they can play an active role in several different ways to support the work with regular expressions. Our evaluation shows that the methods are widely applicable in practice.

The limitations of regular expressions have prompted several designs for improved languages. However, it does not seem that they will be replaced with a new representation anytime soon. Therefore, since regular expressions are here to stay, any support that can help with their use and maintenance should be welcome. The explanation structures and algorithms developed in this paper are a contribution to this end.

References

1. Blackwell, A.F.: See What You Need: Helping End-users to Build Abstractions. J. Visual Languages and Computing 12(5), 475–499 (2001)
2. Blackwell, A.F., Green, T.R.: Notational Systems - The Cognitive Dimensions of Notations Framework. In: HCI Models, Theories, and Frameworks: Toward and Interdisciplinary Science, pp. 103–133 (2003)
3. Boroditsky, L.: Metaphoric structuring: understanding time through spatial metaphors. Cognition 75(1), 1–28 (2000)
4. Bransford, J.D., Johnson, M.K.: Contextual prerequisites for understanding: Some investigations of comprehension and recall. J. Verbal Learning and Verbal Behavior 11(6), 717–726 (1972)
5. Budiselic, I., Srbljic, S., Popovic, M.: RegExpert: A Tool for Visualization of Regular Expressions. In: EUROCON 2007. The Computer as a Tool, pp. 2387–2389 (2007)
6. Graphrex, http://crotonresearch.com/graphrex/
7. Curcio, F., Robbins, O., Ela, S.S.: The Role of Body Parts and Readiness in Acquisition of Number Conservation. Child Development 42, 1641–1646 (1971)
8. Derek, M.J.: The New C Standard: An Economic and Cultural Commentary. Addison-Wesley Professional (2003)
9. Erwig, M., Walkingshaw, E.: A Visual Language for Representing and Explaining Strategies in Game Theory. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing. pp. 101–108 (2008)
10. Erwig, M., Walkingshaw, E.: Visual Explanations of Probabilistic Reasoning. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing, pp. 23–27 (2009)
11. Friedl, J.: Now you have two problems, http://regex.info/blog/2006-09-15/247
12. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley Longman Publishing Co., Inc. (2006)
13. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular Expression Types for XML. In: Proc. of the International Conf. on Functional Programming (ICFP), pp. 11–22 (2000)
14. Hur, J., Schuyler, A.D., States, D.J., Feldman, E.L.: SciMiner: web-based literature mining tool for target identification and functional enrichment analysis. Bioinformatics (Oxford, England) 25(6), 838–840 (2009)
15. Lockwood, J.W., Moscola, J., Kulig, M., Reddick, D., Brooks, T.: Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware. In: Military and Aerospace Programmable Logic Device (MAPLD), p. 10 (2003)
16. Mahalingam, K., Bagasra, O.: Bioinformatics Tools: Searching for Markers in DNA/RNA Sequences. In: BIOCOMP, pp. 612–615 (2008)
17. Miller, R.C., Myers, B.A.: Lightweight Structured Text Processing. In: USENIX Annual Technical Conf., pp. 131–144 (1999)
18. Nakata, A., Higashino, T., Taniguchi, K.: Protocol synthesis from context-free processes using event structures. In: Int. Conf. on Real-Time Computing Systems and Applications, pp. 173–180 (1998)
19. Pike, R.: Structural Regular Expressions. In: EUUG Spring Conf., pp. 21–28 (1987)
20. Regular Expressions, http://en.wikipedia.org/wiki/Regular_expression
21. Sanfilippo, L., Voorhis, J.V.: Categorizing Event Sequences Using Regular Expressions. IASSIST Quarterly 21(3), 36–41 (1997)
22. Scaffidi, C., Myers, B., Shaw, M.: Topes: reusable abstractions for validating data. In: Int. Conf. on Software Engineering, pp. 1–10 (2008)
23. Steele, O.: http://osteele.com/tools/rework/
24. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic Network Protocol Analysis. In: Annual Network and Distributed System Security Symp., NDSS 2008 (2008)

On the Danger of Coverage Directed Test Case Generation

Matt Staats[1], Gregory Gay[2], Michael Whalen[2], and Mats Heimdahl[2]

[1] Korea Advanced Institute of Science & Technology, Daejeon, Republic of Korea
[2] University of Minnesota, Minneapolis MN, USA
staatsm@kaist.ac.kr, greg@greggay.com, {whalen,heimdahl}@cs.umn.edu

Abstract. In the avionics domain, the use of structural coverage criteria is legally required in determining test suite adequacy. With the success of automated test generation tools, it is tempting to use these criteria as the basis for test generation. To more firmly establish the effectiveness of such approaches, we have generated and evaluated test suites to satisfy two coverage criteria using counterexample-based test generation and a random generation approach, contrasted against purely random test suites of equal size.

Our results yield two key conclusions. First, coverage criteria satisfaction alone is a poor indication of test suite effectiveness. Second, the use of structural coverage as a supplement—not a target—for test generation can have a positive impact. These observations points to the dangers inherent in the increase in test automation in critical systems and the need for more research in how coverage criteria, generation approach, and system structure jointly influence test effectiveness.

1 Introduction

In software testing, the need to determine the adequacy of test suites has motivated the development of several test coverage criteria [1]. One such class of criteria are *structural coverage criteria*, which measure test suite adequacy in terms of coverage over the structural elements of the system under test. In the domain of critical systems—particularly in avionics—demonstrating structural coverage is required for certification [2]. In recent years, there has been rapid progress in the creation of tools for automatic directed test generation for structural coverage criteria [3–5]; tools promising to improve coverage and reduce the cost associated with test creation.

In principle, this represents a success for software engineering research: a mandatory—and potentially arduous—engineering task has been automated. However, while there is some evidence that using structural coverage to guide random test generation provides better tests than purely random tests, the effectiveness of test suites automatically generated to satisfy various structural coverage criteria has not been firmly established. In pilot studies, we found that test inputs generated specifically to satisfy three structural coverage criteria via counterexample-based test generation were *less effective* than random test inputs [6]. Further, we found that reducing larger test suites providing a certain

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 409–424, 2012.

coverage—in our case MC/DC—while maintaining coverage reduced their fault finding significantly, hinting that it is not always wise to build test suites solely to satisfy a coverage criterion [7].

These results are concerning. Given the strong incentives and the ability to automate test generation, it is essential to ask: *"Are test suites generated using automated test generation techniques effective?"* In earlier studies, we used a single system to explore this question. In this paper, we report the results of a study conducted using four production avionics systems from Rockwell Collins Inc. and one example system from NASA. Our study measures the fault finding effectiveness of automatically generated test suites satisfying two structural coverage criteria, branch coverage and Modified Condition Decision Coverage (MC/DC coverage) as compared to randomly generated test suites of the same size. We generate tests using both counterexample-based test generation and a random generation approach. In our study we use mutation analysis [8] to compare the effectiveness of the generated test suites as compared to purely randomly generated test suites of equal size.

Our results show that for both coverage criteria, in our industrial systems, the automatically generated test suites perform *significantly worse* than random test suites of equal size when coupled with an output-only oracle (5.2% to 58.8% fewer faults found). On the other hand, for the NASA example—which was selected specifically because its structure is significantly different from the Rockwell Collins systems—test suites generated to satisfy structural coverage perform dramatically better, finding 16 times as many faults as random test suites of equal size. Furthermore, we found that for most combinations of coverage criteria and case examples, randomly generated test suites reduced while maintaining structural coverage find *more* faults than pure randomly generated test suites of equal size, finding up to 7% more faults.

We draw two key conclusions from these results. First, automatic test generation to satisfy branch or MC/DC coverage does not, for the systems investigated, yield effective tests relative to their size. This in turn indicates that satisfying even a highly rigorous coverage criterion such as MC/DC is a poor indication of test suite effectiveness. Second, the use of branch or MC/DC as a supplement—not a target—for test generation (as Chilensky and Miller recommend in their seminal work on MC/DC [9]) does appear effective.

These results in this paper highlight the need for more research in how the coverage criterion, test generation approach, and the structure of the system under test jointly influence the effectiveness of testing. The increasing availability and use of advanced test-generation tools coupled with our lack of knowledge of their effectiveness is worrisome and careful attention must be paid to their use and acceptance.

2 Related Work

There exist a number of empirical studies comparing structural coverage criteria with random testing, with mixed results. Juristo et al. provide a survey of much

of the existing work [10]. With respect to branch coverage, they note that some authors (such as Hutchins et al. [11]) find that it outperforms random testing, while others (such as Frankl and Weiss [12]) discover the opposite. Namin and Andrews have found coverage levels are positively correlated with fault finding effectiveness [13]. Theoretical work comparing the effectiveness of partition testing against random testing yields similarly mixed results. Weyuker and Jeng, and Chen and Yu, indicated that partition testing is not necessarily more effective than random testing [14, 15]. Later theoretical work by Gutjahr [16], however, provides a stronger case for partition testing. Arcuri et al. [17] recently demonstrated that in many scenarios, random testing is more predictable and cost-effective at reaching high levels of structural coverage than previously thought. The authors have also demonstrated that, when cost is taken into account, random testing is often more effective at detecting failures than a popular alternative—adaptive random testing [18].

Most studies concerning automatic test generation for structural coverage criteria are focused on how to generate tests quickly and/or improve coverage [19, 3]. Comparisons of the fault-finding effectiveness of the resulting test suites against other methods of test generation are few. Those that exist apart from our own limited previous work are, to the best of our knowledge, studies in concolic execution [4, 5]. One concolic approach by Majumdar and Sen [20] has even merged random testing with symbolic execution, though their evaluation only focused on two case examples, and did not explore fault finding effectiveness.

Despite the importance of the MC/DC criterion [9, 2], studies of its effectiveness are few. Yu and Lau study several structural coverage criteria, including MC/DC, and find MC/DC is cost effective relative to other criteria [21]. Kandl and Kirner evaluate MC/DC using an example from the automotive domain, and note less than perfect fault finding [22]. Dupuy and Leveson evaluate the MC/DC as a compliment to functional testing, finding that the use of MC/DC improves the quality of tests [23]. None of these studies, however, compare the effectiveness of MC/DC to that of random testing. They therefore do not indicate if test suites satisfying MC/DC are truly effective, or if they are effective merely because MC/DC test suites are generally quite large.

3 Study

Of interest in this paper are two broad classes of approaches: random test generation and directed test generation. In random test generation, tests are randomly generated and then later reduced with respect to the coverage criterion. This approach is useful as a gauge of value of a coverage criterion: if tests randomly generated and reduced with respect to a coverage criterion are more effective than pure randomly generated tests, we can safely conclude the use of the coverage criterion led to the improvement. Unfortunately, evidence demonstrating this is, at best, mixed for branch coverage [10], and non-existent for MC/DC coverage.

Directed test generation is specifically targeted at satisfying a coverage criterion. Examples include heuristic search methods and approaches based on reachability [19, 3, 4]. Such techniques have advanced to the point where they can be effectively applied to real-world avionics systems. Such approaches are usually slower than random testing, but offer the potential to improve the coverage of the resulting test suites. We aim to determine if using existing directed generation techniques with these criteria results in test suites more effective than randomly generated test suites. Evidence addressing this is sparse and, for branch and MC/DC coverage, absent from the critical systems domain.[1]

We expect that a test suite satisfying the coverage criterion to be, at a minimum, at least as effective as randomly generated test suites of equal size. Given the central—and mandated—role the coverage criteria play within certain domains (e.g., DO-178B for airborne software [2]), and the resources required to satisfy them, this area requires additional study. We thus seek answers to the following research questions:

RQ1: *Are random test suites reduced to satisfy branch and MC/DC coverage more effective than purely randomly generated test suites of equal size?*

RQ2: *Are test suites directly generated to satisfy branch and MC/DC coverage more effective than randomly generated test suites of equal size?*

We explore two structural coverage criteria: branch coverage, and MC/DC coverage [10, 9]. Branch coverage is commonly used in software testing research and improving branch coverage is a common goal in automatic test generation. MC/DC coverage is a more rigorous coverage criterion based on exercising complex Boolean conditions (such as the ones present in many avionics systems), and is required when testing critical avionics systems. Accordingly, we view it as likely to be an effective criterion—particularly for the class of systems studied in this report.

3.1 Experimental Setup Overview

In this study, we have used four industrial systems developed by Rockwell Collins, and a fifth system created as a case example at NASA. The Rockwell Collins systems were modeled using the Simulink notation and the NASA system using Stateflow [25, 26], and were translated to the Lustre synchronous programming language [27] to take advantage of existing automation. Two of these systems, *DWM_1* and *DWM_2*, represent portions of a Display Window Manager for a commercial cockpit display system. The other two systems—*Vertmax_Batch* and *Latctl_Batch*—represent the vertical and lateral mode logic

[1] It has been suggested that structural coverage criteria should *only* be used to determine if a test suite has failed to cover functionality in the source code [1, 13]. Nevertheless, test suite adequacy measurement can always be transformed into test suite generation. In mandating that a coverage criterion be used for measurement, it seems inevitable that some testers will opt to perform generation to speed the testing process, and such tools already exist [24].

for a Flight Guidance System (FGS). The NASA system, *Docking_Approach*, describes the behavior of a space shuttle as it docks with the International Space Station.

Information related to these systems is provided in Table 1. We list the number of Simulink subsystems, which are analogous to functions, and the number of blocks, which are analogous to operators. For the NASA example developed in Stateflow, we list the number of states, transitions, and variables.

Table 1. Case Example Information

	# Simulink Subsystems	# Blocks
DWM_1	3,109	11,439
DWM_2	128	429
Vertmax_Batch	396	1,453
Latctl_Batch	120	718

	# Stateflow States	# Transitions	# Vars
Docking_Approach	64	104	51

For each case example, we performed the following steps: (1) mutant generation (described in Section 3.2), (2) random and structural test generation (Section 3.3 and 3.4), and (3) computation of fault finding (Section 3.5).

3.2 Mutant Generation

We have created 250 *mutants* (faulty implementations) for each case example by introducing a single fault into the correct implementation. Each fault was seeded by either inserting a new operator into the system or by replacing an existing operator or variable with a different operator or variable. The mutation operators used in this study are fairly typical and are discussed at length in [28]. They are similar to the operators used by Andrews et al. where they conclude that mutation testing is an adequate proxy for real faults [29].

One risk of mutation testing is *functionally equivalent* mutants—the scenario in which faults exist, but these faults cannot cause a *failure* (an externally visible deviation from correct behavior). This presents a problem when using oracles that consider internal state: we may detect failures that can never propagate to the output. For our study, we used NuSMV to detect and remove functionally equivalent mutants for the four Rockwell Collins systems[2]. This is made possible thanks to our use of synchronous reactive systems—each system is finite, and thus determining equivalence is decidable[3].

The complexity of determining non-equivalence for the *Docking_Approach* system is, unfortunately, prohibitive, and we only report results using the

[2] The percentage of mutants removed is very small, 2.8% on average.

[3] Equivalence checking is fairly routine in the hardware side of the synchronous reactive system community; a good introduction can be found in [30].

output-only oracle. Therefore, for every mutant reported as killed in our study, there exists at least one trace that can lead to a user-visible failure, and all fault finding measurements indeed measure faults detected.

3.3 Test Data Generation

We generated a single set of 1,000 random tests for each case example. The tests in this set are between 2 and 10 steps (evenly distributed in the set). For each test step, we randomly selected a valid value for all inputs. As all inputs are scalar, this is trivial. We refer to this as a *random test suite*.

We have directly generated test suites satisfying the branch and MC/DC [10, 31] criteria. Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is a common criterion within the avionics community [31].

For our directed test generation approach, we used counterexample-based test generation to generate tests satisfying branch and MC/DC coverage [19, 3]. In this approach each coverage obligation is encoded as a temporal logic formula and the model checker can be used to detect a counterexample (test case) illustrating how the coverage obligation can be covered. This approach guarantees that we achieve the maximum possible coverage of the system under test. This guarantee is why we have elected to use counterexample-based test generation, as other directed approaches (such as concolic/SAT-based approaches) do not offer such a straightforward guarantee. We have used the NuSMV model checker in our experiments [32] because we have found that it is efficient and produces tests that are both simple and short [6].

Note that as all of our case examples are modules of larger systems, the tests generated are effectively *unit tests*.

3.4 Test Suite Reduction

Counterexample-based test generation results in a separate test for each coverage obligation. This leads to a large amount of redundancy in the tests generated, as each test likely covers several obligations. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to provide coverage. Given the correlation between test suite size and fault finding effectiveness [13], this has the potential to yield misleading results—an unnecessarily large test suite may lead us to conclude that a coverage criterion has led us to select effective tests, when in reality it is the size of the test suite that is responsible for its effectiveness. To avoid this, we reduce each naïvely generated test suite while maintaining the coverage achieved. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the possible reduced test suites, we produce 50 different test suites for each case example using this process.

Per *RQ1*, we also create tests suites satisfying branch and MC/DC coverage by reducing the random test suite with respect to the coverage criteria (that is, the suite is reduced while maintaining the coverage level of the unreduced suite). Again, we produce 50 tests suites satisfying each coverage criterion.

For both counterexample-based test generation and random testing reduced with respect to a criterion, reduction is done using a simple greedy algorithm. We first determine the coverage obligations satisfied by each test generated, and initialize an empty test set *reduced*. We then randomly select a test input from the full set of tests; if it satisfies obligations not satisfied by any test input in *reduced*, we add it to *reduced*. We continue until all tests have been removed from the full set of tests.

For each of our existing reduced test suites, we also produce a purely random test suite of equal size using the set of random test data. We measure suite size in terms of the number of total test steps, rather than the number of tests, as random tests are on average longer than tests generated using counterexample-based test generation. These random suites are used as a baseline when evaluating the effectiveness of test suites reduced with respect to coverage criteria. We also generate random test suites of sizes varying from 1 to 1,000. These tests are not part of our analysis, but provide context in our illustrations.

When generating tests suites to satisfy a structural coverage criterion, the suite size can vary from the minimum required to satisfy the coverage criterion (generally unknown) to infinity. Previous work has demonstrated that test suite reduction can have a negative impact on test suite effectiveness [7]. Despite this, we believe the test suite size most likely to be used in practice is one designed to be small—reduced with respect to coverage—rather than large (every test generated in the case of counterexample-based generation or, even more arbitrarily, 1,000 random tests)[4].

3.5 Computing Fault Finding

In our study, we use *expected value oracles*, which define concrete expected values for each test input. We explore the use of two oracles: an *output-only oracle* that defines expected values for all outputs, and a *maximum oracle* that defines expected values for all outputs and all internal state variables. The output-only oracle represents the oracle most likely to be used in practice. Both oracles have been used in previous work, and thus we use both to allow for comparison. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or "killed").

4 Results and Analysis

We present the fault finding results in Tables 2 and 3, listing for each case example, coverage criterion, test generation method, and oracle: the average fault finding for test suites reduced to satisfy a coverage criterion, next to the

[4] One could build a counterexample-based test suite generation tool that, upon generating a test, removes from consideration *all* newly covered obligations, and randomly selects a new uncovered obligation to try to satisfy, repeating until finished. Such a tool would produce test suites equivalent to our reduced test suites, and thus require no reduction; alternatively, we could view such test suites as pre-reduced.

Table 2. Average number of faults identified, branch coverage criterion. OO = Output-Only, MX = Maximum

		Counterexample Generation				Random Generation			
Case Example	**Oracle**	**Satisfying Branch**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying Branch**	**Random of Same Size**	**% Change**	**p-val**
Latctl_Batch	MX	217.0	215.8	0.6%	0.24	238.7	234.4	1.8%	
	OO	82.2	140.3	-41.4%		196.4	189.2	3.8%	
Vertmax_Batch	MX	211.2	175.3	20.5%		219.5	209.7	4.6%	< 0.01
	OO	77.1	101.7	-24.2%		153.5	143.4	7.0%	
DWM_1	MX	195.1	227.9	-14.4%	< 0.01	230.2	227.1	1.4%	
	OO	32.1	77.9	-58.8%		79.8	76.9	3.77%	0.04
DWM_2	MX	202.1	215.5	-6.2%		232.0	225.8	2.7%	< 0.01
	OO	131.9	174.7	-24.5%		200.3	192.3	4.2%	
Docking_Approach	OO	38.1	2.0	1805%		2.0	2.0	0.0%	1.0

Table 3. Average number of faults identified, MCDC criterion. OO = Output-Only, MX = Maximum

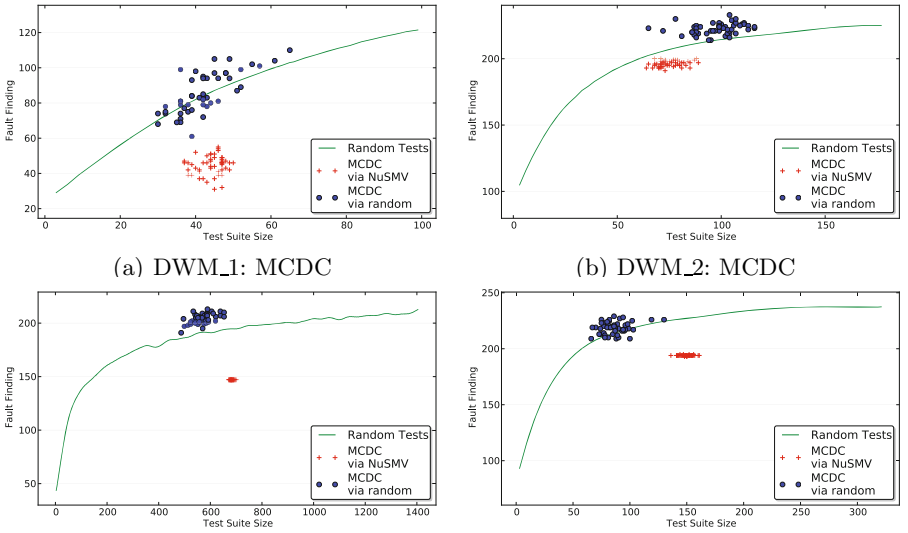
		Counterexample Generation				Random Generation			
Case Example	**Oracle**	**Satisfying MCDC**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying MCDC**	**Random of Same Size**	**% Change**	**p-val**
Latctl_Batch	MX	235.0	241.8	-2.8%		241.5	240.3	0.3%	
	OO	194.2	226.7	-14.4%		218.6	214.6	1.9%	< 0.01
Vertmax_Batch	MX	248.0	239.3	3.6%	< 0.01	248.0	237.1	4.6%	
	OO	147.0	195.5	-24.8%		204.2	191.4	6.7%	
DWM_1	MX	210.0	230.4	-12.8%		230.6	229.5	0.4%	0.048
	OO	44.6	86.6	-48.5%		85.4	86.4	-1.2%	0.6
DWM_2	MX	233.7	232.2	0.7%	0.08	241.9	235.5	2.7%	< 0.01
	OO	196.2	207.0	-5.2%	< 0.01	222.6	213.5	4.3%	
Docking_Approach	OO	37.34	2.0	1750%		2.0	2.0	0.0%	1.0

Table 4. Coverage Achieved (of Maximum Coverage) by Randomly Generated Test Suites Reduced to Satisfy Coverage Criteria

	Branch Coverage	**MCDC Coverage**
DWM_1	100.0%	100.0%
DWM_2	100.0%	97.76%
Vertmax_Batch	100.0%	99.4%
Latctl_Batch	100.0%	100.0%
Docking_Approach	58.1%	37.76%

average fault finding for random test suites of equal size; the relative change in average fault finding when using the test suite satisfying the coverage criteria versus the random test suite of equal size; and the p-value for the statistical analysis below. Note that negative values for *% Change* indicate the test suites satisfying the coverage criterion are less effective on average than random test suites of equal size.

The test suites generated using counterexample-based test generation are guaranteed to achieve the maximum achievable coverage, but the randomly generated test suites reduced to satisfy structural coverage criteria are not. We therefore present the coverage achieved by these test suites (with 100% representing the maximum *achievable* coverage) in Table 4.



(a) DWM_1: MCDC (b) DWM_2: MCDC

(c) Vertmax_Batch: MCDC (d) Latctl_Batch: MCDC

Fig. 1. Faults identified compared to test suite size using NuSMV-generated test suites ('+'), randomly generated test suites reduced to satisfy a coverage criterion ('o'), and pure random test generation (line). Output-only oracles.

In Figure 1, we plot, for MC/DC coverage and four case examples, the test suites size and fault finding effectiveness of every test suite generated when using the output-only oracle.[5] Test suites generated via counterexample-based test generation are shown as pluses, random test suites reduced to satisfy structural coverage criteria are shown as circles, and random test suites of increasing size (including those paired with test suites satisfying coverage criteria) are shown in the line. The line has been smoothed with LOESS smoothing (with a factor of 0.3) to improve the readability of the figure. Note that, while 1,000 random test inputs have been generated, we have only plotted random test suites (i.e., the line) of sizes slightly larger than the test suites satisfying coverage criteria to maintain readability.

4.1 Statistical Analysis

For both *RQ1* and *RQ2*, we are interested in determining if test suites satisfying structural coverage criteria outperform purely random test suites of equal size. We begin by formulating statistical hypotheses H_1 and H_2:

H_1: A test suite generated using random test generation to provide structural coverage will find more faults than a pure random test suite of similar size.

[5] For reasons of space, plots for branch coverage and the maximum oracle are omitted. Figures for the *Docking_Approach* case example are not very illustrative.

H_2: A test suite generated using counterexample-based test generation to provide structural coverage will find more faults than a random test suite of similar size.
We then formulate the appropriate null hypotheses:
$H0_1$: A test suite generated using random test generation to provide structural coverage will find the same number of faults as a pure random test suite of similar size.
$H0_2$: A test suite generated using counterexample-based test generation to provide structural coverage will find the same number of faults as a random test suite of similar size.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate $H0_1$ and $H0_2$ without any assumptions on the distribution of our data, we use the two-tailed bootstrap paired permutation test (a non-parametric test with no distribution assumptions [33]) with 250,000 samples. We pair each test suite reduced to satisfy a coverage criteria with a purely random test suite of equal size. We then apply this statistical test for each case example, structural coverage criteria, and test oracle with $\alpha = 0.05$.[6]

4.2 Evaluation of RQ1 and RQ2

Based on the p-values less than 0.05 in Tables 2 and 3, we *reject* $H0_1$ for nearly all case examples and coverage criteria when using either oracle.[7] For cases with differences that are statistically significant, test suites reduced to satisfy coverage criteria are more effective than purely randomly generated test suites of equal size; for these combinations, we accept H_1. Our results confirm that branch and MC/DC coverage can be effective metrics for test suite adequacy within the domain of critical avionics systems: reducing test suites generated via a non-directed approach to satisfy structural coverage criteria is at least not harmful, and in some instances improves test suite effectiveness relative to their size by up to 7.0%. Thus, considering branch and MC/DC coverage when using random test generation generally leads to a positive, albeit slight, improvement in test suite effectiveness.

Based on the p-values less than 0.05 in Tables 2 and 3, we *reject* $H0_2$ for all case examples and coverage criteria when using the output-only oracle. However, for all but one case example, test suites generated via counterexample-based test generation are *less* effective than pure random test suites by 5.2% to 58.8%; we therefore conclude that our initial hypothesis H_2 is false[8]. Nevertheless, the converse of H_2—randomly generated test suites are more effective that equally large

[6] Note that we do not generalize across case examples, oracles or coverage criteria, as the needed statistical assumption, random selection from the population of case examples, oracles, or coverage criteria, is not met. The statistical tests are used only demonstrate that observed differences are unlikely to have occurred by chance.

[7] We do not reject $H0_1$ for the *DWM_1* case example when using MC/DC coverage and the output-only oracle, nor do we reject $H0_1$ for the *Docking_Approach* case example.

[8] In our previous work we found the opposite effect [34].

test suites generated via counterexample-based test generation—is also false, as the *Docking_Approach* example illustrates. For this case example, random testing is effectively useless, finding a mere 2 faults, while tests generated using counterexample-based test generation find 37-38 faults. We discuss the reasons behind, and implications of, this strong dichotomy in Section 5.

When using the maximum oracle, we see that the test suites generated via counterexample-based test generation fare better. In select instances, counterexample-based test suites outperform random test suites of equal size (notably *Vertmax_Batch*), and otherwise close the gap, being less effective than pure random test suites by at most 14.4%. Nevertheless, we note that for most case examples and coverage criteria, random test suites of equal size are still more effective.

5 Discussion

Our results indicate that for our systems (1) the use of branch and MC/DC coverage as a *supplement* to random testing generally results in more effective tests suites than random testing alone, and (2) the use of branch and MC/DC coverage as a *target* for directed, automatic test case generation (specifically counterexample-based test generation) results in *less* effective test suites than random testing alone, with decreases of up to 58.8%. This indicates that branch and MC/DC coverage are—by themselves—not good indicators of test suite effectiveness. Given the role of structural coverage criteria in software validation in our domain of interest, we find these results quite troublesome.

The lack of effectiveness for test suites generated via counterexample-based test generation is a result of the formulation of these structural coverage criteria, properties of the case examples, and the behavior of NuSMV. We have previously shown that varying the structure of the program can significantly impact the number of tests required to satisfy the MC/DC coverage criterion [35]. These results were linked partly to *masking* present in the systems—some expressions in the systems can easily be prevented from influencing the outputs. This can reduce the effectiveness of a testing process based on structural coverage criteria, as we can satisfy coverage obligations for internal expressions without allowing resulting errors to propagate to the output.

This masking can be a problem; we have found that test inputs generated using counterexample-based generation (including those in this study) tend to be short, and manipulate only a handful of input values, leaving other inputs at default values (`false` or `0`) [6]. Such tests tend to exercise the program just enough to satisfy the coverage obligations for which they were generated and do not consider the propagation of values to the outputs. In contrast, random test inputs can vary arbitrarily in length (up to 10 in this study) and vary all input values; such test inputs may be more likely to overcome any masking present in the system.

As highlighted by the *Docking_Approach* example, however, tests generated to satisfy structural coverage criteria can sometimes dramatically outperform

random test generation. This example differs from the Rockwell Collins systems chiefly in its structure: large portions of the system's behavior are activated only when very specific conditions are met. The state space is both deep and contains bottlenecks; exploration of states requires relatively long tests with specific combinations of input values. Thus, random testing is highly unlikely to reach much of the state space. The impact of structure on the effectiveness of random testing can be seen in the coverage of the *Docking_Approach* (only 37.7% of obligations were covered) and is in contrast to the Rockwell Collins systems which—while stateful—have a state space that is shallow and highly interconnected and is, therefore, easier to cover with random testing.

We see two key implications in our results. First, per *RQ1*, using branch and MC/DC coverage as an addition to another non-structure-based testing method—in this case, random testing—can yield improvements (albeit small) in the testing process. These results are similar to those of other authors, for example, results indicating MC/DC is an effective coverage criterion when used to check the adequacy of manual, requirement-driven test generation [23] and results indicating that reducing randomly generated tests with respect to branch coverage yields improvements over pure random test generation [13]. These results, in conjunction with the results for *RQ2*, reinforce the advice that coverage criteria are best applied after test generation to find areas of the source code that have not been tested. In the case of MC/DC this advice is explicitly stated in regulatory requirements and by experts on the use of the criterion [2, 9].

Second, the dichotomy between the *Docking_Approach* example and the Rockwell Collins systems highlights that while the current methods of determining test suite adequacy in avionics systems are themselves inadequate, some method of determining testing adequacy is needed. While current practice stipulates that coverage criteria should be applied after test generation, in practice, this relies on the honesty of the tester (it is not required in the standard). Therefore, it seems inevitable that at least some practitioners will use automatic test generation to reduce the cost of achieving the required coverage.

Assuming our results generalize, we believe this represents a serious problem. The tools are not at fault: we have asked these tools to produce test inputs satisfying branch and MC/DC coverage, and they have done so admirably; for example, satisfying MC/DC for the *Docking_Approach* example, for which random testing achieves a mere 37.7% of the possible coverage. The problem is that the coverage criteria are simply too weak, which allows for the construction of ineffective tests. We see two possible solutions. First, automatic test generation tools could be improved to avoid pitfalls in using structural coverage criteria. For example, such tools could be encouraged to generate longer test cases increasing the chances that a corrupted internal state would propagate to an observable output (or other monitored variable). Nevertheless, this is a somewhat ad-hoc solution to weak coverage criteria and various tool vendors would provide diverse solutions rendering the coverage criteria themselves useless as certification or quality control tools.

Second, we could improve—or replace—existing structural coverage criteria. Automatic test generation has improved greatly in the last decade, thanks to improvements in search heuristics, SAT solving tools, etc. However, the targets of such tools have not been updated to account for this increase in power. Instead, we continue to use coverage criteria that were originally formulated when manual test generation was the only practical method of ensuring 100% coverage. New and improved coverage metrics are required in order to take full advantage of the improvements in automatic test generation without allowing the generation of inefficient test suites (such as some generated in our study).

6 Threats to Validity

External Validity: Our study has focused on a relatively small number of systems but, nevertheless, we believe the systems are representative of the class of systems in which we are interested, and our results are generalizable to other systems in the avionics domain.

We have used two methods for test generation (random generation and counterexample-based). There are many methods of generating tests and these methods may yield different results. Nevertheless, we have selected methods that we believe are likely to be used in our domain of interest.

For all coverage criteria, we have examined 50 test suites reduced using a simple greedy algorithm. It is possible that larger sample sizes may yield different results. However, in previous studies, smaller numbers of reduced test suites have been seen to produce consistent results [35].

Construct Validity: In our study, we measure fault finding over seeded faults, rather than real faults encountered during development. It is possible real faults would lead to different results. However, Andrews et al. showed that seeded faults leads to conclusions similar to those obtained using real faults [29].

We measure the cost of test suites in terms of the number of steps. Other measurements exist, e.g., the time required to generate and/or execute tests [34]. We chose size to be favorable towards directed test generation. Thus, conclusions concerning the inefficacy of directed test generation are reasonable.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions beyond these analyses are met, and have favored non-parametric methods. In cases in which the base assumptions are clearly not met, we have avoided using statistical methods. (Notably, we have avoided statistical inference *across* case examples.)

7 Conclusion and Future Work

The results presented in this paper indicate that coverage directed test generation may not be an effective means of creating tests within the domain of avionics systems, even when using metrics which can improve random test generation. Simple random test generation can yield equivalently sized, but more

effective test suites (up to twice as effective in our study). This indicates that adequacy criteria are, for the domain explored, potentially unreliable, and thus, unsuitable, for determining test suite adequacy.

The observations in this paper indicate a need for methods of determining test adequacy that (1) provide a reliable measure of test quality and (2) are better suited as targets for automated techniques. At a minimum, such coverage criteria must, when satisfied, indicate that our test suites are better than simple random test suites of equal size. Such criteria must address the problem *holistically* to account for all factors influencing testing, including the program structure, the nature of the state space of the system under test, the test oracle used, and the method of test generation.

Acknowledgements. This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, NSF grants CCF-0916583, CNS-0931931, CNS-1035715, and an NSF graduate research fellowship. Matt Staats was supported by the WCU (World Class University) program under the National Research Foundation of Korea and funded by the Ministry of Eduation, Science and Technology of Korea (Project No: R31-30007) We would also like to thank our collaborators at Rockwell Collins: Matthew Wilding, Steven Miller, and Darren Cofer. Without their continuing support, this investigation would not have been possible. Thank you!

References

1. Zhu, H., Hall, P.: Test data adequacy measurement. Software Engineering Journal 8(1), 21–29 (1993)
2. RTCA, DO-178B: Software Consideration. In: Airborne Systems and Equipment Certification. RTCA (1992)
3. Rayadurgam, S., Heimdahl, M.P.: Coverage based test-case generation using model checkers. In: Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems, pp. 83–91. IEEE Computer Society (April 2001)
4. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proc. of the 10th European Software Engineering Conf. / 13th ACM SIGSOFT Int'l. Symp. on Foundations of Software Engineering. ACM, New York (2005)
5. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI 2005: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (2005)
6. Heimdahl, M.P., Devaraj, G., Weber, R.J.: Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In: Proc. of the Eighth IEEE Int'l Symp. on High Assurance Systems Engineering (HASE), Tampa, Florida (March 2004)
7. Heimdahl, M.P., Devaraj, G.: Test-suite reduction for model based tests: Effects on test quality and implications for testing. In: Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering (ASE), Linz, Austria (September 2004)
8. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering (99), 1 (2010)

9. Chilenski, J.J., Miller, S.P.: Applicability of Modified Condition/Decision Coverage to Software Testing. Software Engineering Journal, 193–200 (September 1994)
10. Juristo, N., Moreno, A., Vegas, S.: Reviewing 25 years of testing technique experiments. Empirical Software Engineering 9(1), 7–44 (2004)
11. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In: Proc. of the 16th Int'l Conf. on Software Engineering. IEEE Computer Society Press, Los Alamitos (1994)
12. Frankl, P., Weiss, S.N.: An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In: Proc. of the Symposium on Testing, Analysis, and Verification (1991)
13. Namin, A., Andrews, J.: The influence of size and coverage on test suite effectiveness. In: Proc. of the 18th Int'l Symp. on Software Testing and Analysis. ACM (2009)
14. Weyuker, E., Jeng, B.: Analyzing partition testing strategies. IEEE Trans. on Software Engineering 17(7), 703–711 (1991)
15. Chen, T.Y., Yu, Y.T.: On the expected number of failures detected by subdomain testing and random testing. IEEE Transactions on Software Engineering 22(2) (1996)
16. Gutjahr, W.J.: Partition testing vs. random testing: The influence of uncertainty. IEEE Transactions on Software Engineering 25(5), 661–674 (1999)
17. Arcuri, A., Iqbal, M.Z.Z., Briand, L.C.: Formal analysis of the effectiveness and predictability of random testing. In: ISSTA 2010, pp. 219–230 (2010)
18. Arcuri, A., Briand, L.C.: Adaptive random testing: An illusion of effectiveness? In: ISSTA (2011)
19. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. Software Engineering Notes 24(6), 146–162 (1999)
20. Majumdar, R., Sen, K.: Hybrid concolic testing. In: ICSE, pp. 416–426 (2007)
21. Yu, Y., Lau, M.: A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. Journal of Systems and Software 79(5), 577–590 (2006)
22. Kandl, S., Kirner, R.: Error detection rate of MC/DC for a case study from the automotive domain. In: Software Technologies for Embedded and Ubiquitous Systems, pp. 131–142 (2011)
23. Dupuy, A., Leveson, N.: An empirical evaluation of the MC/DC coverage criterion on the hete-2 satellite software. In: Proc. of the Digital Aviation Systems Conference (DASC), Philadelphia, USA (October 2000)
24. Reactive systems inc. Reactis Product Description, http://www.reactive-systems.com/index.msp
25. Mathworks Inc. Simulink product web site, http://www.mathworks.com/products/simulink
26. Mathworks Inc. Stateflow product web site, http://www.mathworks.com
27. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic Press (1993)
28. Rajan, A., Whalen, M., Staats, M., Heimdahl, M.P.E.: Requirements Coverage as an Adequacy Measure for Conformance Testing. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 86–104. Springer, Heidelberg (2008)
29. Andrews, J., Briand, L., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: Proc of the 27th Int'l Conf on Software Engineering (ICSE), pp. 402–411 (2005)
30. Van Eijk, C.: Sequential equivalence checking based on structural similarities. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 19(7), 814–819 (2002)

31. Chilenski, J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Office of Aviation Research, Washington, D.C., Tech. Rep. DOT/FAA/AR-01/18 (April 2001)
32. The NuSMV Toolset (2005), http://nusmv.irst.itc.it/
33. Fisher, R.: The Design of Experiment. Hafner, New York (1935)
34. Devaraj, G., Heimdahl, M., Liang, D.: Coverage-directed test generation with model checkers: Challenges and opportunities. In: Annual International Computer Software and Applications Conference, vol. 1, pp. 455–462 (2005)
35. Rajan, A., Whalen, M., Heimdahl, M.: The effect of program and model structure on MC/DC test adequacy coverage. In: Proc. of the 30th Int'l Conference on Software Engineering, pp. 161–170. ACM, New York (2008)

Reduction of Test Suites Using Mutation

Macario Polo Usaola[1], Pedro Reales Mateo[1], and Beatriz Pérez Lamancha[2]

[1] Department of Information Systems and Technologies, University of Castilla-La Mancha,
Paseo de la Universidad 4, 13071-Ciudad Real, Spain
{macario.polo,pedro.reales}@uclm.es
[2] Software Testing Centre (CES), University of Republic
Lauro Müller 1989, Montevideo, Uruguay
bperez@fing.edu.uy

Abstract. This article proposes an algorithm for reducing the size of test suites, using the mutation score as the criterion for selecting the test cases while preserving the quality of the suite. Its utility is also checked with a set of experiments, using benchmark programs and industrial software.

Keywords: Test suites, mutation, test suite reduction, criteria subsumption.

1 Introduction

Mutation is a testing technique, originally proposed in 1978 by DeMillo et al. [1], which relies on the discovery of the artificial faults which are seeded in the system under test (SUT). These faults are injected in the SUT by means of a set of mutation operators, whose purpose is to imitate the faults that a common programmer may commit. Thus, each mutant is a copy of the program under test, but with a small change in its code, which is interpreted as a fault.

Mutants are usually generated by automated tools that apply a set of mutation operators to the sentences of the original program, thus producing a high number of mutants because, in general, each mutant contains only one fault. The fault in a mutant is discovered when the execution of a given test case produces a different output in the original program and in the mutant. When the fault is discovered, it is said that the mutant has been "killed"; otherwise, the mutant is "alive".

In order to obtain a good set of mutants, it is important that the seeded faults be "good", which depends on the quality of the mutation operators applied. This area has been closely studied, with the proposal of operators for different kinds of languages and environments, as for example in [2]. Faults introduced in the mutants must imitate common errors by programmers since, by means of the "coupling effect", a test suite that detects all simple faults in a program is so sensitive that it also detects more complex faults [3].

Figure 1 shows the source code of an original program (the SUT) and of some mutants: three of them proceed from the substitution of an arithmetic operator, whereas in the fourth a unary operator (++) has been added at the end of the sentence. The bottom of the figure presents the results obtained from executing some test cases

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 425–438, 2012.

on the different program versions. The test case corresponding to the test data *(1, 1)* produces different outputs in the original program (whose output is correct) and in Mutant 1: thus, this test case has found the fault introduced in the mutant, leaving the mutant killed. On the other hand, since all test cases offer the same output in the original program and in Mutant 4, it is said that Mutant 4 is alive. Moreover, this mutant will never be killed by any test case, since variable *b* is incremented *after* returning the result. Mutants like this one are called "functionally-equivalent mutants" and may be considered as noise when results are analyzed: they have a syntactic change (actually not a fault) with respect to the original source code that cannot be found.

Original	Mutant 1	Mutant 2	Mutant 3	Mutant 4

int sum(int a,int b)
{
return a + b;
}
	int sum(int a,int b)
	{
	return a - b;
	}
		int sum(int a,int b)
		{
		return a * b;
		}
			int sum(int a,int b)
			{
			return a / b;
			}
				int sum(int a,int b)
				{
				return a + b++;
				}

	Test data (a,b)			
	(1, 1)	(0, 0)	(-1, 0)	(-1, -1)
Orig.	2	0	-1	-2
Mut.1	0	0	-1	0
Mut.2	1	0	0	1
Mut.3	1	Error	Error	1
Mut.4	2	0	-1	-2

Fig. 1. Code of some mutants and their results with some test data

The test suite quality is measured in terms of the Mutation Score [4] (Figure 2), a number between 0 and 1 which takes into account the number of mutants killed, the number of mutants generated and the number of functionally-equivalent mutants. A test suite is mutation-adequate when it discovers all the faults injected in the mutants.

$MS(P,T) = \frac{K}{M - E}$,	being: *P*: program under test; *T*: test suite; *K*: # of killed mutants; *M:* # of generated mutants; *E*: # of equivalent mutants

Fig. 2. Mutation score

Since that paper by DeMillo in 1978, many works have researched and developed tools to improve the different steps of mutation testing: mutant generation, test case execution and result analysis.

Regarding **mutant generation**, most works try to decrease the number of mutants generated, with different studies existing for selecting the most meaningful operators [5, 6], as well as techniques for generating the mutants more quickly [7]. Regarding **test execution,** several authors have proposed the use weak mutation [8, 9], prioritization of the functions of the program under test [10] or the use of n-order mutants [11]. An n-order mutant has *n* faults instead of 1. Polo et al. [11] have shown that the combination of 1-order mutants to produce a suite of 2-order mutants significantly decreases the number of functionally equivalent mutants, whereas the risk of leaving faults undiscovered remains low. This has a positive influence on the **result analysis** step, whose main difficulty resides in the discovery of the functionally

equivalent mutants, which is required to calculate the Mutation Score (Figure 2). Manual detection is very costly, although Offutt and Pan have demonstrated that is possible to automatically detect almost 50% of functionally equivalent mutants if the program under test is annotated with constraints [12].

Since many equivalent mutants are optimizations or de-optimizations of the original program (for example, Mutant 4 in Figure 1 de-optimizes the original program), Offutt and Craft have also investigated how compiler optimization techniques may help in the detection of equivalent mutants [3].

In general, mutation testing has evolved over the years and, today, it is very frequently used to validate the quality of different testing techniques [13]. Some recent works related to mutation propose specific operators for specific programming languages, such as Kim et al. [14], who propose mutation operators for Java and Barbosa et al. [2], with operators for C.

These works, developed so many years after the proposal of mutation, evidence the maturity of this testing technique. With the adequate operators, the mutation score can be considered as a powerful coverage criterion [15].

This article proposes one algorithm (although another one, less efficient, is also described) for reducing the size of a test suite, based on the mutation score: given a test suite T, the goal is to obtain a new test suite T', which obtains the same mutation score as T, being $|T'| \leq T$. Furthermore, the article discusses how the subsumption of criteria may be preserved when the reduction algorithm is executed.

The article is organized as follows: the two parts of Section 2 briefly describe strategies for test case generation (where the problem of redundant test cases is presented) and some works solving the problem of test suite reduction. Section 3 then presents the algorithm for test suite reduction based on mutation, completing its description with an example taken from the literature. The validity of the algorithm is analyzed in Section 4, both with some benchmark programs, widely used in testing literature, and with a set of industrial programs. Finally, we draw our conclusions.

2 Related Work

The fact of having big test suites increases the cost of their writing, validation and maintenance, taking into account the continuous evolution of software and the corresponding regression testing [16]. Due to this, several researchers have proposed different techniques to reduce the size of a test suite, while the coverage reached is preserved. The problem of reducing a test suite to the minimum possible cardinal is known as the "optimal test-suite reduction problem" and has been stated by Jones and Harrold [17] as in Figure 3.

Given:	Test Suite T, a set of test-case requirements r_1, r_2, ..., r_n, that must be satisfied to provide the desired test coverage of the program.
Problem:	Find $T' \subset T$ such that T' satisfies all r_i and $(\forall T'' \subset T, T''$ satisfies all $r \Rightarrow |T'| \trianglelefteq |T''|)$

Fig. 3. The optimal test suite reduction problem

Applied to the *Triangle-type* example, and starting from the results obtained by the All *combinations* strategy, the problem consists of finding a minimal subset of test

cases that obtains the same coverage as the original test suite: i.e., to obtain a set of *n* test cases that reach the same coverage as the original suite, being $n \leq 216$ and *n* being the minimal. The optimal test-suite reduction problem is NP-hard [18] and, thus, its solution has been approached by means of algorithms which provide near-optimal solutions, usually with greedy strategies. The following subsections review some relevant works.

The HGS Algorithm. Harrold et al. [19] give a greedy algorithm (usually referred to as *HGS*) for reducing the suite of test cases into another, fulfilling the test requirements reached by the original suite. The main steps in this algorithm are:

1) Initially, all the test requirements are unmarked.
2) Add to *T'* those test cases that only exercise a test requirement. Mark the requirements covered by the selected test cases.
3) Order the unmarked requirements according to the cardinality of the set of test cases exercising one requirement. If several requirements are tied (since the sets of test cases exercising them have the same cardinality), select the test case that would mark the highest number of unmarked requirements tied for this cardinality. If multiple such test cases are tied, break the tie in favor of the test case that would mark the highest number of requirements with testing sets of successively higher cardinalities; if the highest cardinality is reached and some test cases are still tied, arbitrarily select a test case from among those tied. Mark the requirements exercised by the selected test. Remove test cases that become redundant as they no longer cover any of the unmarked requirements.
4) Repeat the above step until all testing requirements are marked.

Gupta Improvements. With different collaborators, Gupta has proposed several improvements to this algorithm:

- With Jeffrey [20], Gupta adds "selective redundancy" to the algorithm. "Selective redundancy" makes it possible to select test cases that, for any given test requirement, provide the same coverage as another previously selected test case, but that adds the coverage of a new, different test requirement. Thus, maybe *T'* reaches the All-branches criterion but not def-uses; therefore, a new test case t can be added to *T'* if it increases the coverage of the def-uses requirement: now, *T'* will not increase the All-branches criterion, but it will do so with def-uses.
- With Tallam [21], test case selection is based on Concept Analysis techniques. According to the authors, this version achieves same size or smaller size reduced test suites than prior heuristics as well as a similar time performance.

Heimdahl and George Algorithm. Heimdahl and George [22] also propose a greedy algorithm for reducing the test suite. Basically, they take a random test case, execute it and check the coverage reached. If this one is greater than the highest coverage, then they add it to the result. The algorithm is repeated five times to obtain five different reduced sets of test cases. Since chance is an essential component of this algorithm, the good quality of the results is not guaranteed.

McMaster and Memon Algorithm. McMaster and Memon [23] present another greedy algorithm. The parameter taken into account to include test cases in the reduced suite is based on the "unique call stacks" that test cases produce in the program under test. As can be seen, the criterion for selecting test cases (the number of unique call stacks) is not a "usual test requirement".

In **summary**, since the optimal test-suite reduction problem is NP-hard, all the approaches discussed propose a greedy algorithm to find a good solution with a polynomial-time algorithm and, as the discussed algorithms show, test requirement for test case selection can be anything: coverage of sentences, blocks, paths… or, as it is proposed in this paper, number of mutants killed.

According to [24, 25], the degree of automation of testing tasks in the software industry is very low. Often, testing is performed in an artisanal way, and the efforts carried out in the last years to obtain test automation mostly consist of the application of unit testing frameworks, such as JUnit or NUnit. As a matter of fact, the work by Ng et al. [26] shows the best results on test automation: 79.5% of surveyed organizations automate test execution and 75% regression testing. However, only 38 of the 65 organizations (58.5%) use test metrics, with defect count being the most popular (31 organizations). Although the work does not present any data about the testing tools used, these results suggest that most organizations are probably automating their testing processes with X-Unit environments. In order to improve these testing practices, software organizations require cost and time-effective techniques to automate and to improve their testing process. Thus, the introduction of software testing research results in industry is a must.

3 Test Suite Reduction Using Mutation

This section mainly describes a greedy algorithm to reduce the size of a test suite based on the Mutation Score. This algorithm is inspired in the mutation cost reduction algorithms briefly described in [27]. The number of mutants killed by each test case is used as the criterion for the inclusion of a test case in the reduced set of selected test cases.

Figure 4 shows *reduceTestSuite*, the main function of the algorithm. As inputs, it receives the complete set of test cases, the class under test and the complete set of mutants. In line 2, it executes all test cases against the class under test and against the mutants, saving the results in *testCaseResults*.

Then, the algorithm is prepared for selecting, in several iterations, the test cases that kill more mutants, what is done in the loop of lines 5-14.

The first time the algorithm enters this loop and arrives at line 7, the value of n (which is used to stop the iterations) is $|mutants|$: in this special case, the algorithm looks for a test case that kills all the mutants. If it finds it, the algorithm adds the test case to *requiredTC,* updates the value of n to 0 and ends; otherwise, it decreases n in line 16 and goes back into the loop.

Let us suppose that *n* is initially 100 (that is, there are 100 mutants of the class under test), and let us suppose that the algorithm does not find test cases that kill *n* mutants until *n=30*. With this value, the function *getTestCasesThatKillN* (called in line 7) returns as many test cases as test cases kill *n* different mutants: i.e., if there are two test cases (tc_1 and tc_2) that kill the same 30 mutants, *getTestCasesThatKillN* returns only one test case (for example, tc_1). If the intersection of the mutants killed by tc_1 and tc_2 is not empty, then the algorithm returns a set composed of tc_1 and tc_2.

```
1. reduceTestSuite(completeTC : SetOfTestCases, cut :
    CUT, mutants : SetOfMutants) : SetOfTestCases
2. testCaseResults = execute(completeTC, cut, mutants)
3. requiredTC = ∅
4. n=|mutants|
5. while (n>0)
6.  mutantsNowKilled = ∅
7.  testCasesThatKillN =
    getTestCasesThatKillN(completeTC, n, mutants,
    mutantsNowKilled, testCaseResults)
8.  if |testCasesThatKillN>0| then
9.   requiredTC = requiredTC ∪ testCasesThatKillN
10.    n = |mutants|-|mutantsNowKilled|
11.  else
12.    n = n - 1
13.  end if
14. end_while
15. return requiredTC
16.end
```

Fig. 4. Main function of the algorithm, which returns the reduced suite

When test cases killing the current *n* mutants are found, they are added to the *requiredTC* variable (line 9) and *n* is updated to the current number of remaining mutants.

In the actual implementation of the algorithm, the execution of the complete set of test cases against the CUT and the mutants is made in a separate function (*execute*, called in line 2), which returns a collection of *TestCaseResult* objects, which are composed of the name of a test case and the list of the mutants they kill.

The function in charge of collecting the set of test cases that kill *n* mutants is called in the 7th line in Figure 4 and is detailed in Figure 5. It goes through the elements in *testCaseResults* and takes those test cases whose list of has *n* elements. Each time it finds a suitable test case, the function removes the mutants it kills from the set of mutants: in this way, the function guarantees that no two test cases killing the same set of mutants will be included in the result.

```
1. getTestCasesThatKillN(completeTC:SetOfTestCases,
    n:int, mutants:SetOfMutants, mutantsNowKilled :
    SetOfMutants, testCaseResults: SetOfTestCaseResults)
    : SetOfTestCaseResults
2.  testCasesThatKillN = ∅
3.  for i=1 to |testCaseResults|
4.   testCaseResult = testCaseResults[i]
5.  if |testCaseResult.killedMutants| == n and
    testCaseResult.killedMutants ⊆ mutants then
6.   testCasesThatKillN = testCasesThatKillN ∪
    testCaseResult.testCaseName
7.     mutantsNowKilled = mutantsNowKilled ∪
    testCaseResult.killed.Mutants
8.   mutants = mutants – mutantsNowKilled
9.  end_if
10. next
11. return testCasesThatKillN
12.end
```

Fig. 5. Function to obtain the test cases that kill *n* mutants

3.1 Example

Let us suppose the killing matrix in Table 1, corresponding to a supposed program with eight mutants (in the rows) and a test suite with seven test cases. Each column may be understood as an instance of *TestCaseResult*: in fact, there are seven instances of this type, each composed of the test case name and the list of mutants it kills: the first test case result is composed of the *tc1* test case and the mutant *m2*; the second, by *tc2* and *m4, m5* and *m6*; the last one is composed of *tc7* and an empty set of mutants, since it kills none.

Table 1. First killing matrix for a supposed program

	tc1	tc2	tc3	tc4	tc5	tc6	tc7
m1			X		X		
m2	X		X		X		
m3			X				
m4		X				X	
m5		X					
m6		X		X			
m7				X			
m8				X			

Initially, *requiredTC* is the empty set and *n=7*. In the first iteration of the 5th line loop in Figure 4, the set *mutantsNowKilled* is ∅ because there are no test cases killing seven mutants. *n* is decreased to 6, 5, 4 and 3. In this iteration, the function *getTestCasesThatKillN* is called. This function (Figure 5) iterates from *i=1* to 7. When *i=1*, it rejects tc_1 because it does not kill three mutants (the current value of *n*). When *i=2:*

- Adds *{tc2}* to the *testCasesThatKillN* set.
- Adds *{m4, m5, m6}* to the *mutantsNowKilled* set.
- Leaves *mutants* with *{m1, m2, m3, m7,m8}*.

Henceforth, the killed mutants *{m4, m5, m6}* will not be considered in the following iterations. Then, the killing matrix can be now seen such as in Table 2.

Table 2. Second killing matrix for a supposed program

	tc1	tc2	tc3	tc4	tc5	tc6	tc7
m1			X		X		
m2			X		X		
m3			X	X			
m7				X			
m8				X			

Still inside *getTestCasesThatKillN,* the *i* variable is increased to 3 and the *TestCaseResult* corresponding to *tc3* is processed. Now:

- *testCasesThatKillN = {tc2, tc3}*
- *mutantsNowKilled = {m1, m2, m3, m4, m5, m6}*
- *mutants = {m7, m8}*

Mutants *{m1, m2, m3}* will not be considered in next iterations, thus leaving the killing matrix as in **Table 3**.

Table 3. Third killing matrix for a supposed program

	tc1	tc2	tc3	tc4	tc5	tc6	tc7
m7				X			
m8				X			

getTestCasesThatKillN continues increasing *i* to 7, the function exits and the algorithm returns to line 8 of *reduceTestSuite*. Here, *{tc2, tc3}* are added to *requiredTC* and *n* is decreased to 2, which is the initial number of mutants (8) minus the number of mutants now killed (6). Now, the function executes its last iteration calling *getTestCasesThatKillN* with n=*2*, and selects the *tc4* test case and adds it to *requiredTC*. The final value of this set is: *{tc2, tc3, tc4}*.

3.2 "A Motivational Example"

In their paper [20], Jeffrey and Gupta show, using the same title that leads this section, a small program to exemplify their algorithm with selective redundancy, which has been translated into the Java program shown in Figure 7. For this class, MuJava generates 48 traditional (15 of them are functionally-equivalent) and 6 class mutants for this program.

Using the values {-1.0, 0.0, +1.0} for the four parameters of function *f*, and generating test cases with the *All combinations* algorithm, the *testooj* tool [16] generates a test file with 3×3×3×3=81 test cases, that manage to kill 100% of the non-equivalent mutants. Figure 6 shows a piece of the killing matrix for this example, ordered according to the number of mutants killed by each test case (last column).

Of the three test cases killing 28 mutants, the algorithm picks *testTS_0_78* and adds it to the reduced suite, removing *testTS_0_76* and *testTS_0_77* since they kill the same mutants. Moreover, these are removed from the set of mutants. Then, it continues picking *testTS_054* and proceeds in the same way. The reduced suite is finally composed of 7 test cases, which is 8.6% of the original size: *{testTS_0_78, testTS_0_54, testTS_0_39, testTS_0_35, testTS_0_34, testTS_0_33, testTS_0_27}*

As expected, if the reduced suite is executed against the class and its mutants, it is seen that it also reaches the highest mutation score.

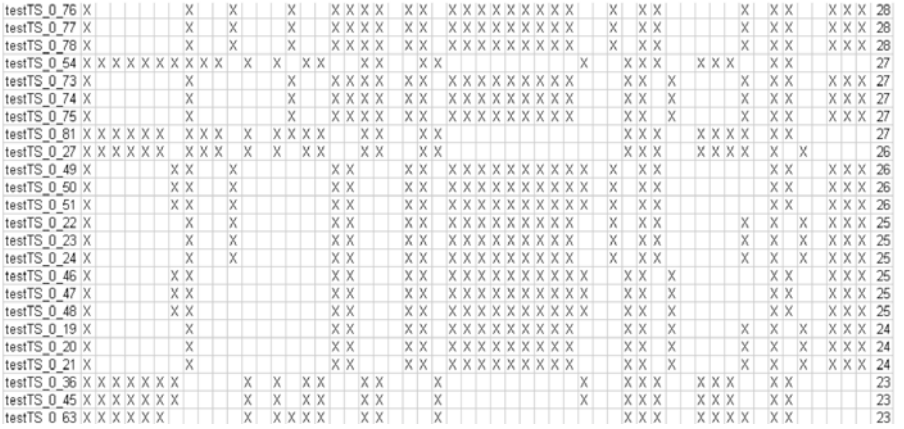


Fig. 6. A piece of the killing matrix for the original test suite, in the Jeffrey and Gupta example

4 Experiments

In order to check the results of the technique, two experiments were carried out:

The first experiment uses a set of programs which have been used in many research papers on software testing (for example, references [8, 11, 28-30]) and, therefore, can be considered as benchmarks. These programs are *Bisect, Bub, Find, Fourballs, Mid* and *TriTyp*. They are implemented as isolated Java classes. For this experiment, all the *traditional* mutation operators of the MuJava tool [31] were used to generate mutants (thus, those operators which are specific for the object-oriented characteristics of the Java language were excluded). Then, the *All combinations* strategy was used to generate test cases to kill these mutants. *All combinations* generates test cases by combining all test values of all the parameters [32].

For the second experiment, several classes were selected from a set of industrial systems. A required characteristic of the systems for this experiment is the availability of both its source code and test cases. In this case, the goal of the research is to evaluate the goodness of the technique in "actual" software. The test cases used for this experiment were those available with the corresponding project, and did not add new test cases to the test suites. All these projects are composed of several classes with different types of relationships among them. As with the first experiment, only the *traditional* operators were applied.

```
public class JGExample {
   float returnValue;
   public float f(float a, float b, float c, float d) {
        float x=0, y=0;
        if (a>0) x=2;
        else  x=5;
        if (b>0) y=1+x;
        if (c>0)
            if (d>0) returnValue=x;
            else returnValue=10;
        else returnValue=(1/(y-6));
        return returnValue;
   }

   public String toString() {
        return "" + returnValue;
   }
}
```

Fig. 7. The "motivational example" from Jeffrey and Gupta

4.1 Experiment 1: Benchmark Programs

Table 4 gives some quantitative information about the benchmark programs: number of lines of code (LOC), number of non-equivalent mutants generated by the MuJava tool (i.e., the equivalent mutants were manually removed), size of the original test suite (automatically generated using the *All combinations* strategy with the *testooj* tool) and size of the reduced suite. Of course, both the original and the reduced suite kill 100% of non-equivalent mutants and, thus, all of them are mutation-adequate.

Table 4. Results for the benchmark programs

Program	LOC	# of mutants	|Test suite|	|Reduced test suite|
Bisect	31	44	25	2 (8%)
Bub	54	70	256	1 (0,04%)
Find	79	179	135	1 (0.07%)
Fourballs	47	168	96	5 (5,2%)
Mid	59	138	125	5 (4%)
TriTyp	61	239	216	17 (7,8%)

4.2 Experiment 2: Industrial Programs

For this experiment, a set of publicly available programs was downloaded (Table 5 shows some quantitative data, measured with the Eclipse Metrics plugin). One important requirement for selecting these programs was the availability of test cases, since now the goal was to check whether among these test cases, written by the developers of the programs, there also exists any redundancy and that they can, therefore, be reduced.

In this way, the three following systems were selected and downloaded:

1) *jtopas,* a system for analyzing and parsing *HTML* pages with *CSS* code embedded. This system is included in the Software-artifact Infrastructure Repository (SIR), a website with a set of publicly available software, left by Do and others to facilitate benchmarking in testing experimentation [33].
2) *jester,* a testing tool for Java. This was developed by Ivan Moore and can be downloaded from *http://jester.sourceforge.net.*
3) *jfreechart,* a free chart library for the Java platform. It was designed for use in applications, applets, servlets and JSP.

Table 5. Quantitative data from the selected projects

Project	# of packages	# of classes	WMC (total)	LOC
jtopas	4	20	747	3,067
jester	7	67	588	3,225
jfreechart	69	877	20,584	124,664

Table 6 shows some quantitative data from the selected classes, all of them having a corresponding testing class. Table 7 shows:

1) The number of mutants generated for the class by the MuJava tool. Note that, in these projects, equivalent mutants have not been removed.
2) The number of available test cases for that class in the corresponding project.
3) The percentage of mutants killed by the original test suite.
4) The number of test cases in the reduced suite, once the original suite has been executed and the reduction algorithm has been applied.

Thus, for example, MuJava generates 94 mutants for *PluginTokenizer* from the *jtopas* project, where there are 14 test cases available on its website. These test cases kill 31% of the mutants. The last column shows the results of applying the test suite reduction algorithm, with the result that a single test case is sufficient for reaching the same coverage as the original 14 test cases.

Table 6. Quantitative data from the selected classes

Project	Program	LOC	WMC	Methods
jtopas	PluginTokenizer	157	27	16
jester	IgnoreList	26	8	3
jfreechart	CompositeTitle	63	14	8
	SimpleHistogramBin	117	32	11
	RendererUtilities	137	29	3
	XYPlot	2,470	591	194

Table 7. Test execution results

Program	Mutants	|Test suite|	Mutants killed	|Reduced test suite|
PluginTokenizer	94	14	31%	1 (7%)
IgnoreList	27	6	85%	2 (33%)
CompositeTitle	9	4	77%	1 (25%)
SimpleHistogramBin	293	5	66%	4 (80%)
RendererUtilities	801	6	81%	6 (100%)
XYPlot	3,012	21	36%	14 (66%)

5 Conclusions and Future Work

This article has presented a greedy algorithm to reduce the size of a test suite with no loss of quality, meaning that the new suite preserves the same coverage that the original suite reaches in the system under test. The testing criterion used to select the test cases is based on the percentage of mutants that each test case kills. The algorithm has been formally verified.

Moreover, it has been applied to both benchmark programs commonly used in testing research papers, as in industrial software, evidencing the utility of the algorithm in almost all cases.

In conclusion, the authors consider that mutation testing has reached sufficient maturity to be applied in the actual testing of real software. The research has now arrived at such a point that the knowledge produced over all these years is ready to be transferred to industrial testing tools.

As a future work, we plan to compare the presented reduction algorithm with the other presented in the literature, in order to determine differences of effectiveness and efficiency between them.

References

1. DeMillo, R., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. IEEE Computer 11(4), 34–41 (1978)
2. Barbosa, E.F., Maldonado, J.C., Vincenzi, A.M.R.: Toward the determination of sufficient mutant operators for C. Software Testing, Verification and Reliability 11(2), 113–136 (2001)
3. Offutt, A.J., Craft, W.: Using compiler optimization techniques to detect equivalent mutants. Software Testing, Verification and Reliability 7, 165–192 (1996)
4. Hamlet, R.: Testing programs with the help of a compiler. IEEE Transactions on Software Engineering 3(4), 279–290 (1977)
5. Mresa, E.S., Bottaci, L.: Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study. Software Testing, Verification and Reliability 9, 205–232 (1999)
6. Wong, W.E., Mathur, A.P.: Reducing the Cost of Mutation Testing: An Empirical Study. Journal of Systems and Software 31(3), 185–196 (1995)

7. Untch, R., Offutt, A., Harrold, M.: Mutation analysis using program schemata. In: International Symposium on Software Testing, and Analysis, Cambridge, Massachusetts, June 28-30, pp. 139–148 (1993)
8. Offutt, A.J., Lee, S.D.: An Empirical Evaluation of Weak Mutation. IEEE Transactions on Software Engineering 20(5), 337–344 (1994)
9. Reales, P., Polo, M., Offutt, J.: Mutation at System and Functional Levels. In: Third International Conference on Software Testing, Verification, and Validation Workshops, Paris, France, pp. 110–119 (April 2010)
10. Hirayama, M., Yamamoto, T., Okayasu, J., Mizuno, O., Kikuno, T.: Elimination of Crucial Faults by a New Selective Testing Method. In: International Symposium on Empirical Software Engineering (ISESE 2002), Nara, Japan, October 3-4, pp. 183–191 (2002)
11. Polo, M., Piattini, M., García-Rodríguez, I.: Decreasing the cost of mutation testing with 2-order mutants. Software Testing, Verification and Reliability 19(2), 111–131 (2008)
12. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Software Testing, Verification and Reliability 7(3), 165–192 (1997)
13. Baudry, B., Fleurey, F., Jézéquel, J.-M., Traon, Y.L.: Automatic test case optimization: a bacteriologic algorithm. IEEE Software 22(2), 76–82 (2005)
14. Kim, S.W., Clark, J.A., McDermid, J.A.: Investigating the effectiveness of object-oriented testing strategies using the mutation method. Software Testing, Verification and Reliability 11, 207–225 (2001)
15. Ammann, P., Offutt, J.: Introduction to software testing. Cambridge University Press (2008)
16. Polo, M., Piattini, M., Tendero, S.: Integrating techniques and tools for testing automation. Software Testing, Verification and Reliability 17(1), 3–39 (2007)
17. Jones, J.A., Harrold, M.J.: Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Transactions on Software Engineering 29(3), 195–209 (2003)
18. Garey, M.R., Johnson, D.S.: Computers and Intractability. W.H. Freeman, New York (1979)
19. Harrold, M., Gupta, R., Soffa, M.: A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 2(3), 270–285 (1993)
20. Jeffrey, D., Gupta, N.: Test suite reduction with selective redundancy. In: International Conference on Software Maintenance, Budapest, Hungary, pp. 549–558 (2005)
21. Tallam, S., Gupta, N.: A concept analysis inspired greedy algorithm for test suite minimization. In: 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 35–42 (2005)
22. Heimdahl, M., George, D.: Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In: 19th IEEE International Conference on Automated Software Engineering, pp. 176–185 (2004)
23. McMaster, S., Memon, A.: Call Stack Coverage for Test Suite Reduction. In: 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, pp. 539–548 (2005)
24. Runeson, P., Andersson, C., Höst, M.: Test processes in software product evolution -a qualitative survey on the state of practice. Journal of Software Maintenance and Evolution: Research and Practice 15(1), 41–59 (2003)
25. Geras, A.M., Smith, M.R., Miller, J.: A survey of software testing practices in Alberta. Canadian Journal of Electrical and Computer Engineering 29(3), 183–191 (2004)
26. Ng, S.P., Murnane, T., Reed, K., Grant, D., Chen, T.Y.: A Preliminary Survey on Software Testing Practices in Australia, Melbourne, Australia, pp. 116–125 (2004)

27. Polo, M., Reales, P.: Mutation Testing Cost Redution Techniques: A Survey. IEEE Software 27(3), 80–86 (2010)
28. Offutt, A.J., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology 5(2), 99–118 (1996)
29. Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-Data Generation Using Genetic Algorithms. Software Testing, Verification and Reliability 9(4), 263–282 (1999)
30. Offut, A.J., Pan, J., Zhang, T., Terwary, K.: Experiments with data flow and mutation testing. Report ISSE-TR-94-105 (1994)
31. Ma, Y.-S., Offutt, J., Kwon, Y.R.: MuJava: an automated class mutation system. Software Testing, Verification and Reliability 15(2), 97–133 (2005)
32. Grindal, M., Offutt, A.J., Andler, S.F.: Combination testing strategies: a survey. Software Testing, Verification and Reliability 15, 167–199 (2005)
33. H., D., Elbaum, S.G., Rothermel, G.: Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Empirical Software Engineering: An International Journal 10(4), 405–435 (2005)

Model-Based Filtering of Combinatorial Test Suites

Taha Triki[1], Yves Ledru[1], Lydie du Bousquet[1],
Frédéric Dadeau[2], and Julien Botella[3]

[1] UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble 2/CNRS,
LIG UMR 5217, F-38041, Grenoble, France
{Taha.Triki,Yves.Ledru,Lydie.du-Bousquet}@imag.fr
[2] LIFC - INRIA CASSIS Project, 16 route de Gray, 25030 Besançon, France
frederic.dadeau@lifc.univ-fcomte.fr
[3] Smartesting, Besançon, France
julien.botella@smartesting.com

Abstract. Tobias is a combinatorial test generation tool which can efficiently generate a large number of test cases by unfolding a test pattern and computing all combinations of parameters. In this paper, we first propose a model-based testing approach where Tobias test cases are first run on an executable UML/OCL specification. This animation of test cases on a model allows to filter out invalid test sequences produced by blind enumeration, typically the ones which violate the pre-conditions of operations, and to provide an oracle for the valid ones. We then introduce recent extensions of the Tobias tool which support an incremental unfolding and filtering process, and its associated toolset. This allows to address explosive test patterns featuring a large number of invalid test cases, and only a small number of valid ones. For instance, these new constructs could mandate test cases to satisfy a given predicate at some point or to follow a given behavior. The early detection of invalid test cases improves the calculation time of the whole generation and execution process, and helps fighting combinatorial explosion.

1 Introduction

Combinatorial testing is an efficient way to produce large test suites. In its basic form, combinatorial testing identifies sets of relevant values for each parameter of a function call, and the production of the test suite simply generates all combinations of the values of the parameters to instantiate the function call. JMLUnit [4] is a simple and efficient tool based on this technique, which uses JML assertions as the test oracle. Extended forms of combinatorial testing allow to sequence sets of operations, each operation being associated to a set of relevant parameters values. This produces more elaborate test cases, which are appropriate to test systems with internal memory whose behaviour depends on previous interactions. Tobias [19,17,18] is one of these combinatorial generators. It was used successfully on several case studies [3,9,10] and has inspired recent

J, de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 439–454, 2012.

combinatorial testing tools, such as the combinatorial facility of the Overture toolset for VDM++[16] or jSynoPSys [8].

Tobias takes as input a test pattern and performs its combinatorial unfolding into a possibly large set of test cases. Each test case usually corresponds to a sequence of test inputs. An additional oracle technology is needed to decide on successful or failed test executions. In the past, we have mainly used the run-time evaluation of JML assertions as a test oracle [3]. But the tool can be used in other contexts than Java/JML. In this paper, we adopt a model-based testing approach where tests are first played on a UML/OCL specification of the system under test. The animation of a sequence of operations on the UML/OCL specification is performed by the Test Designer tool of Smartesting [1] and brings two kinds of answers. First, it reports whether the sequence is valid, i.e. each of its calls satisfies the pre-condition of the corresponding operation and is able to produce an output which verifies the post-condition. Then, it provides the list of intermediate states and operation results after each operation call. This information can be used as test oracle to compare with the actual states and results of the system under test. It must be noted that the model is deterministic, which forces all accepted implementations to produce the same results and intermediate states (if observable). In summary, the model is used (a) to discard invalid sequences, and (b) to provide an oracle for valid ones.

Combinatorial testing naturally leads to combinatorial explosion. This is initially perceived as a strength of such tools: large numbers of tests are produced from a test pattern. This helps to systematically test a system by the exhaustive exploration of all combinations of selected values. The latest version of Tobias has been designed to generate up to 1 million abstract test cases. It is actually only limited by the size of the file system where the generated tests are stored. Unfortunately, the translation of these test cases into a target technology such as JUnit, the compilation of the resulting file and its execution usually require too much computing resources and, in practice, the size of the test suite must be limited between 10 000 and 100 000 test cases.

Several techniques can be adopted to limit combinatorial explosion. The most classical one is the use of pairwise testing techniques [5] which does not cover all combinations of parameter values but simply covers all pairs of parameter values. This technique is very efficient to reduce a large combinatorial test suite to a much smaller number of test cases, but it relies on the hypothesis that faults result from a combination of two parameters. Therefore it may miss faults resulting from a combination of three or more parameters. The technique can be generalized to cover all n-tuples of parameters but it may always miss combinations of $n + 1$ parameters. Another approach is the use of test suite reduction techniques [13] which select a subset of the test suite featuring the same code coverage as the original test suite. This technique has several limitations. First, it requires to play the full test suite in order to collect coverage information. Also, empirical studies have shown that test suite reduction may compromise fault detection [20].

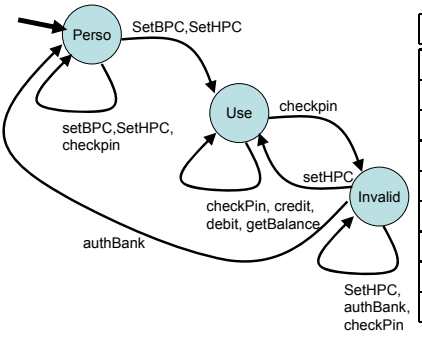
[1] http://smartesting.com

In this paper, we consider the case where combinatorial test suites lead to a large proportion of invalid test cases, i.e. test cases which will not be accepted by the specification. These invalid test cases must be discarded from the test suite because the specification is unable to provide an oracle for these tests. Discarding these invalid test cases leads to a safe reduction of the test suite. We present a tool which incrementally unfolds a test pattern and discards invalid test cases. The tool is based on an evolution of the Tobias tool where several new constructs have been added to the base language.

Section 2 introduces an illustrative case study. Then Section 3 presents the basic constructs of Tobias, using the case study. Section 4 presents additional constructs which help filter combinatorial test suites. Section 5 presents the toolset which incrementally unfolds the test patterns and filters the resulting test suite. Section 6 reports on several experiments carried with this tool set. Section 7 gives an overview of the research literature related to our work. Finally, Section 8 draws the conclusions of this work.

2 An Illustrative Case Study

We consider the example of a smart card application, representing an electronic purse (e-purse). This purse manages the balance of money stored in the purse, and two pin codes, one for the banker and one for the card holder. Similarly to smart cards, the e-purse has a life cycle (Fig. 1), starting with a *Personalization* phase, in which the values of the banker and holder pin codes are set. Then a *Use* phase makes it possible to perform standard operations such as holder authentication (by checking his pin), crediting, debiting, etc. When the holder fails to authenticate three consecutive times, the card is *invalidated*. Unblocking the card is done by a banker's authentication. Three successive failures in the bank authentication attempts make the card return to the *Personalization* phase. Each sequence of operations is performed within sessions, which are initiated through different terminals. This example has originally been designed to illustrate access control mechanisms, and it is used a basis for test generation



Method signature	Informal description
`beginSession(int)`	Opening of session
`endSession()`	Termination of session
`setBpc(int)`	Sets the bank's pin
`setHpc(int)`	Sets the holder's pin
`checkPin(int)`	Identifies the holder
`authBank(int)`	Identifies the bank
`credit(int)`	credit of the purse
`debit(int)`	debit of the purse
`getBalance()`	value of the balance

Fig. 1. The main modes of the bank card and the main operations

Pre-condition:

```
(self.isOpenSess_ = true  and  self.mode_ = Mode::USE and
self.terminal_ = Terminal::PDA  and  self.hptry_ > 0) = true
```

Post-condition:

```
if (pin = self.hpc_) then /**@AIM: HOLDER_AUTHENTICATED */
  self.isHoldAuth_ = true  and  self.hptry_ = self.MAX_TRY
else /**@AIM: HOLDER_IS_NOT_AUTHENTICATED */
  self.hptry_ = self.hptry_@pre - 1 and   self.isHoldAuth_ = false and
  if (self.hptry_ = 0) then /**@AIM: MAX_NUMBER_OF_TRIES_REACHED */
    self.mode_ = Mode::INVALID
  else /**@AIM: MAX_NUMBER_OF_TRIES_IS_NOT_REACHED */
    true
  endif
endif
```

Fig. 2. Pre and post-condition for `checkPin(int)` operation

for access control[2]. It was already used to illustrate test suite reduction with Tobias [7]. The original example was specified in JML. We have translated this specification into a UML/OCL model for the Smartesting Test Designer tool.

In Test Designer (TD), information about the behaviour of operations is captured in assertions associated to the operations. In the perspective of animation, these assertions must characterize a deterministic behaviour. An example of the pre- and post-conditions of the `checkPin(int)` operation is given Fig. 2. Post-conditions represent the code to be animated by TD if the pre-condition is verified. TD uses an imperative variant of OCL[3], inspired by the B language [1]. The variables appearing in the right hand side of a `=` sign are implicitly taken in their pre-state (usually denoted in OCL by `@pre`). In the model, the conditional branches are tagged with special comments. For example if the pin code is equal to the right one (`self.hpc_`), the tag `@AIM:HOLDER_AUTHENTICATED` will be activated and saved by the animator. After animation of an operation call, TD provides the list of all activated tags. The set of activated tags after an execution represents a behaviour of an operation. For example, the set:
B1 = {`@AIM:HOLDER_IS_NOT_AUTHENTICATED`,`@AIM:MAX_NUMBER_OF_TRIES_REACHED`}
is a behaviour of the checkPin operation leading to `INVALID` mode.

3 Basic Tobias Test Patterns

To generate test cases, Tobias unfolds a test pattern (also called "test schema"). The textual Tobias input language (TSLT) contains several types of constructs

[2] the original code of the application (in B and Java/JML) is available at `http://membres-liglab.imag.fr/haddad/exemple_site/index.html`

[3] The example presented in this paper follows the standard OCL syntax.

allowing the definition of complex system scenarios. The key concept in the Tobias input language is the **group** concept which defines a set of values or sequences of instructions. The group concept is subject to combinatorial unfolding. Some other concepts can be applied to instructions like iteration or choice. To illustrate these constructs, let us consider the following pattern:

```
group EPurseSchema1 [us=true, type=instruction] {
   @IUT;  @Personalize; @AuthenticateHolder; @Transaction{1,3};}
group IUT [type=instruction] { EPurse ep = new EPurse(); }
group Personalize [type=instruction] {
   ep.beginSession(Terminal.ADMIN);  ep.setBpc(@BankPinValue);
   ep.setHpc(@UserDebitValue);  ep.endSession(); }
group AuthenticateHolder{
   ep.beginSession(Terminal.PDA); ep.checkPin(@UserPinValue){1,4}; }
group Transaction [type=instruction] {
    (ep.credit(@Amounts) | ep.debit(@Amounts)); }
group BankPinValue [type=value] {values = [12,45];}
group UserPinValue [type=value] {values = [56,89];}
group Amounts [type=value] { values = [-1,0,50]; }
```

EPurseSchema1 is a group of instructions (`type = instruction`), and the flag **us** indicates whether the group will be unfolded (`=true`) or not. This group is a sequence of 4 other groups: IUT, Personalize, AuthenticateHolder and Transaction. This last group will be repeated one to three times in the sequence. The IUT group defines a new instance of class EPurse. Then, the Personalize group opens a new ADMIN session, sets the banker and the holder PIN codes, and finally closes the session. The AuthenticateHolder group authenticates the holder, and finally the Transaction group allows to do transactions. We use groups of values in some operation calls. For instance, the parameter of the `setBpc` method has 2 possible values.

The iteration construct $\{m,n\}$ repeats an instruction, or a sequence of instructions, from m to n times or exactly m times. For example, in group AuthenticateHolder, the `checkPin` operation is iterated 1 to 4 times (to check all possible sequences of correct/incorrect user authentication.

The Transaction group illustrates the choice construct. It consists of an exclusive choice between the two operation calls `Debit` or `Credit`. Each of them can be instantiated by three different amounts.

The `EPurseSchema1` pattern is unfolded into 30 960 test cases : $1 * (2*2) * (2^1+2^2+2^3+2^4) * ((3*2)^1+(3*2)^2+(3*2)^3)$. Only 2776 test cases are valid ones (i.e. satisfy the pre-conditions). In Fig. 3, examples of test cases unfolded from `EPurseSchema1` are given, `TC3` is valid, contrary to `TC26835` (which executes 4 consecutive calls to the checkPin operation with the wrong Pin code) and `TC30960` (which executes a debit operation but never credits).

If we put the maximum iteration bound of Transaction to 10, it would result into 8 707 129 200 test cases and would cause combinatorial explosion. In the next sections, we will see how the new Tobias constructs make it possible to take such explosive test patterns into account.

```
...
TC3: EPurse ep = new EPurse(); ep.beginSession(Terminal.ADMIN);
     ep.setBpc(12);  ep.setHpc(56); ep.endSession();
     ep.beginSession(PDA); ep.checkPin(56); ep.credit(50)
...
TC26835: EPurse ep = new EPurse(); ep.beginSession(ADMIN);
       ep.setBpc(45); ep.setHpc(89); ep.endSession();
       ep.beginSession(PDA); ep.checkPin(56); ep.checkPin(56);
       ep.checkPin(56); ep.checkPin(56); ep.credit(50)
...
TC30960: EPurse ep = new EPurse(); ep.beginSession(Terminal.ADMIN);
     ep.setBpc(45);  ep.setHpc(89); ep.endSession();
     ep.beginSession(PDA); ep.checkPin(89); ep.checkPin(89);
     ep.checkPin(89); ep.checkPin(89); ep.debit(50); ep.debit(50);
     ep.debit(50)
```

Fig. 3. Examples of test cases unfolded from `EPurseSchema1`

4 New Tobias Constructs

Here, we introduce three new constructs for the Tobias input language. These constructs support new techniques for filtering test cases. This allows to control the size of the produced test suite, and incrementally pilot the combinatorial unfolding process. These constructs are inspired by the jSynoPSys scenario language [8] and are syntactically and semantically adjusted to meet our needs.

The State predicate construct inserts an OCL predicate in the test sequence. The predicate expresses that a property is expected to hold at this stage of the test sequence. Tests whose animations do not satisfy this OCL predicate should be discarded from the test suite. It allows the tester to select a subset of the unfolded test suite featuring a given property at execution time. For example, not all AuthenticateHolder animations succeed. Therefore, we define a state predicate to select the tests which succeed the authentication. The pattern is defined as follows:

```
group EPurseSchema5 [us=true, type=instruction] {  @IUT; @Personalize;
@AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true); @Transaction; }
```

AuthenticateHolder performs checkPin one to four times. Then the pattern selects those sequences which end up with a successful authentication. The TSLT construct takes the form `⇝(set of targets , OCL predicate)`, where the set of targets identifies the objects which correspond to `self` in the OCL predicate. Here the set of targets is the singleton including `ep`.

The behaviours construct is another way to filter tests. It applies to an operation and keeps the tests whose animation covers a given behaviour, expressed as a set of tags. For example, in the previous pattern (`EPurseSchema5`), instead of

using a state predicate to keep tests that succeed the holder authentication, we select the authentication sequences whose last call to `checkPin` covers the tag

```
@AIM:HOLDER_AUTHENTICATED.
group EPurseSchema6 [us=true, type=instruction] {
@IUT; @Personalize; @AuthenticateHolder2; @Transaction; }

group AuthenticateHolder2 {
ep.beginSession(Terminal.PDA); ep.checkPin(@UserPinValue){0,3};
ep.checkPin(@UserPinValue)/w{set(@AIM:HOLDER_AUTHENTICATED)}; }
```

After the last call to checkPin, we put the symbol `/w` and we define a set of tags that must be activated after the operation execution. Here, when the pin code is correct, the tag `@AIM:HOLDER_AUTHENTICATED` is covered in the post-condition of `checkPin` (see Sect. 2).

The Filtering key is called after an instruction. It allows to accept a set of succeeded tests at some position and to discard the others. Then it will select some of the succeeded tests. TSLT provides four filtering keys (_ONE, _ALL, _n, _%n) to keep one, all, n or n% of the valid prologues. If we want to accept all of them we use _ALL, just one we use _ONE and _n (resp. _%n) randomly selects n (resp. n% of the) test cases amongst the valid ones. For example consider `EPurseSchema7`. The prologue group leads the purse to a state where the holder is authenticated. If the test engineer simply wants to keep one of these, he can add keyword `_ONE` after the prologue :

```
group EPurseSchema7 [us=true, type=instruction] {
@Prologue_ONE; @Transactions; }
group Prologue [us=true, type=instruction] {
@IUT;@Personalize; @AuthenticateHolder2; }
```

5 The Incremental Unfolding and Filtering Process

5.1 Standard Unfolding and Filtering Process

Before introducing the mechanism of incremental unfolding, we begin by presenting the process of generation, animation and filtering of test cases by coupling the Tobias and Test Designer tools (Fig. 4). The starting point is a schema file including a test pattern written in TSLT. Three steps are automatically involved to produce the test evaluation results:

1. The schema file is unfolded by the Tobias tool which generates one or several test suite files written in the XML output language of the tool (outob file). For each group marked in TSLT as `us=true`, Tobias produces an outob file. This file contains all abstract test cases generated by the combinatorial unfolding of the corresponding group.

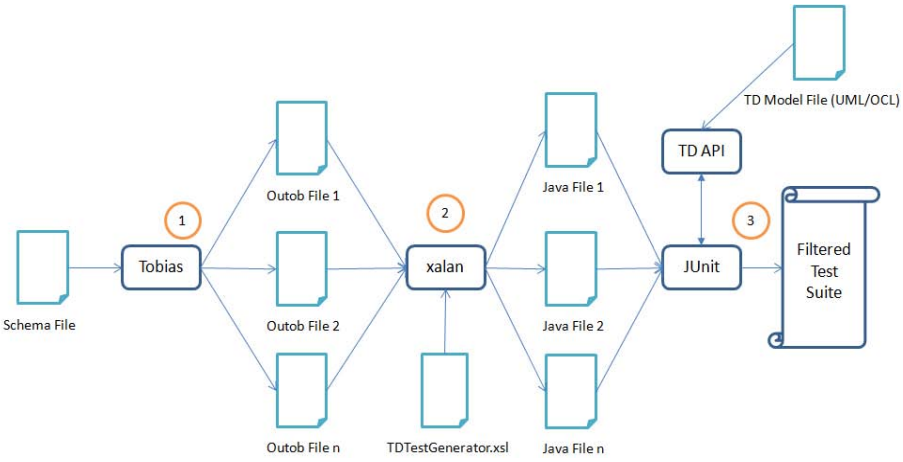


Fig. 4. The process of generation and filtering test cases (standard process)

2. The outob files are translated into JUnit test suites including all necessary information to animate test cases. Each JUnit test case interacts with the API of TD to animate the model. We take advantage of the JUnit framework and the Java API of TD to animate the tests in a popular and familar tool for engineers, and to benefit from the JUnit structure of test suites.
3. JUnit executes the test suites. Each test case is animated on the TD model through the TD API. The animation process allows to identify and filter out invalid test cases, i.e. the ones which:
   - include some operation call that violates its precondition,
   - include some operation call that violates its postcondition,
   - do not fulfill some state predicate or
   - include some operation call that fails to activate its associated behaviours.

   The animation of test cases proceeds sequentially. If an instruction fails because of one of these four reasons, the animation of the test stops and the test case is declared as failed and discarded from the test suite. The valid ones are saved to a repository.

5.2 Incremental Unfolding and Filtering Process

Algorithm The standard process requires to completely unfold the test patterns and to animate each test case of each test suite. At this stage, we did not take advantage of filtering keys (_ONE, _ALL, _n, _n%). These filtering keys can be applied on the resulting test suite to select the relevant test cases. In this section, we will see that the early application of filtering keys may lead to significant optimisations of (a) the unfolding process and (b) the animation of the test suite.

```
algorithm Incremental_Generation_And_Execution_Process (p):
  while( p contains at least one filtering key )
    Let (prefix _1stKey ; postfix) match p in
    validPrefixes := apply_Standard_Process(prefix);
    validPrefixesSubset := Select_Subset_Of_
                           Valid_Prefixes_According_To(1stKey);
    p := (validPrefixesSubset ; postfix);
  end while
  result := apply_Standard_Process(p);
end
```

Fig. 5. The incremental unfolding algorithm

The incremental process is defined for the unfolding of a single pattern p. It can be generalized to unfold multiple patterns. Its algorithm is given in Fig. 5 and performs the following steps:

- At each iteration, pattern p is divided into a prefix, located before the first filtering key, and a postfix, located after it.
- The standard unfolding and filtering process of Sect. 5.1 is applied to the prefix. It results into a group of valid unfolded prefixes.
- A subset of this group is selected according to the filtering key.
- This subset of valid unfolded prefixes is concatenated with the postfix to form the new value of p.
- The process iterates until all filtering keys are processed in the pattern.
- A last unfolding is applied to the resulting pattern stored in p.

Example. To illustrate this incremental process, consider the following pattern:

```
group EPurseSchema9 [us=true, type=instruction] { @IUT; @Personalize;
@AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true)_ONE; @Transactions;}
```

Before calling `@Transactions`, we would like to choose just one (`_ONE`) sequence of operations that succeeds holder authentication.
The prefix of this pattern is:

```
group EPurseSchema9pre [us=true, type=instruction] { @IUT; @Personalize;
@AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true); }
```

This prefix is then unfolded using the standard process. The three steps are executed to generate, animate and filter test cases. It unfolds into 120 tests, where only 56 are valid. A valid test is choosen randomly between them and inserted as a prefix in the new pattern:

```
group EPurseSchema9b [us=true, type=instruction] {
(ep.beginSession(ADMIN) ;  ep.setBpc(45) ;  ep.setHpc(56) ;
ep.endSession() ; ep.beginSession(PDA) ; ep.checkPin(89) ;
ep.checkPin(56) ; ep.checkPin(56) ;);   @Transactions; }
```

Since there is no remaining filtering key, the whole pattern will be unfolded to generate the final test cases. This unfolding leads to 6 test cases, where only 3 are valid. The final number of valid test cases may depend on the prefix that will be chosen randomly. These test cases will be animated to discard the invalid ones, and then produce the filtered test suite. This process is clearly optimized since only 126 test cases were completely unfolded, instead of 720 in the standard process. In the next section, we present experimental results on more complex examples.

6 Some Experimental Results

Let us consider the following example:

```
group EPurseExample [us=true, type=instruction] {@IUT; @Personalize;
@AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true);@Transactions{4};}
```

The `EPurseExample` is unfolded into 155 520 test cases. Our tools succeed to achieve steps 1 and 2 (translation into TSLT and production of an outob file). Unfortunately, the translation of the outob XML file into a JUnit file crashes due to a lack of memory (we used up to 1.5Gb of RAM). If this had succeeded, we presume that the compilation of the JUnit file would also crash. These technical problems can be overcome by decomposing our files into smaller ones, but still the whole process would take time and computing resources. Other group definitions can rapidly reach over 1 million test cases which may require untractable time and memory resources. We redefine the pattern by introducing filtering keys:

```
group EPurseExampleUsingKeys [us=true, type=instruction] {
@IUT; @Personalize; @AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true);
@Transactions_ALL; @Transactions_ALL; @Transactions_ALL; @Transactions; }
```

This pattern will produce the same valid test cases as the previous one, since we used the `_ALL` key. Using the incremental process, we need four iterations to remove the three filtering keys and unfold the resulting pattern. The pattern is completely unfolded and animated in 175 seconds as given in Fig. 6. As a result our 155 520 test cases only include 6496 valid ones. To identify these, our

Iteration	Nb of tests unfolded	Nb of tests accepted
1	720	168
2	1008	560
3	3360	1904
4	11424	6496

Fig. 6. Results of `EPurseExampleUsingKeys` unfolding

incremental process needs four iterations but only unfolds and plays 16512 test cases. In this case, it performed the selection process using 10% of the resources needed for the standard one, and kept the test suites small enough to avoid tool crashes.

Support for a Brute Force Approach. Let us consider another explosive pattern, based on Fig. 1. The aim of this pattern is to find test sequences where the purse goes back to Personalisation mode, before being set in Use mode. The only way to reach this goal is to start from *Perso* mode, go into *Use* and *Invalid* modes, before getting back to *Perso* and finally to *Use*. These major steps are captured in the state predicates of the following pattern:

```
group EPurseSchema18op [us=true, type=instruction] {
@IUT;
@ALLOps{4}⇝({ep}, self.mode_ = Mode::USE);
@ALLOps{5}⇝({ep}, self.mode_ = Mode::INVALID);
@ALLOps{5}⇝({ep}, self.mode_ = Mode::PERSO);
@ALLOps{4}⇝({ep}, self.mode_ = Mode::USE); }
```

In order to change states, we adopt a brute force approach where a single group has been defined for all operations offered by the card. Group `ALLOps` can be unfolded in 19 elements.

```
group ALLOps { ep.beginSession(@TerminalValue) | ep.endSession() |
ep.setBpc(@BankPinValue) | ep.setHpc(@UserDebitValue) |
ep.authBank(@BankPinValue) | ep.checkPin(@UserDebitValue) |
ep.credit(@Amounts) | ep.debit(@Amounts); }
```

EPurseSchema18op repeats all operations 4 times, until it reaches the *Use* mode. Finding that it requires 4 iterations can result from a trial and error process, or from a careful study of the specification. Since we adopt a brute force approach, let us consider that the engineer has attempted to reach the *Use* mode in one to three steps, without success, and finally found that four steps were sufficient (session opening, setting the Holder and Bank codes, and session close). Similarly he found that 5 steps are the minimum to reach state *Invalid* (session opening, three unsuccessful attempts to checkPin and session close), and to then reach state *Perso* (session opening, three unsuccessful attempts to authBank and session close). As a result, to find a valid sequence reaching the *Use* mode and returning to the same mode after visiting the other modes, we need to call at least 18 operations (4+5+5+4). `EPurseSchema18op` represents 19^{18} test cases (about 10^{23} test cases), and thus cannot be directly unfolded. Because of the brute force approach, and because we inserted filtering predicates, a large number of these test cases will be invalid. This enables us to call the incremental process.

We redefine `EPurseSchema18op` using the filtering key `_ALL` to keep all valid prefixes.

Iteration	Nb of tests unfolded	Nb of tests accepted	Iteration	Nb of tests unfolded	Nb of tests accepted
1	19	3	10	1064	72
2	57	7	11	1368	184
3	133	29	12	3496	376
4	551	24	13	7144	1160
5	456	40	14	22040	64
6	760	104	15	1216	80
7	1976	312	16	1520	224
8	5928	1136	17	4256	624
9	21584	56	18	11856	640

Fig. 7. Results of `EPurseSchema18opWFilteringKey` unfolding

```
group EPurseSchema18opWFilteringKey [us=true, type=instruction] {
@IUT;
@ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL;
@ALLOps⇝({ep}, self.mode_ = Mode::USE)_ALL;
@ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL;
@ALLOps⇝({ep}, self.mode_ = Mode::INVALID)_ALL;
@ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL;
@ALLOps⇝({ep}, self.mode_ = Mode::PERSO)_ALL;
@ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL;
@ALLOps⇝({ep}, self.mode_ = Mode::USE); }
```

`EPurseSchema18opWFilteringKey` is unfolded incrementally in 18 iterations. Fig. 7 shows the number of unfolded and accepted tests at each iteration. Steps 9 and 14 show how filtering predicates dramatically decrease the number of accepted tests. Fig. 7 shows that the number of test cases animated at each step remains small enough to be handled with reasonable time and computing resources, and to avoid tool crashes. As a result, we unfolded and animated a total of 85424 test cases for the 18 iterations in less than 17 minutes, instead of 19^{18} in the standard process. We finally found all 640 valid test cases hidden into this huge amount of potential test cases. This second example shows that this incremental technique is efficient to find complex test cases hidden in a huge search space. The key to success is to make sure that the use of filtering keys will effectively reduce or limit the number of test cases at each iteration. Therefore, one should prefer a specification and a test pattern which help identify invalid test cases as soon as possible.

Improvements with Respect to Our Previous Work. In the past [17,18], we have proposed two techniques to master combinatorial explosion with Tobias: test filtering at execution time, and test selection at generation time. Filtering at execution time is based on a simple idea: if the prefix of a test case fails, then all test cases sharing the same prefix will fail. In [17], we have proposed an intelligent test driver which remembers the failed prefixes, and avoids to execute a

test case starting with a prefix which previously failed. This idea is close to the one presented in this paper. Still, there are significant advances in the new technique proposed here. First, the original technique required to produce the full test suite. Every test was examined to check if it included a failing prefix. Our new incremental process does not generate the full test suite, it incrementally builds and filters the prefixes by alternating between unfolding and animation activities. Because we avoid the full unfolding of the test suite, we are able to consider test patterns corresponding to huge numbers of test cases (19^{18} in the last example). Another contribution of this paper is the definition of new constructs for test patterns (state predicates, behaviours, filtering keys), which help invalidate earlier the useless test cases in the unfolding process. Selection at generation time is another technique, where one selects a subset of the test suite based on some criterion. This selection takes place during the unfolding process and does not require to execute or animate test cases. In [18] we filtered the elements of the test suite whose text did not fulfill a given predicate. This predicate is freely chosen by the test engineer and does not prevent to filter out useful test cases. For example, one could filter out all test cases whose length was longer than a given threshold. In [7,18], we investigated the use of random selection techniques. These techniques are by essence unable to distinguish between valid and invalid test cases, but they are able to reduce the number of test cases to an arbitrary number whatever be the size of the initial test suite. Compared to these selection techniques, our incremental process does not discard valid test cases, but makes the assumption that the number of valid test cases is small enough to remain tractable.

7 Related Work

In [15], authors propose to study test reduction in the context of bounded-exhaustive testing, which could be described as a variation of combinatorial testing. Three techniques are proposed to reduce test generation, execution time and diagnosis. In particular, the *Sparse Test Generation* skips some tests at the execution to reduce the time to the first failing test. Unlike ours, this approach does not rely on a model to perform test generation and reduction. Our approach allows filtering large combinatorial test suites by animating them on an UML/OCL model. It eliminates tests which don't verify the pre and post conditions of operations and/or given predicates or states. Generating tests or simply checking their correctness, is a classical approach when a model is available (principle of the "model-based testing" approaches). For instance, in [12], authors generate automatically a combinatorial test suite, that satisfies both the specification and coverage criteria (among which pairwise coverage of parameter values). The generation engine is based on a constraint solver and the specification is expressed in Spec#. In [2], authors also propose to automate test case generation with a constraint solver, but the specification is expressed as contracts extended with state machines. For both, the objective of the work is to generate a test suite which fulfills a coverage criterion on a model. In our

approach, the test schema gives a supplementary selection criterion for test generation. In [6], specification is expressed as IOLTS and generation is done with respect to a test purpose, for conformance testing. In some way, our test schema can be compared to a sort of test purpose, but contrary to this work where only one test is generated for each test purpose, we aim at generating all the test cases satisfying the test purpose.

The problem of test suite reduction is to provide a shorter test suite while maintaining the fault detection power. The approach presented in [14] generates test suites from a model and traps properties corresponding to structural coverage criteria. An algorithm is then executed on the resulting test suite to generate a reduced test suite having the same coverage than the original one. The original and the reduced test suites are animated on a faulty model to compare their fault localization capabilities. In [11], authors propose an approach where test cases created thanks to model-checker are transformed such that redundancy within the test-suite is avoided, and the overall size is reduced. Our approach differs from test suite reduction techniques in two points. First, we don't need to execute all tests of the original test suite to perform the reduction, and second, we consider that all valid test cases are equivalent when performing reduction with the filtering keys. YETI[4] is a random test generation tool which generates test cases from program bytecode. The number of generated test cases is limited by the available time. The report shows in a real time GUI the bugs found sofar, the coverage percentage according to a classical coverage criteria, the number of system calls and the number of variables used in the system to carry out the test generation and execution. Compared to our toolset, our approach performs the generation of tests from a UML/OCL model and not from a program. The choice of the methods under test, the length of the call sequences and the parameter values is not done randomly as in the YETI tool but according to a careful test schema defined by the user. Contrary to our tool, the notion of filtering against specific states or behaviors and the incremental unfolding do not exist in the YETI tool whose purpose is to maximize bugs detection and structural coverage.

8 Conclusion and Perspectives

In this paper, we address the problem of filtering a large combinatorial test suite with respect to a UML/OCL Model. The whole approach relies on three main steps. First the set of tests to generate has to be *defined* in terms of a test pattern, expressed in a textual language called TSLT. Second, this schema is *unfolded* to produce abstract test cases that are *animated* within Smartesting Test Designer tool. This animation allows to identify and remove invalid test cases. The process of unfolding and filtering can be done incrementally so that potential combinatorial explosion can be mastered. Several examples have been presented. They show that the incremental process is able to generate all valid test cases scattered in a huge search space, provided that the number of

[4] Tool website: `http://www.yetitest.org/`

valid test cases remains small enough. This paper has presented several new constructs which help the test engineer to express more precise test patterns and to filter out invalid test cases at early stages of the unfolding process. From a methodological point of view, this requires to augment the test pattern with state predicates, behaviour selectors, and filtering keys, which keep the incremental process within acceptable bounds. This approach has been defined in the context of the ANR TASCCC Project. We intend to apply it soon to the Global Platform case study provided by Gemalto, a last-generation smart card operating system[5]. This model presents 3 billions of possible atomic instantiated operation calls, due to combination of operations parameters values. A large proportion of these latters represent erroneous situations that should not be considered.

Acknowledgment. This research is supported by the ANR TASCCC Project under grant ANR-09-SEGI-014.

References

1. Abrial, J.-R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (August 1996)
2. Belhaouari, H., Peschanski, F.: A Constraint Logic Programming Approach to Automated Testing. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 754–758. Springer, Heidelberg (2008)
3. du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., Lanet, J.L.: Reusing a JML specification dedicated to verification for testing, and vice-versa: case studies. Journal of Automated Reasoning 45(4) (2010)
4. Cheon, Y., Leavens, G.T.: A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002)
5. Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C.: The combinatorial design approach to automatic test generation. IEEE Softw. 13(5), 83–88 (1996)
6. Constant, C., Jeannet, B., Jéron, T.: Automatic Test Generation from Interprocedural Specifications. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 41–57. Springer, Heidelberg (2007)
7. Dadeau, F., Ledru, Y., Bousquet, L.D.: Directed random reduction of combinatorial test suites. In: Random Testing 2007, pp. 18–25. ACM (2007)
8. Dadeau, F., Tissot, R.: jSynoPSys – a scenario-based testing tool based on the symbolic animation of B machines. In: Finkbeiner, B., Gurevich, Y., Petrenko, A.K. (eds.) MBT 2009 Proceedings. ENTCS, vol. 253-2, pp. 117–132 (2009)
9. Dupuy-Chessa, S., du Bousquet, L., Bouchet, J., Ledru, Y.: Test of the ICARE Platform Fusion Mechanism. In: Gilroy, S.W., Harrison, M.D. (eds.) DSVIS 2005. LNCS, vol. 3941, pp. 102–113. Springer, Heidelberg (2006)
10. Ferro, L., Pierre, L., Ledru, Y., du Bousquet, L.: Generation of test programs for the assertion-based verification of TLM models. In: 3rd International Design and Test Workshop, IDT 2008, pp. 237–242. IEEE (December 2008)

[5] http://www.globalplatform.org/specifications.asp

11. Fraser, G., Wotawa, F.: Redundancy Based Test-Suite Reduction. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 291–305. Springer, Heidelberg (2007)
12. Grieskamp, W., Qu, X., Wei, X., Kicillof, N., Cohen, M.B.: Interaction Coverage Meets Path Coverage by SMT Constraint Solving. In: Núñez, M., Baker, P., Merayo, M.G. (eds.) TESTCOM/FATES 2009. LNCS, vol. 5826, pp. 97–112. Springer, Heidelberg (2009)
13. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. 2(3), 270–285 (1993)
14. Heimdahl, M., George, D.: On the effect of test-suite reduction on automatically generated model-based tests. Automated Software Engineering 14, 37–57 (2007)
15. Jagannath, V., Lee, Y.Y., Daniel, B., Marinov, D.: Reducing the Costs of Bounded-Exhaustive Testing. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 171–185. Springer, Heidelberg (2009)
16. Lausdahl, K., Lintrup, H.K.A., Larsen, P.G.: Connecting UML and VDM++ with Open Tool Support. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 563–578. Springer, Heidelberg (2009)
17. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS Combinatorial Test Suites. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 281–294. Springer, Heidelberg (2004)
18. Ledru, Y., Dadeau, F., du Bousquet, L., Ville, S., Rose, E.: Mastering combinatorial explosion with the Tobias-2 test generator. In: IEEE/ACM Int. Conf. on Automated Software Engineering, pp. 535–536. ACM (2007); demonstration
19. Maury, O., Ledru, Y., Bontron, P., du Bousquet, L.: Using Tobias for the automatic generation of VDM test cases. In: 3rd VDM Workshop (in Conjunction with FME 2002) (2002)
20. Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Int. Conf. on Software Maintenance, pp. 34–43. IEEE (1998)

A New Design Defects Classification: Marrying Detection and Correction

Rim Mahouachi[1,*], Marouane Kessentini[2], and Khaled Ghedira[1]

[1] SOIE, University of Tunis, Tunisia
rim.mahouachi@gmail.com, khaled.ghedira@isg.rnu.tn
[2] CS, Missouri University of Science and Technology, USA
marouanek@mst.edu

Abstract. Previous work classify design defects based on symptoms (long methods, large classes, long parameter lists, etc.), and treat separately detection and correction steps. This paper introduces a new classification of defects using correction possibilities. Thus, correcting different code fragments appending to specific defect category need, approximately, the same refactoring operations to apply. To this end, we use genetic programming to generate new form of classification rules combining detection and correction steps. We report the results of our validation using different open-source systems. Our proposal achieved high precision and recall correction scores.

Keywords: Software maintenance, refactoring, search-based software engineering, genetic programming, design defects.

1 Introduction

Software systems are evolving by adding new functions and modifying existing functionalities over time. Through this evolution process, software design becomes more complex. Thus, the understandability and maintainability are difficult and fastidious tasks. So, perfective maintenance [21], defined as software product modification after delivery to improve its performance, is an important maintenance activity. However, perfective maintenance is the most expensive activity in software development [21]. This high cost could potentially be greatly reduced by providing automatic or semi-automatic solutions to increase their understandability, adaptability and extensibility to avoid bad-practices. As a result, these practices have been studied by professionals and researchers alike with a special attention given to design-level problems, also known in the literature as defects, antipatterns [9], smells [21], or anomalies [15].

There has been much research devoted to the study these bad design practices [16, 3, 18, 7, 22]. 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. Refactoring is an effective technique to remove

* Corresponding author.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 455–470, 2012.

defects. It is defined as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [9]." In [9], several refactoring patterns are described. It is necessary to identify the refactoring candidates that contain "bad-smells" in order to apply refactoring patterns.

There has been much work on different techniques and tools for detecting and correcting defects, with over 300 publications [14]. However, there is no existing works on using refactoring solutions to classify defects. The vast majority of these works identify key symptoms that characterize a defect using combinations of mainly quantitative, structural, and/or lexical information. Thus, the different defects are classified based on their symptoms. Then, different possible standard refactoring solutions, for each defect category, are proposed. Completely different refactoring solutions can be used to correct the same defect type.

This paper presents a novel approach to classify defects using correction possibilities. Thus, correcting different code fragments appending to specific defect category need, approximately, the same refactoring operations to apply. Our approach is based on the use of defect examples generally available in defect repositories of software development companies. In fact, we translate regularities that can be found in such defect examples into detection-correction rules. To this end, we use genetic programming [5] to generate new form of classification rules combining detection and correction steps.

The primary contributions of the paper can be summarised as follows:

1. We introduce a new defects classification approach based on correction solutions. Our proposal does not require to define the different defect types, but only to have some refactoring examples; it does not require an expert to write detection or correction rules manually; it combines detection and correction steps; and each defect category has, approximately, the same refactoring operations to apply. However, different limitations are discussed in the discussion section.
2. We report the results of an evaluation of our approach; we used classes from six open source projects, as examples of badly-designed and corrected code. We used a 6-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining two systems as bases of examples. Almost all the identified classes were found, with a precision more than 70%.
3. We report the comparison results of our new defects classification and an existing work [16].We also report the results of a further comparison between genetic programming (GP) and another heuristic search algorithm [8].

The rest of this paper is organised as follows. Section 2 is dedicated to the problem statement, while Section 3 describes our approach details. Section 4 explains the experimental method, the results of which are discussed in Section 5. Section 6 introduces related work, and the paper concludes with Section 7.

2 Problem Statement

In this section, we emphasize the specific problems that are addressed by our approach.

Although there is a consensus that it is necessary to detect and fix design anomalies, our experience with industrial partners showed that there are many open issues that need to be addressed when defining a detection and correction tool. In the following, we introduce some of these open issues. Later, in section 5, we discuss these issues in more detail with respect to our approach.

In general, existing classification of defects are based on symptoms without taking in consideration the correction step. The two detection and correction steps are treated separately. Thus, each defect type could have different correction possibilities that are completely different: how to choose the good correction strategy?

In addition, different issues are related to defects classification based on symptoms:

- 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.
- The goal of identifying the type of defects is to help the maintainer in the correction step. However, with actual identification of defects based on symptoms only, large number of correction strategies can be proposed for the same defect type. Thus, the question is: how to choose the best refactoring solution?
- In the majority of situations, code quality can be improved without fixing maintainability defects. We need to identify if the code modification corrects some specific defects or not. In addition, the code quality is estimated using quality metrics but different problems are related to: how to determine the useful metrics for a given system and how to combine in the best way multiple metrics to detect or correct defects.
- The correction solutions should not be specific to only some defect types. In fact, specifying manually a "standard" refactoring solution for each maintainability defect can be a difficult task. In the majority of cases, these "standard" solutions can remove all symptoms for each defect. However, removing the symptoms does not mean that the defect is corrected.

In addition to these issues, the process of defining rules manually for detection or correction is complex, time-consuming and error-prone. Indeed, the list of all possible defect types or maintenance strategies can be very large [13] and each defect type requires specific rules.

3 Defects Classification Using Genetic Programming

This section shows how the above mentioned issues can be addressed using our proposal.

3.1 Overview

The general structure of our approach is introduced in Figure 1. The following two subsections give more details about our proposals.

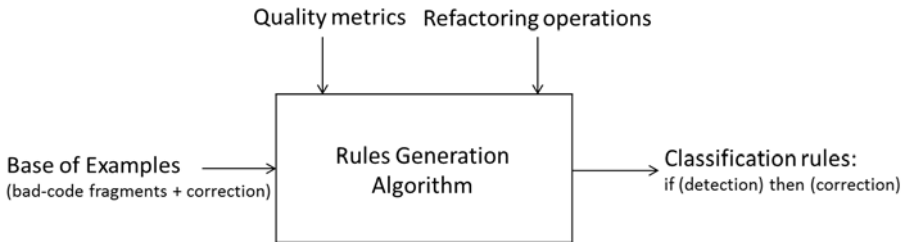


Fig. 1. Overview of the approach : general architecture

As described in Figure 1, knowledge from defect examples and their correction is used to generate our classification rules. In fact, our approach takes as inputs a base (i.e. a set) of defect examples (bad-designed code) with correction (refactorings to fix this bad designed code), and takes as controlling parameters a set of quality metrics (the expressions and the usefulness of these metrics were defined and discussed in the literature [15]) and an exhaustive list of refactoring operations [20]. This step generates a set of rules as output.

The rule generation process combines quality metrics (and their threshold values) and refactoring operations within rule expressions. Consequently, a solution to the defect detection and correction problem is a set of rules that best detect the defects of the base of examples and fix them. For example, the following rule states that a class having more than 10 attributes and 20 methods is fixed using different refactoring operations (Move method and Encapsulate field):

R1: IF NOA≥10 AND NOM≥20 Then MoveMethod ≥6 AND EncapsulateField<18

In this example of a rule, the number of attributes (NOA) and the number of methods (NOM) of a class correspond to two quality metrics that are used to detect a defect. A class will be detected as a defect whenever both thresholds of 10 attributes and 20 methods are exceeded. The second part of the rule fixes the corrected defect by applying more than 6 move method operations and less than 18 encapsulate field.

The rule generation process is executed periodically over large periods of time using the base of examples. The generated rules are used to detect and correct the defects of any system that is required to be evaluated (in the sense of defect detection and correction). The rule generation step needs to be re-executed only if the base of examples is updated with new defect instances.

Our approach assigns a threshold value randomly to each metric and refactoring operation, and combines these threshold values within logical expressions (union OR; intersection AND) to create rules. The number m of possible threshold values is usually very large. The rule generation process consists of finding the best combination between n metrics and k refactorings. In this context, the number NR of possible combinations that have to be explored is given by: $NR = ((n+k)!)^m$

This value quickly becomes huge. Consequently, the rule generation process is a combinatorial optimization problem. Due to the huge number of possible combinations, a deterministic search is not practical, and the use of a heuristic search is warranted. To explore the search space, we use a global heuristic search by means of Genetic Programming [5]. This algorithm will be detailed in the next section.

3.2 Design Defects Classification Using Genetic Programming

This section describes how Genetic programming (GP) can be used to generate rules to detect and correct design defects.

3.2.1 Genetic Programming Overview

Genetic programming is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution [1]. The basic idea is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a "good" solution of a specific problem.

In Genetic Programming, a solution is a (computer) program which is usually represented as a tree, where the internal nodes are functions and the leaf nodes are terminal symbols. Both the function set and the terminal set must contain symbols that are appropriate for the target problem. For instance, the function set can contain arithmetic operators, logic operators, mathematical functions, etc; whereas the terminal set can contain the variables (attributes) of the target problem. Each individual of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem.

Exploration of the search space is achieved by evolution of candidate solutions using selection and genetic operators, such as crossover and mutation. The selection operator insures selection of individuals in the current population proportionally to their fitness values, so that the fitter an individual is, the higher the probability that it is allowed to transmit its features to new individuals by undergoing crossover and/or mutation operators. The crossover operator insures generation of new children, or offspring, based on parent individuals. The crossover operator allows transmission of the features of the best fitted parent individuals to new individuals. This is usually achieved by replacing a randomly selected sub tree of one parent individual with a randomly chosen sub tree from another parent individual to obtain one child. A second child is obtained by inverting parents. Finally, mutation operator is applied, with a probability which is usually inversely proportional to its fitness value, to modify some randomly selected nodes in a single individual.

This process is repeated iteratively, until a termination criterion is met. This criterion usually corresponds to a fixed number of generations. The result of genetic programming (the best solution found) is the fittest individual produced along all generations.

3.2.2 Genetic Programming Adaptation

A high level view of our Genetic Programming approach to the defect detection and correction problem is introduced by Figure 2.

Lines 1–3 construct an initial GP population, which is a set of individuals that stand for possible solutions representing classification rules. Lines 4–14 encode the main GP loop, which explores the search space and constructs new individuals by combining metrics and refactorings within rules. During each iteration, we evaluate the quality of each individual in the population, and save the individual having the best fitness (line 10). We generate a new population (p+1) of individuals (line 11) by iteratively selecting pairs of parent individuals from population p and applying the crossover operator to them; each pair of parent individuals produces two children (new solutions). We include both the parent and child variants in the new population

p. Then we apply the mutation operator, with a probability score, for both parent and child to ensure the solution diversity; this produces the population for the next generation. The algorithm terminates when the termination criterion (maximum iteration number) is met, and returns the best set of classification rules (best solution found during all iterations). The following three subsections describe more precisely our adaption of GP to the defect classification problem.

```
Input: Set of quality metrics
Input: Set of refactoring operations
Input: Set of examples (bad-designed code fragments and their appropriate correction)
Output: Classification rules
1: I:= rules(R)
2: P:= set_of(I)
3: initial_population(P, Max_size)
4: repeat
5:          for all I ∈ P do
6:                    (detected_defects, proposed_refactorings) := execute_rules(R)
7:                    fitness(I) := compare(detected_defects,  defect_examples)    +
8:                              compare(proposed_refactorings, expected_refactorings)
9:          end for
10:         best_solution := best_fitness(I);
11:         P := generate_new_population(P)
12:         it:=it+1;
13: until it=max_it
14: return best_solution
```

Fig. 2. High-level pseudo-code for GP adaptation to our problem

3.2.2.1 Individual Representation. An individual is a set of IF – THEN rules. For example, Figure 3 shows the rule interpretation of an individual (NOA=Number of Attribute, and NOM=Number of Methods).

R1: IF (NOA ≥ 3 AND NOM ≤ 5) THEN (MoveField = 1 AND MoveClass = 1)
R2: IF (CBO≥ 1) THEN (MoveField = 1 AND ExtractClass = 1)

Fig. 3. Rule interpretation of an individual

Consequently, a detection rule has the following structure:

IF "Combination of metrics with their threshold values" THEN "Combination of Refactorings to apply"

The IF clause describes the conditions or situations under which a defect category is detected. These conditions correspond to logical expressions that combine some metrics and their threshold values using logic operators (AND, OR). If some of these

conditions are satisfied by a class, then this class is detected as a defect. Consequently, THEN clauses highlight the correction to apply in order to fix the detected defect. We will have as many rules as types of defects to be detected. In our case, mainly for illustrative reasons, and without loss of generality, we focus on the detection and correction of two defect types (categories). Consequently, as it is shown in Figure 4, we have two rules to detect and correct two categories of defects.

One of the most suitable computer representations of rules is based on the use of trees [17]. In our case, the rule interpretation of an individual will be handled by a tree representation which is composed of two types of nodes: terminals and functions. The terminals (leaf nodes of a tree) correspond to different quality metrics or refactorings with their threshold values. The functions that can be used between these metrics correspond to logical operators, which are Union (OR) and Intersection (AND).

Consequently, the rule interpretation of the individual of Figure 3 has the following tree representation of Figure 4.

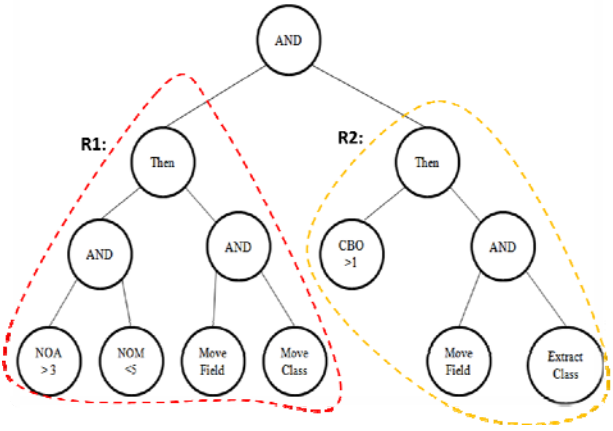


Fig. 4. A tree representation of an individual

3.2.2.2 Generation of an Initial Population. To generate an initial population, we start by defining the maximum tree length including the number of nodes and levels. These parameters can be specified either by the user or randomly. Thus, the individuals have different tree length (structure). Then, for each individual we randomly assign: (1) one metric or refactoring, with its threshold value, to each leaf node, and (2) a logic operator (AND, OR) to each function node.

The root (head) of the tree is unchanged. Since any metric combination is possible and correct semantically, we do need to define some semantic conditions to verify when generating an individual.

3.2.2.3 Operators

Selection

To select the individuals that will undergo the crossover and mutation operators, we used the stochastic universal sampling (SUS) [1], in which the probability of selection

of an individual is directly proportional to its relative fitness in the population. For each iteration, we use SUS to select (population_size/2) individuals from population p for the new populationp+1. These (population_size/2) selected individuals will "give birth" to another (population_size/2) new individuals using crossover operator.

Crossover

Two parent individuals are selected and a sub tree is picked on each one. Then crossover swaps the nodes and their relative sub trees from one parent to the other. Each child thus combines information from both parents.

Figure 5 shows an example of the crossover process. In fact, the rule R1 and a rule from another individual (solution) are combined to generate two new rules.

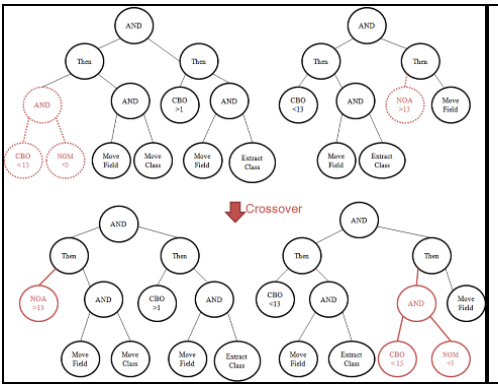


Fig. 5. Crossover operator

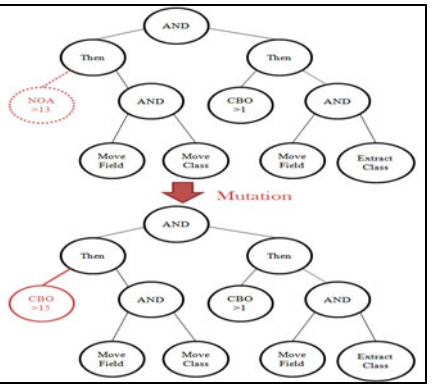


Fig. 6. Mutation operator

As result, after applying the cross operator the new rule R1 to detect blob will be:

R1: IF (NOA ≥ 13) THEN (MoveField = 1 AND MoveClass = 1)

Of course, the crossover can be applied to the second part of the rules (refactorings).

Mutation

The mutation operator can be applied either to function or terminal nodes. This operator can modify one or more nodes. Given a selected individual, the mutation operator first randomly selects a node in the tree representation of the individual. Then, if the selected node is a terminal (threshold value of a quality metric or refactoring), it is replaced by another terminal. If the selected node is a function (AND operator for example), it is replaced by a new function (i.e. AND becomes OR). If a tree mutation is to be carried out, the node and its sub trees are replaced by a new randomly generated sub tree.

To illustrate the mutation process, consider again the example that corresponds to a candidate rule R1. Figure 6 illustrates the effect of a mutation that replaces node NOA by node CBO with a new threshold value. Thus, after applying the mutation operator the new rule R1 will be: *R1: IF (CBO > 15) THEN (MoveField = 1 AND MoveClass = 1).*

3.2.2.4 Decoding of an Individual. The quality of an individual is proportional to the quality of the different rules composing it. In fact, the execution of these rules, on the different projects extracted from the base of examples, detect and fix various classes. Then, the quality of a solution (set of rules) is determined with respect to 1) the number of detected defects in comparison with the expected ones in the base of examples, and 2) the number of proposed refactorings with those in the base of examples. In other words, the best set of rules is the one that detects and corrects the maximum number of defects.

3.2.2.5 Evaluation of an Individual. The decoding of an individual should be formalized as a mathematical function called "fitness function". The fitness function quantifies the quality of the generated rules. The goal is to define an efficient and simple (in the sense not computationally expensive) fitness function in order to reduce the complexity.

As discussed previously, the fitness function aims to maximize the number of detected defects and proposed refactorings in comparison to the expected ones in the base of examples. In this context, we define the fitness function of a solution, normalized in the range [0, 1], as:

$$f_{norm} = \frac{\sum_{r=1}^{nbr} \frac{DQ + CQ}{2}}{nbr} \quad (3), \text{ where } DQ = \frac{\frac{\sum_{i=1}^{dd} a_i}{ed} + \frac{\sum_{i=1}^{dd} a_i}{dd}}{2} \text{ and } CQ = \frac{\frac{\sum_{j=1}^{pr} b_j}{er} + \frac{\sum_{j=1}^{pr} b_j}{pr}}{2}$$

nbr is the number of rules (defects categories) in the solution (individual), *DQ* represents detection quality, *CQ* is correction quality, *ed* is the number of expected defects in the base of examples, *dd* is the number of detected defects with defects, *pr* is the number of proposed refactoring and *er* is the number of expected refactoring in the base of examples.

a_i has value 1 if the i^{th} detected class exists in the base of examples, and value 0 otherwise. b_j has value 1 if the j^{th} proposed refactoring is used (exist in the base of examples) to correct the defect example.

4 Validation

To evaluate the feasibility of our approach, we conducted an experiment with different open-source projects. We start by presenting our research questions. Then, we describe and discuss the obtained results.

4.1 Research Questions

Our study asks three research questions, which we define here, explaining how our experiments are designed to address them. The goal of the study is to evaluate the efficiency of our approach for the detection and correction of maintainability 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 maintainability defects?
RQ2: To what extent our new defects classification is different from an existing classification work?
RQ3: To what extent can the proposed approach correct detected defects?

To answer RQ1, we asked eighteen graduate students to evaluate the precision and recall of our approach. To answer RQ2, we compared our results to those produced by an existing work [16]. Furthermore, we calculate a classification deviation score between the two algorithms. To answer RQ3, eighteen graduate students validated manually if the proposed corrections are useful to fix detected defects.

4.2 Setting

We used six open-source Java projects to perform our experiments: GanttProject (Gantt for short) v1.10.2, Quick UML v2001, AZUREUS v2.3.0.6, LOG4J v1.2.1, ArgoUML v0.19.8, and Xerces-J v2.7.0. Table 1 provides some relevant information about the programs. The base of examples contains more than 317 examples as presented in Table 1.

Table 1. Program statistics

Systems	Number of classes	KLOC	Number of Defects	Number of applied refactoring operations
GanttProject v1.10.2	245	31	41	34
Xerces-J v2.7.0	991	240	66	41
ArgoUML v0.19.8	1230	1160	89	82
Quick UML v2001	142	19	11	29
LOG4J v1.2.1	189	21	17	102
AZUREUS v2.3.0.6	1449	42	93	38

We used a 6-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining five systems as base of examples. For example, Xerces-J is analyzed using detection-correction rules generated from the base of examples containing ArgoUML, LOG4J, AZUREUS, Quick UML and Gantt.

The obtained detection results were compared to those of DECOR based on the recall (number of true positives over the number of maintainability defects) and the precision (ratio of true positives over the number detected). We also calculate a deviation score (number of common detected classes over the number of detected ones) between our new classification and DECOR classification (F.D.: Functional Decomposition, S.C.: Spaghetti Code, and S.A.K.: Swiss Army Knife).

To validate the correction step, eighteen graduate students verified manually the feasibility of the different proposed refactoring sequences for each system. We asked students to apply the proposed refactorings using ECLIPSE. Some semantic errors (programs behavior) were detected. When a semantic error is detected manually, we consider the refactoring operations related to this change as a bad recommendation. We calculate a correctness precision score (ratio of possible refactoring operations over the number of proposed refactoring) as performance indicator of our algorithm.

Finally, we compared our genetic algorithm (GP) detection and correction results with a local search algorithm, called simulated annealing (SA) [8].

4.3 Results

In this sub-section we present the answer to each research question in turn, indicating how the results answer each.

Table 2 shows obtained detection precision and recall scores for each of the 6 folds. Furthermore, this table describes comparison results between our defects classification and DECOR. For Gantt, our average detection precision was 89%. DECOR, on the other hand, had a combined precision of 59% for the same expected bad-classes. The precision for Quick UML was about 62%, more than the value of 53% obtained with DECOR. For Xerces-J, our precision average was 95%, while DECOR had a precision of 68% for the same dataset. Finally, for ArgoUML, AZUREUS and LOG4J our precision was more than 75. However, the recall score for the different systems was less than that of DECOR. In fact, the rules defined in DECOR are large and this is explained by the lower score in terms of precision. Indeed, the hypothesis to have 100% of recall justifies low precision, sometimes, to detect defects. The main reason that our approach finds better precision results is that the threshold values are well-defined using our genetic algorithm. Indeed, with DECOR the user should test different threshold values to find the best ones. Thus, it is a fastidious task to find the best threshold combination for all metrics.

In the context of this experiment, we can conclude that our technique was able to identify design anomalies, in average, more accurately than DECOR (answer to research question RQ1 above).

Table 2. Detection results

Systems	GP-Detection Precision	DECOR-Detection Precision	GP-Detection Recall	DECOR-Detection Recall	Number of Defect types (categories)
GanttProject	89% (33|37)	59% (41|69)	81% (33|41)	100%	6
Xerces-J	95% (47|49)	68% (66|97)	72% (47|66)	100%	7
ArgoUML	77% (74|96)	63% (89|143)	83% (74|89)	100%	6
Quick UML	62% (8|13)	53% (11|21)	73% (8|11)	100%	8
LOG4J	79% (15|19)	60% (17|28)	89% (15|17)	100%	6
AZUREUS	82%(87|106)	67% (93|138)	93% (87|93)	100%	7

For RQ2, we calculate, based on execution of our rules on ArgoUML, a similarity score between a well-known classification of defects [16] and our classification. This similarity score represents the number of common detected classes between a defect type *i* and our defect category *j*. In Table 3 we take the most similar category for each defect type. We can mention that only category 4 can be classified as a specific defect type which is the blob. Indeed, the explanation is that large classes has, in general, the same refactoring operations to be corrected (move methods, extract classes, etc.). Since our classification is based on correction criteria's and not symptoms, table 3 confirms our findings that there is a dissimilarity comparing to DECOR classification.

Table 3. Classification variation

Defect types	Most similar new category %
Blob	88 % (7|9, Category 4)
Functional Decomposition	36% (5|14, Category 1)
Spaghetti Code	54% (6|11, Category 6)
Swiss Army Knife	50% (8|16, Category 2)

Table 4. Correction results

Systems	GP-Correction Precision	GP-Correction Recall
GanttProject v1.10.2	84% (26|31)	76% (26|34)
Xerces-J v2.7.0	62% (29|47)	71% (29|41)
ArgoUML v0.19.8	75% (57|76)	67% (57|82)
Quick UML v2001	81% (21|26)	72% (21|29)
LOG4J v1.2.1	52% (63|123)	61% (63|102)
AZUREUS v2.3.0.6	74% (31|42)	81% (31|38)

We have also obtained very good results for the correction step. As showed in Table 4, the majority of proposed refactoring are feasible and improve the code quality. For example, for Gantt, 26 refactoring operations are feasible over the 31 proposed ones. After applying by hand the feasible refactoring operations for all systems, we evaluated manually that more than 75%, in average, of detected defects was fixed.

As described in Figure 7, we compared our genetic algorithm (GP) detection results with another local search algorithm, simulated annealing (SA). The detection and correction results for SA are also acceptable. Especially, with small systems the precision is better using SA than GP. In fact, GP is a global search that gives good results when the search space is large. For this reason, GP performs with large systems. In addition, the solution representation used in GP (tree) is suitable for rule generation. However, SA used a vector-based representation which is not suitable for rules. Furthermore, SA takes a lot of time, comparing to GP, to converge to an optimal solution (more than 1 hour).

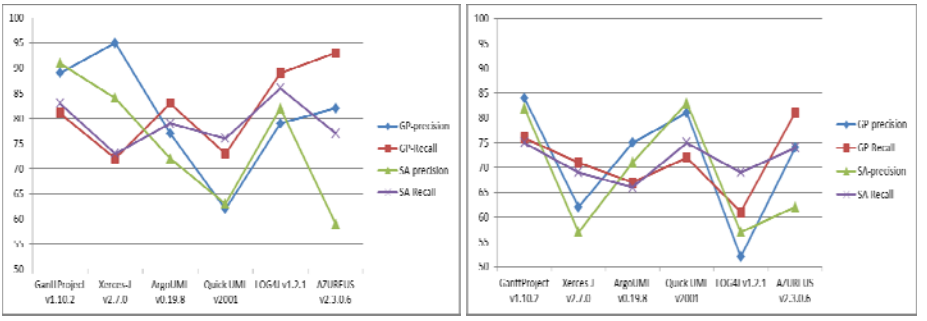


Fig. 7. (a) Detection results comparison: GP and SA, and (b) Correction results comparison: GP and SA

5 Discussions

An important consideration is the impact of the example base size on detection-correction quality. In general, our approach does not needs a large number of examples to obtain good detection results. The reliability of the proposed approach requires an example set of bad code and his correction (refactoring operations). 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 six open source projects directly, without any adaptation, the technique can be used out of the box and will produce good detection, correction results for the studied systems. However, we agree that, sometimes, with specifying programming languages and contexts it is difficult to find bad code fragments with the correction version.

The performance of detection was superior to that of DECOR. In an industrial setting, we could expect a company to start with some few open source projects, 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 rule generation process. The detection and correction results might vary depending on the rules used, which are randomly generated, though guided by a meta-heuristic. To ensure that our results are relatively stable, we compared the results of multiple executions for rule generation. We consequently believe that our technique is stable, since the precision and recall scores are approximately the same for different executions. In addition, our classification algorithm generates, approximately, the same defects categories as showed in Table .

Another important advantage in comparison to machine learning techniques is that our GP algorithm does not need both positive (good code) and negative (bad code) examples to generate rules like, for example, Inductive Logic Programming [4].

Finally, since we viewed the maintainability 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 (i7 CPU running at 3 GHz with 4GB of RAM). The execution time for rule generation with a number of iterations (stopping criteria) fixed to 10000 was less than fifty minutes for both detection and correction. This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the number of used metrics, refactorings and the size of the base of examples.

6 Related Work

There are several studies that have recently focused on detecting and fixing design defects in software using different techniques. These techniques range from fully automatic detection and correction to guided manual inspection. Nevertheless, few works focused on combining detection and correction steps to classify defects based on correction possibilties.

Marinescu [18] 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. [6] use

metrics to evaluate frameworks with the goal of improving them. They introduce the concept of multi-metrics, n-tuples of metrics expressing a quality criterion (e.g., modularity). The main limitation of the two previous contributions is the difficulty to manually define threshold values for metrics in the rules. To circumvent this problem, Alikacem et al. [2] express defect detection as fuzzy rules, with fuzzy labels for metrics, e.g., small, medium, large. When evaluating the rules, actual metric values are mapped to truth values for the labels by means of membership functions. Although no crisp thresholds need to be defined, still, it is not obvious to determine the membership functions. Moha et al. [16], in their DECOR approach, 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 which results in an important rate of false positives.The detection outputs are probabilities that a class is an occurrence of a defect type. In our approach, 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.

The majority of existing approaches to automate refactoring activities are based on rules that can be expressed as assertions (invariants, pre- and post condition), or graph transformation. The use of invariants has been proposed to detect parts of program that require refactoring by [22]. Opdyke [20] suggest the use of pre- and postcondition with invariants to preserve the behavior of the software. All these conditions could be expressed in the form of rules. [19] considers refactorings activities as graph production rules (programs expressed as graphs). However, a full specification of refactorings would require sometimes large number of rules. In addition, refactoring-rules sets have to be complete, consistent, non redundant, and correct. Furthermore, we need to find the best sequence of applying these refactoring rules. In such situations, search-based techniques represent a good alternative. In [8], we have proposed another approach, based on search-based techniques, 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. The two approaches are completely different. We use in [7] a good quality of examples in order to detect defects; however in this work we use defect examples to generate rules. Both works do not need a formal definition of defects to detect them. In another work [11], we generate detection rules defined as combinations of metrics/thresholds that better conform to known instances of design defects (defect examples). Then, the correction solutions, a combination of refactoring operations, should minimize, as much as possible, the number of defects detected using the detection rules. Our previous work treats the detection and correction as two different steps. In this paper, we generate also new form of detection-correction rules that are completely different from [11].

7 Conclusion

In this paper we introduced a new classification of defects based on correction criteria's. Existing work classify different types of common maintainability defects based on symptoms to search for in order to locate the maintainability defects in a

system. In this work, we have shown that this knowledge is not necessary to perform the detection and correction. Instead, we use examples of maintainability defects and their corrections to generate detection-correction classification rules. Our results show that our classification is able to achieve correction precision scores on different open-source projects.

As part of future work, we plan to compare our results with some existing approaches. In addition, we will modify our algorithm to generate sub-rules in order to specify code elements name (fully-automate the correction step). Furthermore, we need to extend our base of examples in order to improve precision scores.

References

1. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
2. Alikacem, H., Sahraoui, H.: Détection d'anomalies utilisant un langage de description de règle de qualité, in actes du 12e colloque LMO (2006)
3. 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)
4. Bratko, I., Muggleton, S.: Applications of inductive logic programming. Commun. ACM 38(11), 65–70 (1995)
5. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
6. Erni, K., Lewerentz, C.: Applying design metrics to object-oriented frameworks. In: Proc. IEEE Symp. Software Metrics. IEEE Computer Society Press (1996)
7. Kessentini, M., Vaucher, S., Sahraoui, H.: Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: Proc. of ASE 2010. IEEE (2010)
8. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by simulated annealing. Sciences 220(4598), 671–680 (1983)
9. Fowler, M.: Refactoring – Improving the Design of Existing Code, 1st edn. Addison-Wesley (June 1999)
10. Harman, M., Clark, J.A.: Metrics are fitness functions too. In: IEEE METRICS, pp. 58–69 (2004)
11. Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A.: Design Defects Detection and Correction by Example. In: Proc. ICPC 2011, pp. 81–90. IEEE (2011)
12. 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 (2003)
13. O'Keeffe, M., Cinnéide, M.: Search-based refactoring: an empirical study. Journal of Software Maintenance 20(5), 345–364 (2008)
14. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Trans. Softw. 30(2), 126–139 (2004)
15. Fenton, N., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. International Thomson Computer Press, London (1997)
16. Moha, N., Guéhéneuc, Y.-G., Duchien, L., Meu, A.-F.L.: DECOR: A method for the specification and detection of code and design smells. Transactions on Software Engineering (TSE), 16 pages (2009)

17. Davis, R., Buchanan, B., Shortcliffe, E.H.: Production Rules as a Representation for a Knowledge-base Consultation Program. Artificial Intelligence 8, 15–45 (1977)
18. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: Proc. of ICM 2004, pp. 350–359 (2004)
19. Heckel, R.: Algebraic graph transformations with application conditions. M.S. thesis, TU Berlin (1995)
20. Opdyke, W.F.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
21. 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 (March 1998)
22. Kataoka, Y., Ernst, M.D., Griswold, W.G., Notkin, D.: Automated support for program refactoring using invariants. In: Proc. Int'l Conf. Software Maintenance, pp. 736–743. IEEE Computer Society (2001)

Fine Slicing

Theory and Applications for Computation Extraction

Aharon Abadi*, Ran Ettinger, and Yishai A. Feldman

IBM Research – Haifa
{aharona,rane,yishai}@il.ibm.com

Abstract. Software evolution often requires the untangling of code. Particularly challenging and error-prone is the task of separating computations that are intertwined in a loop. The lack of automatic tools for such transformations complicates maintenance and hinders reuse. We present a theory and implementation of fine slicing, a method for computing executable program slices that can be finely tuned, and can be used to extract non-contiguous pieces of code and untangle loops. Unlike previous solutions, it supports temporal abstraction of series of values computed in a loop in the form of newly-created sequences. Fine slicing has proved useful in capturing meaningful subprograms and has enabled the creation of an advanced computation-extraction algorithm and its implementation in a prototype refactoring tool for Cobol and Java.

1 Introduction

Automated refactoring support is becoming common in many development environments. It improves programmer productivity by increasing both development speed as well as reliability. This is true in spite of various limitations and errors due to insufficiently detailed analysis. In a case study we performed [1], we recast a manual transformation scenario[1] as a series of 36 refactoring steps. We found that only 13 steps out of these 36 could be performed automatically by modern IDEs. Many of the unsupported cases were versions of the Extract Method refactoring, mostly involving non-contiguous code.

The example of Figure 1(a) shows the most difficult case we encountered. At this point in the scenario, we want to untangle the code that outputs the selected pictures to the HTML view (lines 1, 7, and 9) from the code that decides which pictures to present. The subprogram that consists only of these three lines does not even compile, because the variable `picture` is undefined. However, the more serious defect is that it does not preserve the meaning of the original program, since the loop is missing, and this program fragment seems to use only one picture. To preserve the semantics, the extracted subprogram needs to receive the picures to be shown in some collection, as shown in Figure 1(b). The rest of the code needs to create the collection of pictures and pass it to the new

* Also at Blavatnik School of Computer Science, Tel Aviv University.

[1] http://www.purpletech.com/articles/mvc/refactoring-to-mvc.html

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 471–485, 2012.

(a)

```
1 out.println("<table>");
2 int start = page * 20;
3 int end = start + 20;
4 end = Math.min(end, album.getPictures().size());
5 for (int i = start; i < end; i++) {
6     Picture picture = album.getPicture(i);
7     printPicture(out, picture);
8 }
9 out.println("</table>");
```

(b)

```
1 public void display(PrintStream out, int start,
2                     int end, Queue<Picture> pictures) {
3     out.println("<table>");
4     for (int i = start; i < end; i++)
5         printPicture(out, pictures.remove());
6     out.println("</table>");
7 }
```

(c)

```
1 int start = page * 20;
2 int end = start + 20;
3 end = Math.min(end, album.getPictures().size());
4 Queue<Picture> pictures = new LinkedList<Picture>();
5 for (int i = start; i < end; i++) {
6     Picture picture = album.getPicture(i);
7     pictures.add(picture);
8 }
9 display(out, start, end, pictures);
```

Fig. 1. (a) A program that tangles the logic of fetching pictures to be shown with their presentation. (b) Presentation extracted into a separate method. (c) Remaining code calls the new method.

method, as in Figure 1(c). This transformation is crucial in the scenario, as it forms the basis of the separation of layers. The code that deals with the HTML presentation is now encapsulated in the `display` method, and can easily be replaced by another type of presentation.

One possibility for specifying the subprogram to be extracted is just to select a part of the program, which need not necessarily be contiguous. In fact, the subprogram need not even contain complete statements; it is quite common to extract a piece of code replacing some expression by a parameter [1, Fig. 2]. However, in most cases the subprogram to be extracted is not some arbitrary piece of code, but has some inherent cohesiveness. In the example of Figure 1, the user wanted to extract the computations that write to the `out` stream, but without the computations of `start`, `end`, and `picture`.

This description is reminiscent of program slicing [17]. A (backward) slice of a variable in a program is a subprogram that computes that variable's value. The smallest such subprogram is of course desirable, although it is not computable

in general. However, a full slice is often too large. In our example, the slice of `out` at the end of the program in Figure 1(a) is the whole program, since all of it contributes to the output that will be written to `out`. In general, a slice needs to be closed under data and control dependences. Intuitively, one statement or expression is data-dependent on another if the latter computes a value used by the former. A statement is control-dependent on a test[2] if the test determines whether, or how many times, the statement will be executed.

In this paper we present the concept of *fine slicing*, a method that can produce executable and extractable slices while restricting their scopes and sizes in a way that enables fine control. This is done by allowing the user (or an application that uses fine slicing) to specify which data and control dependences to ignore when computing a slice. In particular, the subprogram we wanted to extract from Figure 1(a) can be specified as a fine slice of the variable `out` at the end of the program, ignoring data dependences on `start`, `end`, and `picture`.

Our fine-slicing algorithm will add to the slice control structures that are needed to retain its semantics, even when these control structures embody dependences that were specified to be ignored. For example, suppose that instead of line 1 in Figure 1(a) the following conditional appeared:

```
if (album.pictureSize() == SMALL)
    out.println("<table cellspacing='10'>");
else
    out.println("<table>");
```

The conditional will be added to the fine slice even if control dependences on it were specified to be ignored, since the subprogram that does not contain it will always execute both printing statements instead of exactly one of them. However, the data that the test depends on will not be included in the fine slice in that case. This part of the fine-slicing algorithm is called *semantic restoration*.

Fine slicing has many applications. For example, it can be used to make an arbitrary subprogram executable by adding the minimum necessary control structures. (This can be construed as a fine slice that starts from the given subprogram and ignores all dependences it has on other code.) In this paper we show in detail how fine slicing can be used in a generalization of Extract Method, which we call *Extract Computation*, that can handle non-contiguous code and other difficult transformations.

The contributions of this paper include:

- a theory of fine slicing, with an oracle-based semantics;
- an algorithm for fine slicing, including semantic restoration;
- a demonstration of the utility of fine slicing for the Extract Computation refactoring; and
- a prototype implementation of fine slicing and Extract Computation for Cobol and Java in Eclipse.

[2] In this context, we use the word "test" to refer to any conditional branch in the program's flow of control.

1.1 Fine Slicing

Slicing algorithms typically use some representation of the program with pre-computed data and control dependences. In order to compute a (backward) slice, the algorithm starts from an initial slice containing the user-selected locations (also called the *slicing criteria*). It then repeatedly adds to the slice any program location on which some part of the current slice has a data or control dependence. The final slice is available when the process converges. In the case of backward slices, the result is executable. (Forward slices are usually not executable.)

A fine slice can be computed in the same way, except that those dependences specified to be ignored are not followed. This, however, can result in a slice that has compilation errors, is not executable, or does not preserve the original semantics. This may be due to two types of problems: missing data, and missing control. Missing data manifests itself as the use of a variable one or more of whose sources (the assignments in the original program from which it receives its value) are unavailable. We consider the variable to have missing sources when it is disconnected from its sources in the original program, even if the subprogram appears to supply other sources for it.

Missing control creates control paths in the subprogram that are different from those in the original program, as in the case of the two table-header printing statements that would appear to be executed sequentially without the surrouding conditional.

We offer two different ways to deal with these problems. For missing data, we provide an oracle-based semantics, where the oracle supplies the missing values. In order to make the subprogram executable, the oracle can be simulated by appropriate variables or sequences. For missing control, our semantic restoration algorithm adds to the subprogram just those control structures that are necessary to make it preserve its original semantics. However, the data for these control structures is *not* added, being supplied by the oracle instead. These notions are formalized in Section 2.

While fine slicing can be used directly by a user using a tool that displays slices based on various criteria, we expect fine slicing to be used as a component by other applications, such as Extract Computation. In particular, we do not expect users to directly specify control dependences to be ignored. Data dependences are much easier for users to understand, and our graphical user interface for Extract Computation provides a convenient way to specify data dependences to be ignored.

1.2 Extract Computation

The example of Figure 1 shows the two types of difficulties involved in untangling computations. First, it is necessary to identify the relevant data sources as well as the control structures the subprogram to be extracted needs in order to preserve its semantics. This information can be used to generate the method encapsulating the extracted subprogram: the control structures are included in the new method, and the data sources are passed as parameters. This is achieved by the fine slicing algorithm.

Second, it is necessary to modify the original code; among other things, it needs to prepare any parameters and call the new method. As shown in the example, some parts of the original code (such as the loop) need to be duplicated. A *co-slice*, or complement slice [6], is the part of the program that should be left behind once a slice has been extracted from it. As shown above, the co-slice may contain some code that is also part of the extracted fine slice. It turns out that a co-slice is a special case of a fine slice, which starts from all locations not extracted and ignores data values to be returned. In the example, it is the slice from lines 2–4 and 6 of Figure 1(a), ignoring the final value of `out`.

The Extract Computation refactoring extracts the selected code and replaces it with an appropriate call. It computes the two fine slices and determines where the call to the extracted code should be placed. Some of the parameters need to aggregate values computed through a loop. The Extract Computation algorithm determines which data values are multiple-valued, and creates the code to generate the lists containing these values.

The Extract Computation refactoring is general enough to support all the cases in our case study [1] that were not supported by existing implementations of Extract Method. In particular, it can support the extraction of non-contiguous code in several varieties. In addition to the example above, demonstrating the extraction of part of a loop with the minimal required duplication of the loop, they include: extracting multiple fragments of code; extracting a partial fragment, where some expressions are not extracted but passed as parameters instead; and extracting code that has conditional exits (caused by `return` statements in the code to be extracted) [2]. A detailed description of the algorithm appears in Section 3.

2 A Theory of Fine Slicing

We assume a standard representation of programs, which consists of a control-flow graph (CFG), with (at least) the following relationships defined on it: domination and post-domination, data dependence and control dependence. We use $\mathrm{dflow}_P(d_1, d_2)$ to denote the fact that a variable definition d_1 reaches the use d_2 of the same variable in program P. We require that variable definitions include assignments to the variable as well as any operation that can modify the object it points to. In the example of Figure 1, any method applied to `out` can modify the output stream, and needs to be considered a definition of `out`. We assume a standard operational semantics, in which each state consists of a current location in the CFG of the program, and an environment that provides values of some of the program's variables. We also assume some mechanism that makes states in the same execution unique; for example, a counter of the number of times each node was visited.

We extend this representation to *open programs*, in which some variable uses can be marked as disconnected; these have no definitions reaching them. After defining the notion of a subprogram, we extend the operational semantics with oracles for disconnected variables, show how an oracle for an open subprogram is

induced from the corresponding execution of the original program, and formalize fine slices as open subprograms that, given the induced oracle, compute the same values for all variables of interest.

Definition 1 (subprogram). *A (possibly open) program Q is a* subprogram *of a program P if*

1. *all CFG nodes of Q belong to P;*
2. *for every variable definition point d and variable use point u in Q that is not disconnected,* $\mathrm{dflow}_Q(d,u)$ *is true iff* $\mathrm{dflow}_P(d,u)$ *is true; and*
3. *there is an edge from CFG node $n_1 \in Q$ to $n_2 \in Q$ iff there is a non-empty path from n_1 to n_2 in P that does not pass through other nodes in Q, except when n_1 is the exit node of Q and n_2 is the entry node of Q.*

We define a state s of P and a state s' of Q to be equivalent with respect to the connected variables (denoted $E(s,s')$) if their environments coincide on all common variables, except possibly for those that are disconnected in the current node. The initial states of Q will be restricted to those that are equivalent to states of P that can be reached by executions of P without visiting any nodes of Q before reaching those states.

An oracle $O(s,u)$ for an open program Q is a partial function that provides values for each disconnected variable u at each possible state s of the program. An execution of an open program is defined by extending the operational semantics of programs so that at any point where the value of a disconnected variable u is required in an execution state s, the value used will be $O(s,u)$. If this value is undefined, the execution is deemed to have failed.

The execution of P provides the oracle that can be used to supply the missing values in a corresponding execution of Q. Denote a single step in the operational semantics of P by $\mathrm{step}_P(s)$, and the value of a variable use u in state s by $\mathrm{env}_P(s,u)$.

Definition 2 (induced oracle). *The* oracle induced by a program P on an open subprogram Q of P from an initial state s_0 of P *is defined as follows: if* $\mathrm{step}_P^k(s_0) = s$ *for some $k \geq 0$, the current location in state s belongs to Q, u is a disconnected variable use in the current location, and $E(s,s')$, then* $\mathrm{oracle}_Q^{P,s_0}(s',u) = \mathrm{env}_P(s,u)$.

Under this definition, any open subprogram of P is executable with an oracle and preserves the behavior of P.

Theorem 1 (correctness of execution with oracle). *Let Q be an open subprogram of P, s_0 an initial state of P, and q_0 the corresponding initial state of Q (assuming one exists). If P halts*[3] *(i.e., reaches its exit node) when started at p_0, then Q will also halt when started at q_0 with the induced oracle,* $\mathrm{oracle}_Q^{P,s_0}$, *and will compute the same values for all common variable occurrences.*

[3] It is possible to relax this condition to specify that P reaches a state from which it cannot return to Q.

This theorem does not imply that an arbitrary sub-graph of the CFG of P is similarly executable and semantics-preserving, since the theorem only applies to subprograms (Def. 1), whose structure is constrained to preserve the data and control flow of P. The semantic restoration algorithm can be applied to any collection of CFG nodes, and will complete it into a subprogram by adding the required tests, even when control dependences on them were specified to be ignored. This is a crucial feature that makes fine slices executable and semantics-preserving. However, the data on which this control is based may still be disconnected. Therefore, adding these tests will not require the addition of potentially large parts of the program involved in the computation of the specific conditions used in these tests.

There are different ways to specify the dependences to be ignored by a particular fine slice, but ultimately these must be cast in terms of a set D of variable uses to be disconnected, and a set C of control dependences to be ignored (each represented as a pair (t, n) where a node n depend on a test t). As usual, the slice is started from a set of slicing criteria, which we represent as a set S of nodes in the CFG of P.

Definition 3 (fine slice). *Let P be a program, S a set of slicing criteria, D a set of variable uses to be disconnected in P, and C a set of control dependences from P to be ignored. A* fine slice of P with respect to S, D, and C *is an open subprogram Q of P that contains all nodes in S, and in which every disconnected variable use d satisfies at least one of the following conditions:*

1. *the variable use d was allowed to be disconnected: $d \in D$; or*
2. *d is variable use in a test node t on which all control dependences from elements in the slice are to be ignored: if $n \in Q$ is control-dependent on t then $(t, n) \in C$.*

We now present an algorithm that computes fine slices. The algorithm accepts as inputs a program P, a set S of slicing criteria, a set D of input variable uses that are allowed to remain disconnected in the fine slice, and a set C of control dependences that may be ignored.

The algorithm consists of the following main steps:

1. Compute the core slice Q by following data and control dependence relations backwards in P, starting from S. Traversal of data dependences does not continue from variable uses in D, and traversal of control dependences does not follow dependences that belong to C.
2. (Semantic Restoration) Add necessary tests to make the slice executable.
3. Connect each node $n_1 \in Q$ to a node $n_2 \in Q$ iff there is a path from n_1 to n_2 in P that does not pass through any other node in Q.

As explained above, in order to turn the fine slice into a subprogram, it is necessary to add some tests from the original program even though all their control dependences have been removed. This is the purpose of the semantic-restoration step, which comprises the following sub-steps:

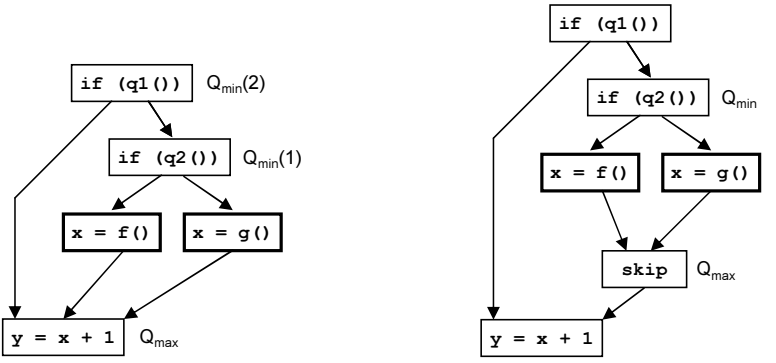


Fig. 2. Semantic restoration: original program (left); after separation of merges (right)

2.1 Let $Q_{\min}$ be the lowest node in P that dominates all the nodes of Q. Add $Q_{\min}$ to Q if it is not already there.
2.2 Let $Q_{\max}$ be the highest node in P that postdominates all the nodes of Q. Add $Q_{\max}$ to Q if it is not already there.
2.3 Add to Q all tests t from P that are on a path from some $q \in Q$ to $q' \in Q$ where q' is control-dependent on t, except when $q = Q_{\max}$ and $q' = Q_{\min}$. Do *not* add any data these tests depend on.
2.4 Repeat steps 2.1–2.3, taking into account the new nodes added each time, until there is no change.

In step 2.1, the "lowest" node is the one that is dominated by all other nodes in P that dominate all the nodes in Q. Similarly, in step 2.2 the "highest" node is the one that is postdominated by all other nodes in P that postdominate all nodes in Q. The smallest control context surrounding the fine slice is given by the part of the program between $Q_{\min}$ and $Q_{\max}$, and this determines the extraction context. This computation may need to be iterated for unstructured programs (including unstructured constructs in so-called structured languages); this is the purpose of step 2.4. When choosing $Q_{\max}$, the algorithm may need to add dummy nodes so as to separate merge points that are reachable from Q from those that are not. This will ensure that the new value of $Q_{\max}$ will not move $Q_{\min}$ to include unnecessary parts of the program. In the example of Figure 2, Q consists of the two assignments to `x` (with emphasized borders). On the left side of the figure, no separation of merges has been done. Initially, $Q_{\min}$ will be the second conditional (marked $Q_{\min}(1)$), and $Q_{\max}$ will be the assignment to `y`. This will force $Q_{\min}$ to move to the first conditional (marked $Q_{\min}(2)$), making Q contain the whole program. With the optimization of separating merges, shown on the right of the figure, $Q_{\max}$ is the new dummy node, and $Q_{\min}$ remains at the second conditional, making the fine slice smaller.

Step 2.3 excludes paths from $Q_{\max}$ to $Q_{\min}$, since these correspond to loops that contain the whole fine slice, and should not be included in it. When the fine slice is extracted, the call will appear inside any such loops.

Step 2.4 is only necessary for unstructured programs; with fully structured code a single iteration is always sufficient.

Theorem 2. *The fine slicing algorithm is well defined (i.e., nodes in steps 2.1 and 2.2 always exist), and produces a valid fine slice.*

Because the result of the algorithm is an open subprogram of P, it will compute the same values as P given the induced oracle (Theorem 1).

Theorem 3. *The worst-case time complexity of the fine-slicing algorithm is linear in the size of the program-dependence graph (i.e., the size of the control and data dependence relations).*

3 Extract Computation

The Extract Computation refactoring starts with a (possibly open) subprogram Q to be extracted from a program P. The subprogram can be a fine slice, chosen by the user or by another application; it can also be the result of applying semantic restoration to an arbitrary collection of statements. As a subprogram, it preserves the original semantics given the appropriate values from the induced oracle. Not all the code of Q can be removed from its original location, since some of it may be used for other purposes in P, as in the case of the common loop in Figure 1. The algorithm needs to compute the co-slice, replace the extracted code with an appropriate call, and implement the data-flow to the extracted code in the form of parameters and return values. Some of these values may be sequences, and the algorithm determines which they are and how to compute them.

The co-slice will contain all parts of the program that have not been extracted; it must also contain all the control and data elements required to preserve its semantics. However, any data it uses that is computed by the extracted program need not be part of the co-slice; instead, it can be returned by the extracted method. These values can therefore be disconnected, making the co-slice an instance of a fine slice. Many modern languages do not allow a function or method to return more than one value. When more than one value needs to be returned to the co-slice in such languages, they can be packed into a single object. Alternatively, it is possible to selectively disconnect only one such value, making the others be recomputed by the co-slice. Another inhibitor for many languages would be the necessity of passing sequences to the extracted code and back to the co-slice. This can only be done in languages (or frameworks) that support coroutines, since it requires intertwining of the computations of the co-slice and the extracted code.

Consider a disconnected variable use u in Q. In order to determine whether it requires a single value or a sequence, we need to know whether there is a loop in P but not in Q that contains the original source of u in P and its use in Q. If there is such a loop, a sequence is necessary. We define the source of u to be the CFG node in P at which the value to be used in u is uniquely determined.

This can be a definition of the same variable, but it can also be a join in the flow at which one of several definitions is chosen. For example, a variable `x` may be set to different values in two sides of a conditional; in this case, the source of a use of `x` following the conditional is the first node that joins the flow from both assignments. Formally, we define the source of u to be the node n such that: (1) n dominates the node of u; (2) the value of the variable does not change on every path from n to the first occurrence of the node of u; and (3) n dominates every other node that fulfills conditions (1) and (2). (This will put the source of u at the same point in which a ϕ-function will be generated in the Static Single Assignment form [5].)

Theorem 4. *Given a variable use u in Q, let G_Q be the smallest strongly-connected component of Q that contains the node of u. Each edge in Q corresponds to a set of paths in P; let G_P be the sub-graph of the CFG of P that contains all the nodes and edges in P that correspond to the edges of Q. If the source of u is not in G_P, then u has a single value in the induced oracle for Q in P.*

In the example of Figure 1, the extracted code has a disconnected input `picture` in the node for `printPicture`, which is contained in a single cycle. The source of this input in the full program is the `getPicture` node, which is on the same cycle. Theorem 4 does not apply, and therefore a sequence needs to be generated for it. In contrast, the use of `end` in the predicate `i < end` is on the same cycle, but its source is the node for `Math.min`, which is not on this cycle. Therefore the theorem applies, and no sequence is necessary.

Sequences can be implemented in various ways. For simplicity of the exposition we will consider a queue, but extensions to other data structures are trivial. For those parameters that are sequences, the algorithm needs to decide where to put the call that enqueues elements in the co-slice, and where to put the call that dequeues the elements in the extracted code. This is done by locating the unique place where the data passes into Q. This place is represented by a control edge in P whose target is in Q but whose source is not, such that all control paths from the source of u to the node of u itself pass through that edge. We call this edge the *injection point for u in Q*.

Theorem 5. *The injection point for every disconnected input of an open subprogram always exists and is unique.*

Consider now the variant P' of P in which an enqueue operation immediately followed by a dequeue operation is inserted at the injection point for u. This obviously does not change the behavior of P, since the queue is always empty except between the two new operations. We now define Q' to be the subprogram of P' that, in addition to Q itself, contains the dequeue operation, and in which the input of the dequeue operation is disconnected instead of u. When performing Extract Computation on P' and Q', the enqueue operation will belong to the co-slice, while the dequeue operation will be extracted. The behavior of the

resulting program will still be the same, since the same values are enqueued and dequeued as in P'; the only difference is that now all enqueue operations precede all dequeue operations.

In order to select a location in the co-slice in which to place the call to the extracted code, it is necessary to identify a control edge in the co-slice where the call will be spliced. Such an edge is uniquely defined by its source node c, which must satisfy the following conditions (in the context of P):

- c is contained in exactly the same control cycles as $Q_{\max}$;
- c must be dominated by all sources of parameters to the extracted code;
- every path from c to any of the added enqueue operations must pass through $Q_{\min}$; and
- c dominates each node containing any input data port that is disconnected in the co-slice (and therefore expects to get its value from the extracted code).

The first condition ensures that the call will be executed the same number of times as the extracted code was in the original program. The next two conditions ensure that all parameters will be ready before the call (since passing through $Q_{\min}$ initiates a new pass through the extracted code). There may be more than one legal place for the call, in which case any can be chosen; if there is no legal place, the transformation fails (this can happen when sequences need to be passed in both directions, as mentioned above). Note that the control successor of $Q_{\max}$ satisfies the first three conditions, and the call can always be placed there unless there are results to be returned from the extracted code to the co-slice. In the example, the only valid c is the exit node.

Given a subprogram and a set of expected results, the Extract Computation algorithm proceeds as follows: (1) extract the subprogram into a separate procedure; (2) identify parameters and create sequences as necessary; (3) replace the original code by the co-slice together with a call to the extracted procedure. The Extract Computation transformation is provably correct under the assumption that all potential data flow is represented by the data dependence relation. This is relatively easy to achieve for languages such as Cobol, but may not be the case in the presence of aliasing and sharing, as in Java. In all the cases we examined as part of our evaluation (Section 4.2) this has not been an issue.

4 Discussion

4.1 Implementation

We have implemented the Extract Computation and fine-slicing algorithms on top of our plan-based slicer [3]. They are therefore language-independent, and we are using them for transformations in Cobol as well as Java in Eclipse. In particular, the example of Figure 1 is supported by our tool for both languages.

For Extract Computation, our implementation uses an extension of the Eclipse highlighting mechanism, allowing the selection of non-contiguous blocks of text. Variables or expressions that are left unmarked indicate inputs to be disconnected. In addition, we disconnect all control dependences of marked code on

unmarked code. However, as mentioned above, semantic restoration will add control structures as necessary to maintain the semantics of the extracted code.

We are investigating other applications of fine slicing. For example, clone detectors identify similar pieces of code. The obvious next step is to extract them all into one method, taking their differences into account [9]. In this case, no user input is necessary, since the parameters of the fine-slicing algorithm, and in particular, which dependences should be ignored, can be determined based on the similarities between the clones in a way that will make the extracted part identical.

Another application of fine slicing is described in the next section.

4.2 Evaluation

We conducted an initial evaluation of fine slicing in the context of a prototype system that can automatically correct certain kinds of SQL injection security vulnerabilities [4] by replacing `Statement` by `PreparedStatement` objects. A vulnerable query is constructed as a string that contains user input, and should be replaced with a query that contains question-mark placeholders for the inputs; these are later inserted into the prepared statement via method calls that sanitize the inputs if necessary. However, sometimes the query is also used elsewhere; typically, it is written to a log file. The log file should contain the actual user input; in such cases, the proposed solution is to extract the part of the code that computes the query string into a separate method, which can be called once with the actual inputs, to contstuct the log string, and again with question marks, to construct the prepared statement.

In order to automate this process, we need to determine the precise part of the code that computes the query string, with all relevant tests. The test conditions, however, should not be extracted; they can be computed once and their values passed as parameters to the two calls. This describes a fine slice that starts at the string given to the query-execution method, and ignores all data dependences on non-string values and all control dependences.

In a survey of 52 real-world projects used to test a commercial product that discovers security vulnerabilities, we found over 300 examples of the construction of SQL queries. Most of these consisted of trivial straight-line code, but 46 cases involved non-trivial control flow. In these cases, we computed a full backward slice, a fine slice according to the criteria stated above, and a data-only slice. The fine slice was computed intra-procedurally, for soundness assuming that called methods may change any field. We compared these results to the code that should really be extracted, based on manual inspection of the code.

In 21 cases, the construction of the query contained conditional parts, where the condition was the result of a method call. In all these cases, the fine slice was the same as the full slice, except that it didn't contain the method call in the conditional. In terms of lines of code, the fine slice had the same size as the full slice, although it always contained the minimal part of code that needed to be extracted. In all these cases, the data slice was too small. In practical terms, if the condition is simple and quick to compute, has no side effects, and does

not depend on any additional data, a developer may include it in the extracted code. An automatic tool, such as the one we are developing, has no information about computational complexity, and so should by default choose the minimal code, which is the fine slice.

Twenty-five cases contained more interesting phenomena, where the fine slice was strictly smaller than the full slice even in terms of lines of code. The size of the full slice was between 5 and 23 lines, with an average of 13. The fine slices were between 1 and 14 lines, with an average of 6, and always coincided with the minimal part of the code that should be extracted. The data slices were sometimes larger and sometimes smaller than the fine slices, but were correct only in three cases.

As can be seen from these results, fine slicing has proved to be the correct tool for this application. Many other applications seem to require this technology, and we will continue this evaluation on other cases as well.

4.3 Related Work

Full slices are often too large to be useful in practice. The slicing literature includes a wide range of techniques that yield collections of program statements, including forward slicing [7], chopping [8], barrier slicing [12], and thin slicing [15]. These can be used for code exploration, program understanding, change impact analysis, and bugs localization, but they are not intended to be executable, and do not preserve the semantics of the selected fragment in the original program. In particular, they do not add the required control structures. We believe that all these techniques for finding interesting collections of statements could benefit from the added meaning given by semantic restoration, not only for use in program transformations, where executability with semantics preservation is a must, but also in assisting program understanding and related programming tasks such as debugging, testing, and verification.

Tucking [13] extracts the slice of an arbitrary selection of seed statements by focusing on some single-entry-single-exit region of the control flow graph that includes all the selected statements. They refer to this limited-scoped slice as a *wedge*. A tuck transformation adds to the identified region a call to the extracted wedge and removes from it statements that are not included in the full slice starting from all statements outside the wedge. Our computation of the co-slice by starting a fine slice from all nodes not in the extracted code is similar in this respect.

Using *block-based slicing*, Maruyama [14] extracts a slice associated with a single variable in the scope of a given block into a new method. The algorithm disconnects all data dependences on the chosen variable; this could lead to incorrect results in some cases. Tsantalis and Chatzigeorgiou [16] extended this work in several ways, including rejecting the transformation in such problematic cases. They still use the same framework of limiting the slice to a block. Fine slicing provides much finer control over slice boundaries.

A procedure-extraction algorithm by Komondoor and Horwitz [10] considers all permutations of selected and surrounding statements. Their following paper

[11] improves on that algorithm by reducing the complexity and allowing some duplication of conditionals and jumps but not of assignments or loops. Instead of backward slicing, this algorithm completes control structures but only some of the data. If a statement in a loop is selected, all the loop is added.

Sliding [6] computes the slice of selected variables from the end of a selected fragment of code, and composes the slice and its complement in a sequence. The complement can be thought of as a fine slice of all non-selected variables, ignoring all dependences of final uses of the selected variables. Our Extract Computation refactoring is more general by possibly ignoring other dependences, and by allowing more flexible placement of the slice. The concept of a final use of a variable can also help choosing which dependences to ignore when extracting a computation.

None of these approaches support passing sequences of values to what we call an oracle variable.

4.4 Future Work

This work is part of a long-term research project focusing on advanced enterprise refactoring tools, aiming to assist both in daily software development and in legacy modernization. The Extract Computation refactoring is a crucial building block in this endeavor. It will be used to enhance the automation for complex code-motion refactorings in order to support enterprise transformations such as the move to MVC [1,2]. As the prototype matures, it will be possible to evaluate to what extent such enterprise transformations can be automated.

We intend to make a number of improvements to the underlying analysis. Most important are interprocedural analysis and some form of pointer analysis. These will also support interprocedural transformations. The semantic restoration algorithm is useful on its own in order to make any subprogram executable. Based on our preliminary investigation we believe that an interprocedural extension of this algorithm is straightforward. It only requires pointer analysis in the presence of polymorphism, in order to compute the most accurate calling chain to be restored.

Acknowledgments. We are grateful to Mati Shomrat for his help with the implementation, and to Moti Nisenson for the name fine slicing.

References

1. Abadi, A., Ettinger, R., Feldman, Y.A.: Re-approaching the refactoring Rubicon. In: Second Workshop on Refactoring Tools (October 2008)
2. Abadi, A., Ettinger, R., Feldman, Y.A.: Fine slicing for advanced method extraction. In: Proc. Third Workshop on Refactoring Tools (October 2009)
3. Abadi, A., Ettinger, R., Feldman, Y.A.: Improving slice accuracy by compression of data and control flow paths. In: Proc. 7th Joint Mtg. European Software Engineering Conf. (ESEC) and ACM Symp. Foundations of Software Engineering (FSE) (August 2009)

4. Abadi, A., Feldman, Y.A., Shomrat, M.: Code-motion for API migration: Fixing SQL injection vulnerabilities in Java. In: Proc. Fourth Workshop on Refactoring Tools (May 2011)
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Programming Languages and Systems 13(4), 451–490 (1991)
6. Ettinger, R.: Refactoring via Program Slicing and Sliding. Ph.D. thesis, University of Oxford, Oxford, UK (2006)
7. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12(1), 26–60 (1990)
8. Jackson, D., Rollins, E.J.: A new model of program dependences for reverse engineering. In: Proc. 2nd ACM Symp. Foundations of Software Engineering (FSE), pp. 2–10 (1994)
9. Komondoor, R.: Automated Duplicated-Code Detection and Procedure Extraction. Ph.D. thesis, University of Wisconsin–Madison (2003)
10. Komondoor, R., Horwitz, S.: Semantics-preserving procedure extraction. In: Proc. 27th ACM Symp. on Principles of Programming Languages (POPL), pp. 155–169 (2000)
11. Komondoor, R., Horwitz, S.: Effective automatic procedure extraction. In: Proc. 11th Int'l Workshop on Program Comprehension (2003)
12. Krinke, J.: Barrier slicing and chopping. In: Proc. 3rd IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM) (September 2003)
13. Lakhotia, A., Deprez, J.C.: Restructuring programs by tucking statements into functions. In: Harman, M., Gallagher, K. (eds.) Special Issue on Program Slicing, Information and Software Technology, vol. 40, pp. 677–689. Elsevier (1998)
14. Maruyama, K.: Automated method-extraction refactoring by using block-based slicing. In: Proc. Symp. Software Reusability, pp. 31–40 (2001)
15. Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: Proc. Conf. Programming Lang. Design and Implementation (PLDI), pp. 112–122 (June 2007)
16. Tsantalis, N., Chatzigeorgiou, A.: Identification of Extract Method refactoring opportunities for the decomposition of methods. J. Systems and Software 84(10), 1757–1782 (2011)
17. Weiser, M.: Program slicing. IEEE Trans. Software Engineering SE-10(4) (1984)

System Dependence Graphs in Sequential Erlang*

Josep Silva, Salvador Tamarit, and César Tomás

Universitat Politècnica de València,
Camino de Vera s/n, E-46022 Valencia, Spain
{jsilva,stamarit,ctomas}@dsic.upv.es

Abstract. The system dependence graph (SDG) is a data structure used in the imperative paradigm for different static analysis, and particularly, for program slicing. Program slicing allows us to determine the part of a program (called slice) that influences a given variable of interest. Thanks to the SDG, we can produce precise slices for interprocedural programs. Unfortunately, the SDG cannot be used in the functional paradigm due to important features that are not considered in this formalism (e.g., pattern matching, higher-order, composite expressions, etc.). In this work we propose the first adaptation of the SDG to a functional language facing these problems. We take Erlang as the host language and we adapt the algorithms used to slice the SDG to produce precise slices of Erlang interprocedural programs. As a proof-of-concept, we have implemented a program slicer for Erlang based on our SDGs.

1 Introduction

Program slicing is a general technique of program analysis and transformation whose main aim is to extract the part of a program (the so-called *slice*) that influences or is influenced by a given point of interest (called *slicing criterion*) [18,15]. Program slicing can be dynamic (if we only consider one particular execution of the program) or static (if we consider all possible executions). While the dynamic version is based on a data structure representing the particular execution (a trace) [7,1], the static version has been traditionally based on a data structure called *program dependence graph* (PDG) [4] that represents all statements in a program with nodes and their control and data dependencies with edges. Once the PDG is computed, slicing is reduced to a graph reachability problem, and slices can be computed in linear time.

Unfortunately, the PDG is imprecise when we use it to slice interprocedural programs, and an improved version called *system dependence graph* (SDG) [6]

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 486–500, 2012.

```
(1)  main() ->
(2)    Sum = 0,
(3)    I = 1,
(4)    {Result,_} = while(Sum,I,11),
(5)    Result.

(6)  while(Sum,I,Top) ->
(7)    if
(8)     I /= Top -> {NSum,NI} = a(Sum,I),
(9)                 while(NSum,NI,Top-1)
(10)    I == Top -> {Sum,Top};
(11)   end.

(12) a(X,Y) ->
(13)  {add(X,Y),
(14)  fun(Z)->add(Z,1) end(Y)}.

(15) add(A,0) -> A;
(16) add(A,B) -> A+B.
```

Fig. 1. Original Program

```
(1) main() ->
(2)
(3)  I = 1,
(4)  {_,_} = while(undef,I,11).
(5)

(6) while(_,I,Top) ->
(7)    if
(8)     I /= Top -> {_,NI} = a(undef,I),
(9)                 while(undef,NI,Top)
(10)
(11)   end.

(12) a(_,Y)->
(13)   {undef,
(14)   fun(Z)->add(Z,1) end(Y)}.

(15)
(16) add(A,B) -> A+B.
```

Fig. 2. Sliced Program

has been defined. The SDG has the advantage that it records the calling context of each function call and can distinguish between different calls. This allows us to define algorithms that are more precise in the interprocedural case.

In this paper we adapt the SDG to the functional language Erlang. This adaptation is interesting because it is the first adaptation of the SDG to a functional language. Functional languages pose new difficulties in the SDG, and in the definition of algorithms to produce precise slices. For instance, Erlang does not contain loop commands such as `while`, `repeat` or `for`. All loops are made through recursion. In Erlang, variables can only be assigned once, and pattern matching is used to select one of the clauses of a given function. In addition, we can use higher-order functions and other syntactical constructs not present in imperative programs. All these features make the traditional SDG definition useless for Erlang, and a non-trivial redefinition is needed.

Example 1. The interprocedural Erlang program[1] of Figure 1 is an Erlang translation of an example in [6]. We take as the slicing criterion the expression `add(Z,1)` in line `(14)`. This means that we are interested in those parts of the code that might affect the value produced by the expression `add(Z,1)`. A precise slice w.r.t. this slicing criterion would discard lines `(2)`, `(5)`, `(10)` and `(15)`, and also replace some parameters by anonymous variables (represented by underscore), and some expressions by a representation of an undefined value (atom `undef`). This is exactly the result computed by the slicing algorithm described in this paper and shown in Figure 2. Note that the resulting program is still executable.

[1] We refer those readers non familiar with Erlang syntax to Section 3 where we provide a brief introduction to Erlang.

The structure of the paper is as follows. Section 2 presents the related work. Section 3 introduces some preliminaries. The Erlang Dependence Graph is introduced in Section 4, and the slicing algorithm is presented in Section 5. Finally, Section 6 presents some future work and concludes.

2 Related Work

Program slicing has been traditionally associated with the imperative paradigm. Practically all slicing-based techniques have been defined in the context of imperative programs and very few works exist for functional languages (notable exceptions are [5,13,12]). However, the SDG has been adapted to other paradigms such as the object-oriented paradigm [8,9,17] or the aspect-oriented paradigm [21].

There have been previous attempts to define a PDG-like data structure for functional languages. The first attempt to adapt the PDG to the functional paradigm was [14] where they introduced the *functional dependence graph* (FDG). Unfortunately, the FDGs are useful at a high abstraction level (i.e., they can slice modules or functions), but they cannot slice expressions and thus they are insufficient for Erlang. Another approach is based on the *term dependence graphs* (TDG) [3]. However, these graphs only consider term rewriting systems with function calls and data constructors (i.e., no complex structures such as if-expressions, case-expressions, etc. are considered). Moreover, they are not able to work with higher-order programs. Finally, another use of program slicing has been done in [2] for Haskell. But in this case, no new data structure was defined and the abstract syntax tree of Haskell was used with extra annotations about data dependencies.

In [19,20] the authors propose a flow graph for the sequential component of Erlang programs. This graph has been used for testing, because it allows us to determine a set of different flow paths that test cases should cover. Unfortunately, this graph is not based on the SDG and it does not contain the information needed to perform precise program slicing. For instance, it does not contain summary edges, and it does not decompose expressions, thus in some cases it is not possible to select single variables as the slicing criterion. However, this graph solve the problem of flow dependence and thus it is subsumed by our graphs. Another related approach is based on the *behavior dependency graphs* (BDG) [16] that has been also defined for Erlang. Even though the BDG is able to handle pattern matching, composite expressions and all constructs present in Erlang, it has the same problem as previous approaches: the lack of the summary edges [6] used in the SDG implies a loss of precision.

All these works have been designed for intra-procedural slicing, but they lose precision in the inter-procedural case. This problem can be solved with the use of a SDG. From the best of our knowledge, this is the first adaptation of the SDG to a functional language.

3 Preliminaries

In this section we introduce some preliminary definitions used in the rest of the paper. For the sake of concreteness, we will consider the following subset of the Erlang language:

pr	::=	$\overline{f_n}$	(Program)
f	::=	$\overline{atom\ fc_n}$	(Function Definition)
fc	::=	$(\overline{p_m}) \rightarrow \overline{e_n} \mid (\overline{p_m})$ `when` $\overline{g_o} \rightarrow \overline{e_n}$	(Function Clause)
p	::=	$l \mid V \mid \langle\overline{p_n}\rangle \mid [\overline{p_n}] \mid p_1 = p_2$	(Pattern)
g	::=	$l \mid V \mid \langle\overline{g_n}\rangle \mid [\overline{g_n}] \mid g_1\ op\ g_2 \mid op\ g$	(Guard)
e	::=	$l \mid V \mid \langle\overline{e_n}\rangle \mid [\overline{e_n}] \mid$ `begin` $\overline{e_n}$ `end`	(Expression)
	$\mid$	$e_1\ op\ e_2 \mid op\ e \mid e(\overline{e_n}) \mid p = e$	
	$\mid$	$[e \mid\mid \overline{gf_n}] \mid$ `if` $\overline{ic_n}$ `end` $\mid$ `case` e `of` $\overline{cc_n}$ `end`	
	$\mid$	`fun` $atom/number \mid$ `fun` $\overline{fc_n}$ `end`	
l	::=	$number \mid string \mid atom$	(Literal)
gf	::=	$p \leftarrow e \mid e$	(Generator | Filter)
ic	::=	$\overline{g_m} \rightarrow \overline{e_n}$	(If Clause)
cc	::=	$p \rightarrow \overline{e_n} \mid p$ `when` $\overline{g_m} \rightarrow \overline{e_n}$	(Case Clause)
op	::=	$+ \mid - \mid * \mid / \mid div \mid rem \mid ++ \mid --$	(Operation)
	$\mid$	$not \mid and \mid or \mid xor \mid == \mid /=$	
	$\mid$	$=< \mid < \mid >= \mid > \mid =:= \mid =/=$	

An Erlang program is a collection of function definitions. Note that we use the notation $\overline{f_n}$ to represent the sequence $f_1 \ldots f_n$. Each function definition is formed in turn by a sequence of n pairs $atom\ fc$ where $atom$ is the name of the function with arity n and fc is a function clause. Function clauses are formed by a sequence of patterns enclosed in parentheses followed optionally by a sequence of guards, and then an arrow and a sequence of expressions (e.g., $f(X,\ Y,\ Z)$ `when` $X > 0; Y > 1; Y < 5\ \rightarrow\ X+Y, Z.$). A pattern can be a literal (a number, a string, or an atom), a variable, a compound pattern or a tuple or list of other patterns. Guards are similar to patterns, but they must evaluate to represent a boolean value, and they do not allow compound patterns. Expressions can be literals, variables, tuples, lists, blocks composed of sequences of expressions, operations, applications, pattern matching, list comprehensions, if-expressions and case-expressions, function identifiers and declarations of anonymous functions, which are formed by a sequence of function clauses as in function definitions. In Erlang, when a call to a function is evaluated, the compiler tries to do pattern matching with the first clause of the associated function definition and it continues with the others until one succeeds. When pattern matching succeeds with a clause then its body is evaluated and the rest of clauses are ignored. If no clause succeeds then an error is raised.

In the following we will assume that each syntactic construct of a program (e.g., patterns, guards, expressions, etc.) can be identified with *program positions*. Program positions are used to uniquely identify each element of a program.

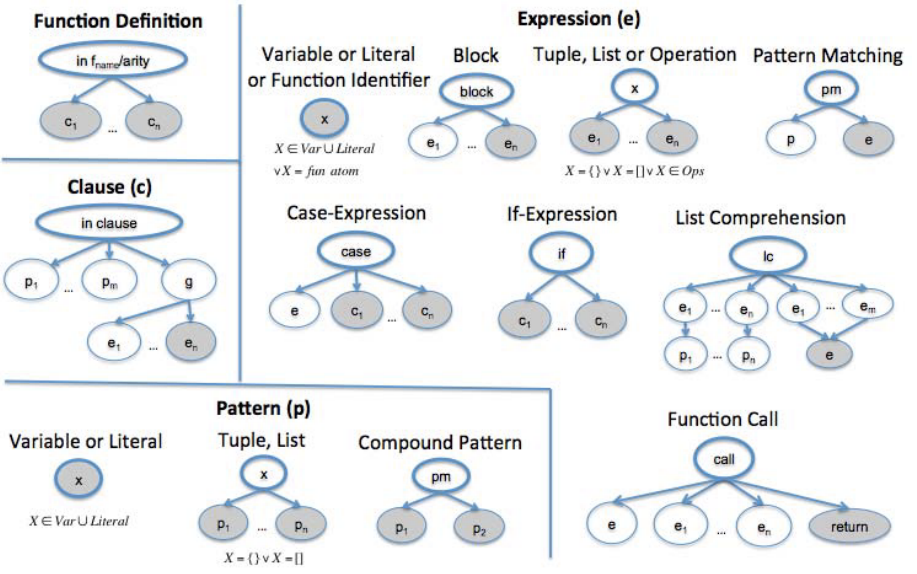


Fig. 3. Graph representation of Erlang programs

In particular, the program position of an element identifies the row and column where it starts, and the row and column where it ends. We also assume the existence of a function *elem* that returns the element associated to a given program position. Additionally, we use the finite sets *Vars*, *Literal*, *Ops* and $\mathcal{P}$ that respectively contain all variables, literals, operators and positions in a program.

4 Erlang Dependence Graphs

In this section we adapt the SDG to Erlang. We call this adaptation Erlang dependence graph. Its definition is based on a graph representation of the components of a program that is depicted in Figure 3.

Figure 3 is divided into four sections: function definitions, clauses, expressions and patterns. Each graph in the figure represents a syntactical construct, and they all can be composited to build complex syntactical definitions. The composition is done by replacing some nodes by a particular graph. In particular, nodes labeled with `c` must be replaced by a clause graph. Nodes labeled with `e` must be replaced by one of the graphs representing expressions or function definitions (for anonymous functions). And nodes labeled with `p` must be replaced by one of the graphs representing patterns. In order to replace one node by a graph, we connect all input arcs of the node to the initial node of the graph that is represented with a bold line; and we connect all output arcs of the node to the final nodes of the graph that are the dark nodes. Note that in the case that a final node is replaced by a graph, then the final nodes become recursively the dark nodes of this graph. We explain each graph separately:

Function Definition: The initial node includes information about the function name and its arity. The value of f_{name} is $\perp$ for all anonymous functions. Clauses are represented with $\mathtt{c_1} \ldots \mathtt{c_n}$ and there must be at least one.

Clause: They are used by functions and by case- and if-expressions. In function clauses each clause contains zero or more patterns ($\mathtt{p_1} \ldots \mathtt{p_m}$) that represent the arguments of the function. In case-expressions each clause contains exactly one pattern and in if-expressions no pattern exists. Node `g` represents all the (zero or more) guards in the clause. If the clause does not have guards it contains the empty list []. There is one graph for each expression ($\mathtt{e_1} \ldots \mathtt{e_n}$) in the body of the clause.

Variable/Literal: They can be used either as patterns or as expressions, and they are represented by a single (both initial and final) node.

Function Identifier: It is used for higher order calls. It identifies a function with its name and its arity and it is represented by a single (both initial and final) node.

Pattern Matching/Compound Pattern: It can be used either as a pattern or as an expression. The only difference is that if it is a pattern, then the final nodes of both subpatterns are the final nodes. In contrast, if it is an expression, then only the final nodes of the subexpression are the final nodes.

Block: It contains a number of expressions ($\mathtt{e_1} \ldots \mathtt{e_n}$), being the final nodes the last nodes of the last expression ($\mathtt{e_n}$).

Tuple/List/Operation: Tuples and lists can be patterns or expressions. Operations can only be expressions. The initial node is the tuple ({}), list ([]) or operator (+, ∗, etc.) and the final nodes are the final nodes of all participating expressions ($\mathtt{e_1} \ldots \mathtt{e_n}$).

Case-Expression: The evaluated expression is represented by `e`, and the last nodes of its clauses are its final nodes.

If-Expression: Similar to case-expressions but missing the evaluated expression.

Function Call: The function is represented by `e`, the arguments are $\mathtt{e_1} \ldots \mathtt{e_n}$ and the final node is the `return` node that represents the output of the function call.

List Comprehension: A list comprehension contains n generators formed by an expression and a pattern; m filters ($\mathtt{e_1} \ldots \mathtt{e_m}$) and the final expression (`e`).

Definition 1 (Graph Representation). *The graph representation of an Erlang program is a labelled graph* $(\mathcal{N}, \mathcal{C})$ *where* $\mathcal{N}$ *are the nodes and* $\mathcal{C}$ *are the edges. Additionally, the following functions are associated to the graph:*

$$
\begin{aligned}
&\mathit{type} : \mathcal{N} \to \mathcal{T} \\
&\mathit{pos} : \mathcal{N} \to \mathcal{P} \\
&\mathit{function} : \mathcal{N} \to (\mathit{atom}, \mathit{number}) \\
&\mathit{child} : (\mathcal{N}, \mathit{number}) \to \mathcal{N} \\
&\mathit{children} : \mathcal{N} \to \{\mathcal{N}\} \\
&\mathit{lasts} : \mathcal{N} \to \{\mathcal{N}\} \\
&\mathit{rootLasts} : \mathcal{N} \to \{\mathcal{N}\}
\end{aligned}
$$

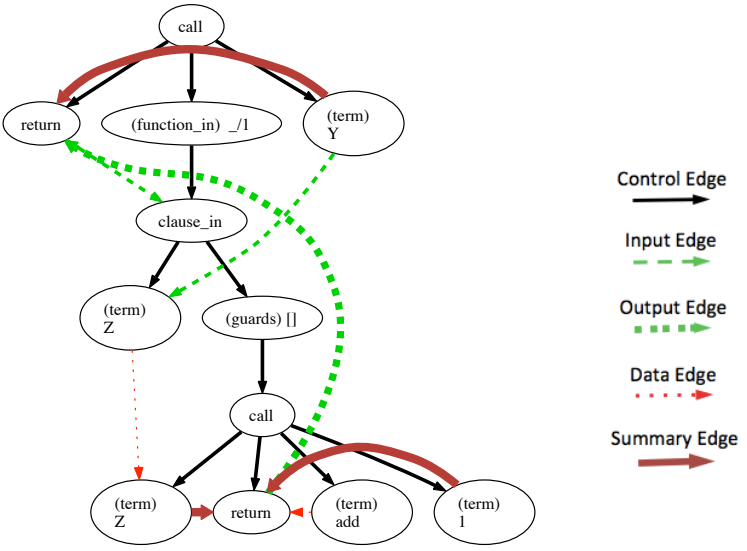


Fig. 4. EDG associated to expression `fun(Z)->add(Z,1) end(Y)` of Example 1

For each function of the program there is a function definition graph that is compositionally constructed according to the cases of Figure 3.

Total function *type* returns the type of a node. $\mathcal{T}$ is the set of node types: `function_in`, `clause_in`, `pm`, `guards`, `fid` (function identifier), `var`, `lit`, `block`, `case`, `if`, `tuple`, `list`, `op`, `call`, `lc`, and `return`. The total function *pos* returns the program position associated to a node. Partial function *function* is defined for nodes of types `function_in`, and it returns a tuple containing the function name and its arity. Function *child* returns the child that is in the position specified by the inputed number of a given node. Function *children* returns all the children of a given node. Given a node in the EDG, function *lasts* returns the final nodes associated to this node (observe that these nodes will always be leafs). Finally, given a node in an EDG that is associated to one of the graphs in Figure 3, function *rootLast* returns for each final node of this graph, (1) the initial node of the graph that must replace this node (in the case the node is gray), or (2) the node itself (in the case the node is white). This function is useful to collect the initial nodes of all arguments of a function clause.

Example 2. The graph in Figure 4 has been automatically generated by our implementation, and it illustrates the composition of some graphs associated to the code in Example 1. This graph corresponds to the function call `fun(Z)->add(Z,1) end(Y)`. For the time being, the reader can ignore all dashed, dotted and bold edges. In this graph, the final nodes of the `call` nodes are their

respective `return` nodes. Also, the result produced by function *rootLast* taking the `clause_in` node as input is the `call` node that is its descendant.

4.1 Control Edges

The graph representation of a program implicitly defines the control dependence between the components of the program.

Definition 2 (Control Dependence). *Given the graph representation of an Erlang program* $(\mathcal{N}, \mathcal{C})$ *and two nodes* $n, n' \in \mathcal{N}$*, we say that* n' *is* control dependent *on* n *if and only if* $(n \rightarrow n') \in \mathcal{C}$*.*

In Figure 4, there are control edges, e.g., between nodes $clause_{in}$ and *tuple*.

4.2 Data Edges

The definition of data dependence in Erlang is more complicated than in the imperative paradigm mainly due to pattern matching. Data dependence is used in four cases: (i) to represent the flow dependence between the definition of a variable and its later use (as in the imperative paradigm), (ii) to represent the matches in pattern matching, (iii) to represent the implicit restrictions imposed by patterns in clauses, and (iv) to relate the name of a function with the result produced by this function in a function call. Let us explain and define each case separately.

Dependence Produced by Flow Relations. In the imperative paradigm data dependence relations are due to flow dependences. These relations also happen in Erlang. As usual it is based on the sets $Def(n)$ and $Use(n)$ [15] that in Erlang contain the (single) variable defined (respectively used) in node $n \in \mathcal{N}$.

Given two nodes $n, n' \in \mathcal{N}$, we say that n' is *flow dependent* on n if and only if $Def(n) = Use(n')$ and n' is in the scope of n. We define the set $\mathcal{D}_f$ as the set containing all data dependencies of this kind, i.e., $\mathcal{D}_f = \{(n, n') \mid n' \text{ is } \textit{flow dependent} \text{ on } n\}$.

As an example, there is a data dependence of type $\mathcal{D}_f$ between the pairs of nodes containing variables `Z` in Figure 4, and between variables `A` in Figure 5.

Dependence Produced by Pattern Matching. In this section, when we talk about pattern matching, we refer to the matching of an expression against a pattern. For instance, the graph of `{X,Y}` matches the graph of `{Z,42}` with three matching nodes: {} with {}, `X` with `Z` and `Y` with 42. Also, the graph of the expression `if X>1 -> true; _ -> false end` matches the graph of the pattern `Y` with two matching nodes: `Y` with `true` and `Y` with `false`. Pattern matching is used in three situations, namely, (i) in case-expressions to match the expression against each of the patterns, (ii) in pattern-matching-expressions, and (iii) in function calls to match each of the parameters to the arguments of the called function. Here we only consider the first two items because the third one is represented with another kind of edge that will be discussed in Section 4.3. Given

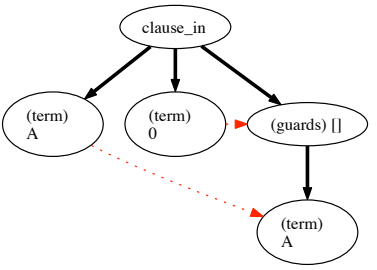


Fig. 5. EDG associated to clause `add(A,0) -> A` of Example 1

the initial node of a pattern (say n_p) and the initial node of an expression (say n_e) we can compute all matching pairs in the graph with function *match* that is recursively defined as:

$$
\begin{array}{l}
match(n_p, n_e) = \\
\{(n_e, n_p) \mid type(n_e) = \texttt{var} \vee \\
\qquad (\ type(n_p), type(n_e) \in \{\texttt{lit}, \texttt{fid}\} \\
\qquad\quad \wedge\ elem(pos(n_p)) = elem(pos(n_e))\ )\} \cup \\
\{(last_e, n_p) \mid (\ type(n_p) = \texttt{var} \vee \\
\qquad\qquad (type(n_p) = \texttt{lit} \wedge\ type(n_e) \in \{\texttt{op}, \texttt{call}\}) \vee \\
\qquad\qquad (type(n_p) = \texttt{tuple} \wedge\ type(n_e) = \texttt{call}) \vee \\
\qquad\qquad (type(n_p) = \texttt{list} \wedge\ type(n_e) \in \{\texttt{op}, \texttt{call}, \texttt{lc}\})\ ) \\
\qquad\quad \wedge\ last_e \in lasts(n_e)\} \cup \\
\{edge \mid (\ (type(n_p) \in \{\texttt{lit}, \texttt{tuple}, \texttt{list}\} \wedge\ type(n_e) \in \{\texttt{case}, \texttt{if}, \texttt{pm}, \texttt{block}\}) \\
\qquad \wedge\ edge \in \bigcup_{n'_e \in rootLasts(n_e)} match(n_p, n'_e)\ ) \vee \\
\qquad (\ type(n_p) = \texttt{pm} \wedge\ edge \in \bigcup_{n'_p \in rootLasts(n_p)} match(n'_p, n_e)\ )\} \cup \\
\{edge \mid type(n_p) \in \{\texttt{tuple}, \texttt{list}\} \wedge\ type(n_e) = type(n_p) \\
\qquad \wedge\ |\{n' \mid (n_e \to n') \in \mathcal{C}\}| = |\{n' \mid (n_p \to n') \in \mathcal{C}\}| \\
\qquad \wedge \bigwedge_{i \in 1 \ldots |children(n_e)|} match(child(n_p, i), child(n_e, i)) \neq \emptyset \\
\qquad \wedge\ edge \in (\ (n_e, n_p) \cup (\ \bigcup_{i \in 1 \ldots |children(n_e)|} match(child(n_p, i), child(n_e, i))\ )\}
\end{array}
$$

The set of all pattern matching edges in a graph is denoted with $\mathcal{D}_{pm}$.

Dependence Produced by Restrictions Imposed by Patterns. The patterns that appear in clauses can impose restrictions to the possible values of the expressions that can match these patterns. For instance, the patterns used in the function definition `foo(X,X,0,Y) -> Y` impose two restrictions that must be fulfilled in order to return the value `Y`: (1) The first two arguments must be equal, and (2) the third argument must be `0`.

These restrictions can be represented with an arc from the pattern that imposes a restriction to the guards node of the clause; meaning that, in order to reach the guards, the restrictions of the nodes that point to the guards must be fulfilled. The set of all restrictions in a graph is denoted with $\mathcal{D}_r$, and it can be easily computed with function *constraints* that takes the initial node of a pattern and the set of repeated variables in the parameters of the clause associated to the pattern, and it returns all nodes in the pattern that impose restrictions.

$$constraints(n, RVars) = \begin{cases} \{n\} & type(n) = \texttt{lit} \ \vee \\ & (type(n) = \texttt{var} \ \wedge \ elem(pos(n)) \in RVars) \\ \{n\} \cup \bigcup\limits_{n' \in children(n)} constraints(n', RVars) & type(n) \in \{\texttt{list}, \texttt{tuple}\} \\ \bigcup\limits_{n' \in children(n)} constraints(n', RVars) & type(n) = \texttt{pm} \\ \emptyset & otherwise \end{cases}$$

As an example, there is a data dependence of type $\mathcal{D}_r$ between the term `0` and the guard node in Figure 5.

Dependence Produced in Function Calls. The returned value of a function call always depends on the function that has been called. In order to represent this kind of dependence, we add an edge from any node that can represent the name of the function that is being called to the return node of the function call. Note that the name of the function is always represented by a node of type `atom`, `variable` or `fid`. We represent the set containing all dependences of this kind with $\mathcal{D}_{fc}$.

As an example, there is a data dependence of type $\mathcal{D}_{fc}$ between the node containing the literal `add` and the `return` node in Figure 4.

We are now in a position to define a notion of data dependence in Erlang.

Definition 3 (Data Dependence). *Given the graph representation of an Erlang program* $(\mathcal{N}, \mathcal{C})$ *and two nodes* $n, n' \in \mathcal{N}$*, we say that* n' *is* data dependent *on* n *if and only if* $(n, n') \in (\mathcal{D}_f \cup \mathcal{D}_{pm} \cup \mathcal{D}_r \cup \mathcal{D}_{fc})$.

4.3 Input/Output Edges

Input and output edges represent the information flow in function calls. One of the problems of functional languages such as Erlang is that higher-order calls can hide the name of the function that is being called. And even if we know the name of the function, it is not always possible to know the actual clause that will match the function call.

Example 3. In the following program, it is impossible to statically know what clause will match the function call `g(X)` and thus we need to connect the function call to all possible clauses that could make pattern matching at execution time.

```
-export(f/1).

f(X) -> g(X).

g(1)-> a;
g(X)-> b.
```

Determining all possible clauses that can pattern match a call is an undecidable problem because a call can depend on the termination of a function call, and proving termination is undecidable. Therefore, we are facing a fundamental problem regarding the precision of the graphs. Conceptually, we can assume the existence of a function `clauses(call)` that returns all clauses that match a given call. In practice, some static analysis must be used to approximate the clauses. In our implementation we use Typer [10] that uses the type inference system of Dialyzer [11] producing a complete approximation.

Given a graph $(\mathcal{N}, \mathcal{C})$ we define the set $\mathcal{I}$ of input edges as a set of directed edges. For each function call graph *call*, we make graph matching between each parameter subgraph in the call to each argument subgraph in the clauses belonging to *clauses*(*call*). There is an edge in $\mathcal{I}$ for each pair of nodes matching. Moreover, there is an edge from the `return` node of the call to the $\texttt{clause}_{\texttt{in}}$ node of the clause. As an example, in Figure 4, there are input edges from node with variable `Y` to node with variable `Z` and from the `return` node to the `clause_in` node.

Given a graph $(\mathcal{N}, \mathcal{C})$ we define the set $\mathcal{O}$ of output edges as a set of directed edges. For each function call graph *call* and each clause belonging to `clauses(call)`. There is an edge in $\mathcal{O}$ for each final node of the clause graph to the `return` node of the call. As an example, in Figure 4, there is an output edge between the two `return` nodes.

4.4 Summary Edges

Summary edges are used to precisely capture inter-function dependences. Basically, they say what arguments of a function do influence the result of this function (see [6] for a deep explanation about summary edges). Given a graph $(\mathcal{N}, \mathcal{C})$, we define the set $\mathcal{S}$ of summary edges as a set of directed edges. As in the imperative paradigm, they can be computed once all the other dependencies have been computed. We have a summary edge between two nodes n, n' of the graph if n belongs to the graph representing (a part of) an argument of a function call, n' is the return-node of the function call, and there is an input edge starting at n. In Figure 4, the summary edges are all bold edges.

We are now in a position to formally introduce the Erlang Dependence Graphs.

Definition 4 (Erlang Dependence Graph). *Given an Erlang program $\mathcal{P}$, its associated* Erlang Dependence Graph (EDG) *is the directed labelled graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ are the nodes and $\mathcal{E} = (\mathcal{C}, \mathcal{D}, \mathcal{I}, \mathcal{O}, \mathcal{S})$ are the edges.*

Example 4. The EDG corresponding to the expression `fun(Z)-> add(Z,1) end(Y)` in line `(14)` of Figure 1 is shown in Figure 4.

5 Slicing Erlang Specifications

The EDG is a powerful tool to perform different static analysis and it is particularly useful for program slicing.

In this section we show that our adaptation of the SDG to Erlang keeps the most important property of the SDG: computing a slice from the EDG has a cost linear with the size of the EDG. This means that we can compute slices with a single traversal of the EDG. However, the algorithm used to traverse the EDG is not the standard one. We only need to make one small modification that allows us to improve precision.

One important advantage of the EDG with respect to the SDG is that it minimizes the granularity level. In the EDG all syntactical constructs are decomposed to the maximum (i.e, literals, variables, etc.). Contrarily, in the imperative paradigm, each node represents a complete line in the source code. Therefore, we can produce slices that allows us to know what parts of the program affect a given (sub)expression at any point.

Definition 5. *Given an EDG $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, a slicing criterion for $\mathcal{G}$ is a node $n \in \mathcal{N}$.*

In practice, the EDG is hidden to the programmer, and the slicing criterion is selected in the source code. In our implementation this is done by just highlighting an expression in the code. This action is automatically translated to a position that in turn is the identifier of one node in the EDG. This node is the input of Algorithm 1 that allows us to extract slices from an EDG. Essentially, Algorithm 1 first collects all nodes that are reachable from the slicing criterion following backwards all edges in $\mathcal{C} \cup \mathcal{D} \cup \mathcal{I}$. And then it collects from these nodes, all nodes that are reachable following backwards all edges in $\mathcal{C} \cup \mathcal{D} \cup \mathcal{O}$. In both phases, the nodes that are reachable following backwards edges in $\mathcal{S}$ are also collected, but only if they are connected to a node that belongs to the slice through an input edge.

The behavior of the algorithm is similar to the standard one except for the treatment of summary edges. In the SDG, summary edges go from the input parameters to the output parameters of the function and they are always traversed. Moreover, each of the parameters cannot be decomposed. In contrast, in Erlang, the arguments of a function can be composite data structures, and thus, it is possible that only a part of this data structure influences the slicing criterion. Therefore, in function calls, we only traverse the summary edges if they come from nodes that are actually needed. The way to know that they are actually needed is to observe their outgoing input edge and know if the node pointed does belong to the slice. Of course this can only be known after having analyzed the function that is called.

Once we have collected the nodes that belong to the slice, it is easy to map the slice into the source code. For a program $\mathcal{P}$, the exact collection of positions (lines and columns) that belong to the slice is $\{pos(n) \mid n \in Slice(\mathcal{P})\}$ where function *Slice* implements Algorithm 1. In order to ensure that the final transformed

Algorithm 1. Slicing interprocedural programs

Input: An EDG $\mathcal{G} = (\mathcal{N}, \mathcal{E} = (\mathcal{C}, \mathcal{D}, \mathcal{I}, \mathcal{O}, \mathcal{S}))$ and a slicing criterion $\mathcal{SC}$
Output: A collection of nodes $Slice \in \mathcal{N}$
return traverse(traverse($\{\mathcal{SC}\}, \mathcal{I}$), $\mathcal{O}$)

function traverse($Slice, X$)
  repeat
    $Slice = Slice \cup \{n' \mid (n' \to n) \in (\mathcal{C} \cup \mathcal{D} \cup X)$ with $n \in Slice\}$
                $\cup \{n_2 \mid (n_2 \to n_1) \in \mathcal{S} \wedge (n_2 \to n_3) \in \mathcal{I}$ with $n_1, n_3 \in Slice\}$
  until a fix point is reached
return $Slice$

program is executable, we also have to replace those expressions that are not in the slice by the (fresh) atom `undef` and those unused patterns by an anonymous variable. The result of our algorithm with respect to the program in Figure 1 is shown in Figure 2.

6 Conclusions and Future Work

This work adapts the SDG to be used with Erlang programs. Based on this adaptation, we introduce a program slicing technique that precisely produces slices of interprocedural Erlang programs. This is the first adaptation of the SDG for a functional language. Even though we implemented it for Erlang, we think that it can be easily adapted to other functional languages with slight modifications.

The slices produced by our technique are executable. This means that other analysis and tools could use our technique as a preprocessing transformation stage simplifying the initial program and producing a more accurate and reduced one that will predictably speed up the subsequent transformations. We have implemented a slicer for Erlang that generates EDGs, this tool is called `Slicerl` and it is publicly available at:

`http://kaz.dsic.upv.es/slicerl`

The current implementation of `Slicerl` accepts more syntactical constructs that those described in this paper. It is able to produce slices of its own code. Even though the use of summary edges together with the algorithm proposed provides a solution to the interprocedural loss of precision, there is still a loss of precision that is not faced by our solution. This loss of precision is produced by the expansion and compression of data structures.

Example 5. Consider the Erlang program at the left and the slicing criterion `Y` in line `(4)`:

```
(1)  main() ->
(2)    X={1,2},
(3)    {Y,Z}=X,
(4)    Y.
```

```
(1)  main() ->
(2)    X={1,2},
(3)    {Y,_}=X,
(4)    Y.
```

```
(1)  main() ->
(2)    X={1,_},
(3)    {Y,_}=X,
(4)    Y.
```

Our slicing algorithm produces the slice shown in the center. It is not able to produce the more accurate slice shown at the right because it losses precision.

The loss of precision shown in Example 5 is due to the fact that the EDG does not provide any mechanism to trace an expression when it is part of a data structure that is collapsed into a variable and then expanded again. In the example, there is a dependence between variable `Y` and variable `X` in line `(3)`, This dependence means "*The value of* `Y` *depends on the value of* `X`". Unfortunately, this is only partially true. The real meaning should be "*The value of* `Y` *depends on a part of the value of* `X`". We are currently defining a new dependence called *partial-dependence* to solve this problem. A solution to this problem has already been defined in [16]. Its implementation will be available soon in the webpage of `Slicerl`.

References

1. Agrawal, H., Horgan, J.R.: Dynamic program slicing. In: Programming Language Design and Implementation (PLDI), pp. 246–256 (1990)
2. Brown, C.: Tool Support for Refactoring Haskell Programs. PhD thesis, School of Computing, University of Kent, Canterbury, Kent, UK (2008)
3. Cheda, D., Silva, J., Vidal, G.: Static slicing of rewrite systems. Electron. Notes Theor. Comput. Sci. 177, 123–136 (2007)
4. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The Program Dependence Graph and Its Use in Optimization. ACM Transactions on Programming Languages and Systems 9(3), 319–349 (1987)
5. Field, J., Ramalingam, G., Tip, F.: Parametric program slicing. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, pp. 379–392. ACM, New York (1995)
6. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions Programming Languages and Systems 12(1), 26–60 (1990)
7. Korel, B., Laski, J.: Dynamic Program Slicing. Information Processing Letters 29(3), 155–163 (1988)
8. Larsen, L., Harrold, M.J.: Slicing object-oriented software. In: Proceedings of the 18th International Conference on Software Engineering, ICSE 1996, pp. 495–505. IEEE Computer Society, Washington, DC (1996)
9. Liang, D., Harrold, M.J.: Slicing objects using system dependence graphs. In: Proceedings of the International Conference on Software Maintenance, ICSM 1998, pp. 358–367. IEEE Computer Society, Washington, DC (1998)
10. Lindahl, T., Sagonas, K.F.: Typer: a type annotator of erlang code. In: Sagonas, K.F., Armstrong, J. (eds.) Erlang Workshop, pp. 17–25. ACM (2005)
11. Lindahl, T., Sagonas, K.F.: Practical type inference based on success typings. In: Bossi, A., Maher, M.J. (eds.) PPDP, pp. 167–178. ACM (2006)
12. Ochoa, C., Silva, J., Vidal, G.: Dynamic slicing based on redex trails. In: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2004, pp. 123–134. ACM, New York (2004)
13. Reps, T., Turnidge, T.: Program Specialization via Program Slicing. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Dagstuhl Seminar 1996. LNCS, vol. 1110, pp. 409–429. Springer, Heidelberg (1996)

14. Rodrigues, N.F., Barbosa, L.S.: Component identification through program slicing. In: Proc. of Formal Aspects of Component Software (FACS 2005). Elsevier ENTCS, pp. 291–304. Elsevier (2005)
15. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3(3), 121–189 (1995)
16. Tóth, M., Bozó, I., Horváth, Z., Lövei, L., Tejfel, M., Kozsik, T.: Impact Analysis of Erlang Programs Using Behaviour Dependency Graphs. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) CEFP 2009. LNCS, vol. 6299, pp. 372–390. Springer, Heidelberg (2010)
17. Walkinshaw, N., Roper, M., Wood, M., Roper, N.W.M.: The java system dependence graph. In: Third IEEE International Workshop on Source Code Analysis and Manipulation, p. 5 (2003)
18. Weiser, M.: Program Slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449. IEEE Press (1981)
19. Widera, M.: Flow graphs for testing sequential erlang programs. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang, ERLANG 2004, pp. 48–53. ACM, New York (2004)
20. Widera, M., Informatik, F.: Concurrent erlang flow graphs. In: Proceedings of the Erlang/OTP User Conference (2005)
21. Zhao, J.: Slicing aspect-oriented software. In: Proceedings of the 10th International Workshop on Program Comprehension, IWPC 2002, pp. 251–260. IEEE Computer Society, Washington, DC (2002)

A Domain-Specific Language for Scripting Refactorings in Erlang

Huiqing Li and Simon Thompson

School of Computing, University of Kent, UK
{H.Li,S.J.Thompson}@kent.ac.uk

Abstract. Refactoring is the process of changing the design of a program without changing its behaviour. Many refactoring tools have been developed for various programming languages; however, their support for composite refactorings – refactorings that are composed from a number of primitive refactorings – is limited. In particular, there is a lack of powerful and easy-to-use frameworks that allow users to script their own large-scale refactorings efficiently and effectively.

This paper introduces the domain-specific language framework of Wrangler – a refactoring and code inspection tool for Erlang programs – that allows users to script composite refactorings, test them and apply them on the fly. The composite refactorings are fully integrated into Wrangler and so can be previewed, applied and 'undone' interactively.

Keywords: analysis, API, DSL, Erlang, refactoring, transformation, Wrangler.

1 Introduction

Refactoring [1] is the process of changing the design of a program without changing what it does. A variety of refactoring tools have been developed to provide refactoring support for various programming languages, such as the Refactoring Browser for Smalltalk [2], IntelliJ Idea [3] for Java, ReSharper [3] for C#, VB.NET, Eclipse [4]'s refactoring support for C++, Java, and much more. For functional programming languages there is, for example, the HaRe [5] system for Haskell, and for Erlang the two systems Wrangler [6] and RefactorErl [7].

In their recent study on how programmers refactor in practice [8], Murphy-Hill et. al. point out that *"refactoring has been embraced by a large community of users, many of whom include refactoring as a constant companion to the development process"*. However, following the observation that about forty percent of refactorings performed using a tool occur in batches, they also claim that existing tools could be improved to support *batching* refactorings together.

Indeed, it is a common refactoring practice for a set of primitive refactorings to be applied in sequence in order to achieve a complex refactoring effect, or for a single primitive refactoring to be applied multiple times across a project to perform a large-scale refactoring. For example, a refactoring that extracts a sequence of expressions into a new function might be followed by refactorings

J, de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 501–515, 2012.

that rename and re-order the parameters of the new function. This could be followed by 'folding' all instances of the new function body into applications of the new function, thus eliminating any clones of the original expression. As another example, in order to turn all 'camelCase' names into 'camel_case' format, a renaming refactoring will have to be applied to each candidate.

Although composite refactorings are applied very often in practice, tool support for composite refactorings lags behind. While some refactoring tools, such as the Eclipse LTK [9], expose an API for users to compose their own refactorings, these APIs are usually too low-level to be useful to the working programmer.

In this paper, we present a simple, but powerful, Domain Specific Language (DSL) based framework built into Wrangler, a user-extensible refactoring and code inspection tool for Erlang programs. The framework allows users to:

- script reusable composite refactorings from the existing refactorings in a declarative and program independent way;
- have fine control over the execution of each primitive refactoring step;
- control the propagation of failure during execution;
- generate refactoring commands in a lazy and dynamic way.

User-defined composite refactorings can be invoked from the *Refactor* menu in the IDE, and so benefit from features such as result preview, undo, etc.

Using the DSL allows us to write refactorings – such as the change of naming style discussed earlier – in a fraction of the time that would be required to do this by hand; we therefore make the cost of learning the DSL negligible in comparison to the benefits that accrue to the user. Moreover, once written these refactorings can be reused by the author and others.

Not only does the DSL make descriptions more compact, it also allows us to describe refactorings that are impossible to describe in advance. For example, suppose that a program element is renamed at some point in the operation; in the DSL we can refer to it by its old name rather than its new name, which may only be known once it has been input interactively during the transformation.

While this work is described in the context of Erlang. the underlying ideas and DSL design are equally applicable to other programming languages, and can be implemented in a variety of ways (e.g. reflection, meta-programming).

The rest of the paper is organised as follows. Section 2 gives an overview of Erlang, Wrangler and its template-based API. Section 3 formalises some concepts that we use in describing our work. In Section 4 we explain the rationale for our approach to designing the DSL, which we then describe in detail in Section 5. Examples are given in Section 6, and the implementation is discussed in Section 7. Sections 8 and 9 conclude after addressing related and future work.

The work reported here is supported by ProTest, EU FP7 project 215868.

2 Erlang, Wrangler and Its Template-Based API

Erlang 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.

```
-module (fact).
-export ([fac/1]).

fac(0) -> 1;
fac(N) when N > 0 ->
    N * fac(N-1).
```

Fig. 1. Factorial in Erlang

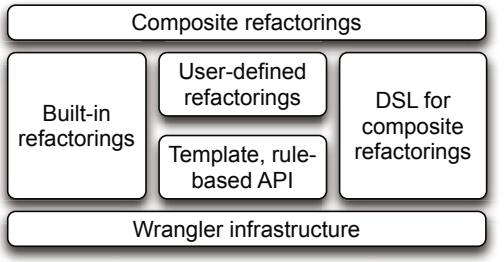


Fig. 2. The Wrangler Architecture

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; furthermore, a module may only export functions that are defined in the module itself.

Calls to functions defined in other modules generally qualify the function name with the module name: the function `foo` from the module `bar` is called as: `bar:foo(...)`. Figure 1 shows an Erlang module containing a definition of the factorial function. In this example, `fac/1` denotes the function `fac` with arity of 1. 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 [6], downloadable from `https://github.com/RefactoringTools`, is a tool that supports interactive refactoring and "code smell" inspection of Erlang programs, and is integrated with (X)Emacs and with Eclipse. Wrangler is itself implemented in Erlang. Abstract Syntax Trees (ASTs) expressed as Erlang data structures are used as the internal representation of Erlang programs. The AST representation is structured in such a way that all the AST nodes have a uniformed structure, and each node can be attached with various annotations, such as location, source-code comments, static-semantic information, etc.

One of the problems faced by refactoring tool developers is the fact that the number of refactorings that they are able to support through the tool is limited, whereas the number of potential refactorings is unbounded. With Wrangler, this problem is solved by providing a high-level *template- and rule-based API*, so that users can write refactorings that meet their own needs in a concise and intuitive way without having to understand the underlying AST representation and other implementation details. A similar strategy is used to solve the problem of describing composite refactorings, that is, a high-level DSL-based framework is provided to allow users to script their own composite refactorings. Wrangler's architecture is shown in Figure 2.

Wrangler's Template-based API [10] allows Erlang programmers to express program analysis and transformation in concrete Erlang syntax. In Wrangler, a code template is denoted by an Erlang macro `?T` whose only argument is the string representation of an Erlang code fragment that may contain meta-variables or

meta-atoms. A meta-variable is a placeholder for a syntax element in the program, or a sequence of syntax elements of the same kind; and a meta-atom is a place holder for a syntax element that can only be an atom, such as the function name part of a function definition.

Syntactically a meta-variable/atom is an Erlang variable/atom, ending with the character '@'. A meta-variable, or atom, ending with a single '@' represents a single language element, and matches a single subtree in the AST; a meta-variable ending with '@@' represents a list meta-variable that matches a sequence of elements of the same sort. For instance, the template

```
?T("erlang:spawn(Arg@@)")
```

matches the application of `spawn` to an arbitrary number of arguments, and `Args@@` is a placeholder for the sequence of arguments; whereas the template

```
?T("erlang:spawn(Args@@, Arg1@)")
```

only matches the applications of `spawn` to one or more arguments, where `Arg1@` is a placeholder for the last argument, and `Args@@` is the placeholder for the remaining leading arguments (if any).

Templates are matched at AST level, that is, the template's AST is pattern matched to the program's AST using structural pattern matching techniques. If the pattern matching succeeds, the meta-variables/atoms in the template are bound to AST subtrees, and the context and static semantic information attached to the AST subtrees matched can be retrieved through functions from the API suite provided by Wrangler.

The Erlang macro `?COLLECT` is defined to allow information collection from nodes that match the template specified and satisfies certain conditions. Calls to the macro `?COLLECT` have the format:

```
?COLLECT(Template, Collector, Cond)
```

in which `Template` is a template representation of the kind of code fragments of interest; `Cond` is an Erlang expression that evaluates to either `true` or `false`; and `Collector` is an Erlang expression which retrieves information from the current AST node. We call an application of the `?COLLECT` macro as a *collector*.

Information collection is typically accomplished by a tree-walking algorithm. In Wrangler, various AST traversal strategies have been defined, in the format of macros, to allow the walking of ASTs in different orders and/or for different purposes. A tree traversal strategy takes two arguments: the first is a list of collectors or transformation rules, and the second specifies the scope to which the analysis, or transformation, is applied to.

For example, the macro `?FULL_TD_TU` encapsulates a tree-walking algorithm that traverses the AST in a top-down order (`TD`), visits every node in the AST (`FULL`), and returns information collected during the traversal (`TU` for 'type unifying', as opposed to 'type preserving'). The code in Figure 3 shows how to collect all the application occurrences of function `lists:append/2` in an Erlang file. For each application occurrence, its source location is collected. `_This@` is a predefined meta-variable representing the current node that matches the template.

The template-based API can be used to retrieve information about a program during the scripting of composite refactorings, as will be shown in Section 6.

```
?FULL_TD_TU([?COLLECT(?T("lists:append(L1@, L2@)"),
                      api_refac:start_end_loc(_This@), true)], [File])
```

Fig. 3. Collect the application instances of lists:append/2

As was explained above, more details about the API can be found in [10]; in particular, it explains how the API can be used to define transformation rules which are also applied to ASTs by means of a tree-walking algorithm (as above).

3 Terminology

This section introduces terminology that we use in discussing our DSL. Particularly we explain what we mean by success and failure for a composite refactoring.

Definition 1. *A* ***precondition*** *is a predicate, possibly with parameters, over a program or a sub-program that returns either* `true` *or* `false`*.*

Definition 2. *A* ***transformation rule*** *maps one program into another.*

Definition 3. *A* ***primitive refactoring*** *is an elementary behaviour-preserving source-to-source program transformation that consists of a set of preconditions C, and a set of transformation rules T. When a primitive refactoring is applied to a program, all the preconditions are checked before the program is actually transformed by applying all the transformation rules. We say a primitive refactoring* ***fails*** *if the conjunction of the set of preconditions returns false; otherwise we say the primitive refactoring* ***succeeds****.*

Definition 4. ***Atomic composition*** *Given a sequence of refactorings* $R_1, ..., R_n, n \geq 1$*, the* ***atomic composition*** *of* $R_1, ..., R_n$*, denoted as* $R_1 \circ R_2 \circ \cdots \circ R_n$*, creates a new refactoring consisting of the sequential application of refactorings from* R_1 *to* R_n*.*

If any of the applications of $R_i, 1 \leq i \leq n$ *fails, then the whole refactoring fails and the original program is returned unchanged. The composite refactoring succeeds if all the applications* R_i *for* $1 \leq i \leq n$ *succeeds, and the result program is the program returned after applying* R_n*.*

Definition 5. ***Non-atomic composition*** *Given a sequence of refactorings* $R_1, ..., R_n, n \geq 1$*, the* ***non-atomic composition*** *of* $R_1, ..., R_n$*, denoted as* $R_1 \diamond R_2 \diamond \cdots \diamond R_n$*, creates a new refactoring consisting of the sequential application of refactorings from* R_1 *to* R_n*.*

If refactoring R_i *fails, the execution of* R_{i+1} *continues, on the last succeeding application (or the original program if none has succeeded so far). A failed refactoring does not change the status of the program. The program returned by applying* R_n *is the final result of the application of the composite refactoring. As a convention, we say that a non-atomic composite refactoring always succeeds.*

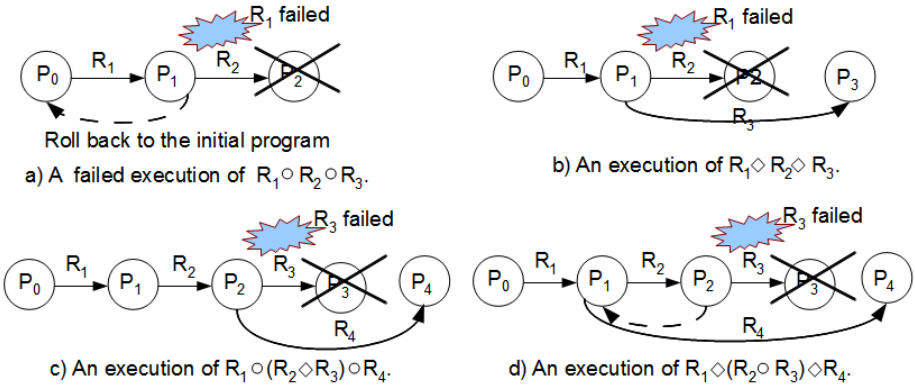


Fig. 4. Execution of composite refactorings

Figure 4 illustrates some execution scenarios of both atomic and non-atomic composite refactorings. As shown in c) and d), an atomic composite refactoring can be part of a non-atomic composite refactoring, and *vice versa*; this feature allows the tool to handle more complex refactoring scenarios.

In practice, the choice of atomic or non-atomic composition depends on the nature of the refactoring to be performed. Atomic composition is necessary if the failure of a constituent refactoring could lead to an inconsistent or incorrect program, whereas a non-atomic composition can be used when the failure of a constituent refactoring does not affect the consistency of the program, and the final program returned is still acceptable from the user's point of view.

For example, it is reasonable to make a non-atomic composition of a set of renaming refactorings that turn 'camelCase' function names into 'camel_case' format; even if one of these fails, perhaps because the new name is already used, the program still works as before. Moreover, the user can manually make the changes to the remaining 'camel_case' identifiers; if 90% of the work has been done by the script, 90% of user effort is correspondingly saved.

4 Rationale

Here we discuss the rationale for the design of the DSL. While it is possible to describe composite refactorings manually; that approach is limited:

- When the number of primitive refactoring steps involved is large, enumerating all the primitive refactoring commands could be tedious and error prone.
- The static composition of refactorings does not support generation of refactoring commands that are program-dependent or refactoring scenario dependent, or where a subsequent refactoring command is somehow dependent on the results of an earlier application.
- Some refactorings refer to program entities by source location instead of name, as this information may be extracted from cursor position in an editor or IDE, say. Tracking of locations is again tedious and error prone; furthermore, the

location of a program entity might be changed after a number of refactoring steps, and in that case locations become untrackable.
– Even though some refactorings refer to program entities by name (rather than location), the name of a program entity could also be changed after a number of refactoring steps, which makes the tracking of entity names hard or sometimes impossible, particularly when non-atomic composite refactorings are involved.

We resolve these problems in a number of ways:

– Each primitive refactoring has been extended with a *refactoring command generator* that can be used to generate refactoring commands in batch mode.
– A command generator can generate commands lazily, i.e., a refactoring command is generated only as it is to be applied, so we can make sure that the information gathered by the generator always reflects the latest status, including source locations, of the program under refactoring.
– Wrangler always allows a program entity to be referenced using its original name, as it performs name tracking behind the scenes.
– Finally, and most importantly, we provide a small domain-specific language (DSL) to allow composition of refactorings in a compact and intuitive way. The DSL allows users to have a fine control over the generation of refactoring commands and the interaction between the user and the refactoring engine so as to allow decision making during the execution of the composite refactoring.

Our work defines a small DSL, rather than a (fluent) API, since it supports a variety of ways of combining refactorings, including arbitrary nesting of refactoring descriptions within others, rather than offering a variety of parameters on a fixed set of API functions.

Existing approaches to composite refactoring tend to focus on the derivation of a combined precondition for a composite refactoring, so that the entire precondition of the composite refactoring can be checked on the initial program before performing any transformation [11,12]. The ostensible rationale for this is to give improved performance of the refactoring engine. However, given the usual way in which refactoring tools are used in practice – where the time to decide on the appropriate refactoring to apply will outweigh the execution time – we do not see that the efficiency gains that this approach might give are of primary importance to the user.

In contrast, our aim is to increase the *usability* and *applicability* of the refactoring tool, by expanding the way in which refactorings can be put together. Our work does not try to carry out precondition derivation, instead each primitive refactoring is executed in the same way as it is invoked individually, i.e., precondition checking followed by program transformation. While it may be less efficient when an atomic composite refactoring fails during the execution, it does have its advantages in expressibility.

5 A Framework for Scripting Composite Refactorings

In this section we give a detailed account of Wrangler's support for composite refactorings, treating each aspect of the DSL in turn.

5.1 Refactoring Command Generators

For each primitive refactoring we have introduced a corresponding *command generator* of the same name. The interface of a command generator is enriched in such a way that it accepts not only concrete values as a primitive refactoring does, but also structures that specify the constraints that a parameter should meet or structures that specify how the value for a parameter should be generated. In general, generators will have different type signatures, corresponding to the different signatures of their associated refactorings.

When applied to an Erlang program, a command generator searches the AST representation of the program for refactoring candidates according to the constraints on arguments. A command generator can also be instructed to run lazily or strictly; if applied strictly, it returns the complete list of primitive refactoring commands that can be generated in one go; otherwise, it returns a single refactoring command together with another command generator wrapped in a function closure, or an empty list if no more commands can be generated. Lazy refactoring command generation is especially useful when the primitive refactoring command refers some program entities by locations, or the effect of a previous refactoring could affect the refactorings that follow; on the other hand, strict refactoring command generation is useful for testing a command generator, as it gives the user an overall idea of the refactoring commands to be generated.

Each primitive refactoring command generated is a tuple in the format: {`refactoring, RefacName, Args`}, where `RefacName` is the name of the refactoring command, and `Args` is the list of the arguments for that refactoring command. A refactoring command generator is also syntactically represented as a three-element tuple, but with a different tag, in the format of {`refac_, RefacName, Args`}, where `RefacName` is the name of the command generator, and `Args` are the arguments that are specified by the user and supplied to the command generator. Both `refactoring` and `refac_` are Erlang atoms.

Taking the 'rename function' refactoring as an example, the type specification of the refactoring command is shown in Figure 5 (a), which should be clear enough to explain itself. The type specification of the command generator is given in Figure 5 (b). As it shows, a command generator accepts not only actual values, but also function closures that allow values to be generated by analysing the code to be refactored .

- The first parameter of the generator accepts either a file name, or a condition that a file (name) should satisfy to be refactored. In the latter case, Wrangler searches the program for files that meet the condition specified, and only those files are further analysed to generate values for the remaining parameters.
- The second parameter accepts either a function name tupled with its arity, or a condition that a function should meet in order to be refactored. In the latter case, every function in an Erlang file will be checked, and those functions that do not meet the condition are filtered out, and a primitive refactoring command is generated for each function that meets the condition.
- The third argument specifies how the new function name should be generated. It could be a fixed function name, a generator function that generates the

```
-spec rename_fun(File::filename(), FunNameArity::{atom(), integer()},
                 NewName::atom()) -> ok | {error, Reason::string()}.
```

(a) type spec of the 'rename function' refactoring.

```
-spec rename_fun(File::filename() | fun((filename()) -> boolean()),
                FunNameArity::{atom(), integer()}
                             | fun(({atom(),integer()}) -> boolean()),
                NewName::atom()
                  |{generator, fun(({filename(), {atom(), integer()}})
                                   -> atom())}
                     |{user_input,fun(({filename(), {atom(), integer()}})
                                   -> string())},
                 Lazy :: boolean())
           -> [{refactoring, rename_fun, Args::[term()]}] |
              {{refactoring, rename_fun, Args::[term()]}, function()}.
```

(b) type spec of the 'rename function' command generator.

```
   {refac_, rename_fun, [fun(_File)-> true end,
                  fun({FunName, _Arity}) -> is_camelCase(FunName) end,
                  {generator, fun({_File,{FunName,_Arity}}) ->
                                    camelCase_to_camel_case(FunName)
                               end}, false]}
```

(c) An instance of the 'rename function' command generator.

Fig. 5. Primitive refactoring command vs. refactoring command generator

new function based on the previous parameter values, or a name that will be supplied by the user before the execution of the refactoring, in which case the function closure is used to generate the prompt string that will be shown to the user when prompting for input.

– Finally, the last parameter allows the user to choose whether to generate the commands lazily or not.

The example shown in Figure 5 (c) illustrates the script for generating refactoring commands that rename all functions in a program whose name is in `camelCase` format to `camel_case` format. As the condition for the first parameter always returns true, every file in the program should be checked. The second argument checks if the function name is in `camelCase` format using the utility function `is_camelCase`, and a refactoring command is generated for each function whose name is in `camelCase` format. The new function name is generated by applying the utility function `camelCase_to_camel_case` to the old function name. In this example, we choose to generate the refactoring commands in a strict way.

For some command generators, it is also possible to specify the order in which the functions in an Erlang file are visited. By default, functions are visited as they occur in the file, but it is also possible for them to be visited according to the function callgraph in either top-down or bottom-up order.

$RefacName ::= rename\_fun \mid rename\_mod \mid rename\_var \mid new\_fun \mid gen\_fun \mid ...$
$PR ::= \{refactoring,\ RefacName,\ Args\}$
$CR ::= PR$
$\quad \mid \{interactive,\ Qualifier,\ [PRs]\}$
$\quad \mid \{repeat\_interactive,\ Qualifier,\ [PRs]\}$
$\quad \mid \{if\_then,\ \text{fun}() \rightarrow Cond \text{ end},\ CR\}$
$\quad \mid \{while,\ \text{fun}() \rightarrow Cond \text{ end},\ Qualifier,\ CR\}$
$\quad \mid \{Qualifier,\ [CRs]\}$
$PRs ::= PR \mid PRs,\ PR$
$CRs ::= CR \mid CRs,\ CR$
$Qualifier ::= atomic \mid non\_atomic$
$Args ::= ...A\ list\ of\ Erlang\ terms...$
$Cond ::= ...An\ Erlang\ expression\ that\ evaluates\ to\ a$ `boolean` $value...$

Fig. 6. The DSL for scripting composite refactorings

5.2 The Domain-Specific Language

To allow fine control over the generation of refactoring commands and the way a refactoring command to be run, we have defined a small language for scripting composite refactorings. The DSL, as shown in Figure 6, is defined in Erlang syntax, using tuples and atoms. In the definition, PR denotes a primitive refactoring, and CR denotes a composite refactoring. We explain the definition of CR in more detail now, and some examples are given in Section 6.

- A primitive refactoring is, by definition, an atomic composite refactoring.
- $\{interactive,\ Qualifier,\ [PRs]\}$ represents a list of primitive refactorings that to be executed in an interactive way, that is, before the execution of every primitive refactoring, Wrangler asks the user for confirmation that he/she really wants that refactoring to be applied. The confirmation question is generated automatically by Wrangler.
- $\{repeat\_interative,\ Qualifier,\ [PRs]\}$ also represents a list of primitive refactorings to be executed in an interactive way, but different from the previous one, it allows user to repeatedly apply one refactoring, with different parameters supplied, multiple times. The user-interaction is carried out before each run of a primitive refactoring.
- $\{if\_then,\ \text{fun}() \rightarrow Cond \text{ end},\ CR\}$ represents the conditional application of CR, i.e. CR is applied only if $Cond$, which is an Erlang expression, evaluates to `true`. We wrap $Cond$ in an Erlang function closure to delay its evaluation until it is needed.
- $\{while,\ \text{fun}() \rightarrow Cond \text{ end},\ Qualifier,\ CR\}$ allows CR, which is generated dynamically, to be continually applied until $Cond$ evaluates to `false`. $Qualifier$ specifies whether the refactoring is to be applied atomically or not.

– $\{Qualifier, [\ CRs\ ]\}$ represents the composition of a list of composite refactorings into a new composite refactoring, where the qualifier states whether the resulting refactoring is applied atomically or not.

5.3 Tracking of Entity Names

In a composite refactoring, it is possible that a refactoring needs to refer to a program entity that might have be renamed by previous refactoring steps. Tracking the change of names statically is problematic given the dynamic nature of a refactoring process.Wrangler allows users to refer to a program entity through its initial name, i.e. the name of the entity before the refactoring process is started. For this purpose, we have defined a macro `?current`. An entity name, tagged with its category, wrapped in a `?current` macro tells Wrangler that this entity might have been renamed, therefore Wrangler needs to search its renaming history, and replaces the macro application with the entity's latest name. If no renaming history can be found for that entity, its original name is used.

6 Examples

In this section, we demonstrate how the DSL, together with Wrangler's template-based API, can be used to script large-scale refactorings in practice. The examples are written in a deliberately verbose way for clarity. In practice, a collection of pre-defined macros can be used to write the script more concisely.

Example 1. Batch clone elimination Wrangler's similar code detection functionality [13] is able to detect code clones in an Erlang program, and help with the clone elimination process. For each set of code fragments that are clones to each other, Wrangler generates a function, named as `new_fun`, which represents the *least general common abstraction* of the set of clones; the application of this function can be then used to replace those cloned code fragments, therefore to eliminate code duplication. The general procedure to remove such a clone in Wrangler is to copy and paste the function `new_fun` into a module, then carry out a sequence of refactoring steps as follows:

– Rename the function to some name that reflects its meaning.
– Rename the variables if necessary, especially those variable names in the format of `NewVar`i, which are generated by the clone detector.
– Swap the order of parameters if necessary.
– Export this function if the cloned code fragments are from multiple modules.
– For each module that contains a cloned code fragment, apply the 'fold expression against function definition' refactoring to replace the cloned code fragments in that module with the application of the new function.

The above clone elimination process can be scripted as a composite refactoring as shown in Figure 7. The function takes four parameters as input:

– the name of the file to which the new function belongs,

```
1 batch_clone_removal(File, Fun, Arity, ModNames) ->
2   ModName = list_to_atom(filename:basename(File, ".erl")),
3   {atomic,
4     [{interactive, atomic,
5       {refac_, rename_fun, [File, {Fun, Arity},
7                             {user_input, fun(_)->"New name:" end},
8                             false]}},
9      {atomic, {refac_, rename_var,
10       [File, current_fa({ModName,Fun,Arity}),
11        fun(V) -> lists:prefix("NewVar", V) end,
12        {user_input,
13          fun({_, _MFA, V})->io_lib:format("Rename ~p to:", [V]) end},
14        true]}},
15     {repeat_interactive, atomic,
16       {refac_, swap_args, [File, current_fa({ModName, Fun, Arity}),
17                             {user_input, fun(_, _)->"Index 1: " end},
18                             {user_input, fun(_, _)->"Index 2: " end},
19                             false]}},
20     {if_then, [ModName] /= ModNames,
21       {atomic, {refac_, add_to_export,
22                 [File, current_fa({ModName, Fun, Arity}), false]}}},
23     {non_atomic, {refac, fold_expr,
24       [{file, fun(FileName)->M=filename:basename(FileName, ".erl"),
25                               lists:member(M, ModNames)
26               end}, ?current({mfa, {ModName, Fun, Arity}}),1, false]}}
27     ]}.
29 current_fa({Mod, Fun, Arity}) ->
30   {M, F, A} = ?current({mfa, {Mod, Fun, Arity}), {F, A}.
```

Fig. 7. Batch Clone Elimination

– the name of the new function and its arity,
– and the name of the modules that contain one or more of cloned code fragments, which is available from the clone report generated by the clone detector.

We explain the script in detail now.

– **Lines 4-8.** This lets the user decide whether to rename the function. The new name is provided by the user if the function is to be renamed.
– **Lines 9-14.** This section generates a sequence of 'rename variable' refactorings to form an atomic composite refactoring. Making this sequence of 'rename variable' refactorings atomic means that we expect all the renamings to succeed, however, in this particular scenario, it is also acceptable to make it non-atomic, which means that we allow a constituent renaming refactoring to fail, and if that happens the user could redo the renaming of that variable after the whole clone elimination process has been finished.
The second argument of the generator specifies the function to be searched. An utility function `current_fa`, as defined between lines 29-30, is used to ensure

```
tuple_args(Prog) ->
  Pars = ?STOP_TD_TU(
           [?COLLECT(?T("f@(As1@@, Line, Col, As2@@) when G@@ -> B@@."),
             {api_refac:fun_def_info(f@),length(As1@@)+1}, true)], Prog),
  {non_atomic, lists:append(
    [{refactoring,tuple_args,[MFA,Index,Index+1]}||{MFA, Index}<-Pars])}.
```

Fig. 8. Batch tupling of function arguments

the latest name is referenced. The function on line 11 gives the searching criterion for the variables to be renamed, and in this case it requires that all the variables with a name starting with "NewVar" should be renamed. New variable names are provided by the user as shown by the third argument. Refactoring commands are generated lazily, as indicated by the last argument, to ensure that the variables to be renamed are correctly identified.

- **Lines 15-19.** The code here allows re-ordering of function parameters. The user can choose to re-order as many times as necessary, or not at all.
- **Lines 20-22.** This generates a refactoring that adds the new function to the export of the module only if the clones are from multiple modules.
- **Lines 23-26.** Finally, this code generates a list of 'fold expression against function definition' refactoring commands, one for each module listed in `ModNames`. We allow these refactorings to be composed in a non_atomic way so that the refactoring process will continue if a refactoring fails for some reason.

Example 2. Batch tupling of function arguments The example in Figure 8 shows how Wrangler's template-based API can help to create composite refactorings. This example searches an Erlang program for single-clause function definitions whose parameters include `Line` and `Col` next to each other, and generates a 'tuple arguments' refactoring command for each candidate found to put `Line` and `Col` into a tuple. `Prog` specifies the scope of the project, i.e, the places to search for Erlang files.

7 Implementation

Wrangler has been extended with another layer to support scripted composite refactorings, and this includes a number of extensions as follows.

- *An interpreter of the DSL language.* The interpreter takes a composite refactoring script as input, and generates refactoring commands that to be executed by the refactoring engine. Only one refactoring command is passed to the refactorer engine a time. Depending on the result returned and the context, the interpreter could continue to generate another refactoring command or ask for a rollback of the program to a particular point if an atomic refactoring fails.
- *Support for rolling back* a program to the starting point of an atomic composition when it fails. This is an extension of Wrangler's original *undo* mechanism.
- *A command generator* for each primary refactoring as discussed in Section 5.1.

– *A mechanism for recording each primitive refactoring command executed.* Wrangler records each primitive refactoring command executed and the result returned during the execution of a composite refactoring. This information provides valuable insights into the refactoring commands generated/executed, as well as the reason of failure if some refactorings fail during the execution.
– *A generic composite refactoring behaviour.* A *behaviour* in Erlang is an application framework that is parameterized by a *callback* module. The behaviour solves the generic parts of the problem, while the callback module solves the specific parts. In this spirit, a behaviour, named *gen_composite_refac*, has been implemented especially for composite refactorings. Two callback functions are specified by the behaviour. To implement a composite refactoring, the user needs to create a callback module, implement and export the callback functions. Once the callback module is compiled, the refactoring can be invoked and tested from the IDE. The result can be previewed before being committed/aborted. A composite refactoring can also be undone.

8 Related Work

The idea of composite refactorings was proposed by Opdyke [14], and investigated by Roberts [15]. This work focused on the derivation of a composite refactoring's preconditions from the pre- and postconditions of its constituent refactorings.This is non-trivial because when performing refactorings $R_1, R_2, ...R_n$ sequentially, performing R_i may establish, or invalidate, the preconditions of $R_j, j > i$. Ó Cinnéide [12] extends Roberts' approach in various ways including static manual derivation of pre- and postconditions for a composite refactoring.

`ContTraCT` is an experimental refactoring editor for Java developed by G. Kniesel, et. al. [11]. It allows composition of larger refactorings from existing ones. The authors identify two basic composition operations: AND- and OR-sequence, which correspond to the atomic and non-atomic composition described in this paper. A formal model based on the notion of *backward transformation description* is used to derive the preconditions of an AND-sequence.

While the above approaches could potentially detect a composition that is deemed to fail earlier, they suffer the same limitations because of the static nature of the composition. Apart from that, the derivation of preconditions and postconditions requires preconditions to be atomic and canonical. In contrast, our approach might be less efficient when a composite refactoring fails because of the conflict of pre-conditions, but it allows dynamic and lazy generations of refactoring commands, dynamic generation of parameter values, conditional composition of refactorings, rich interaction between users and the refactoring engine, etc. Our approach is also less restrictive on the design of underlying refactoring engine.

The refactoring API – described in a companion paper [10] – uses the general style of 'strategic programming' in the style of Stratego [16]. More detailed references to related work in that area are to be found in [10].

9 Conclusions and Future Work

Support for scripting composite refactorings in a high-level way is one of those features that are desired by users, but not supported by most serious refactoring tools. In this paper, we present Wrangler's DSL and API[10] based approach for scripting composite refactorings. We believe that being able to allow users to compose their own refactorings is the crucial step towards solving the imbalance between the limited number of refactorings supported by a tool and the unlimited possible refactorings in practice.

Our future work goes in a number of directions. First, we would like to carry out case studies to see how the support for user-defined refactorings is perceived by users, and whether this changes the way they refactor their code; second, we will add more composite refactorings to Wrangler, but also make Wrangler a hub for users to contribute and share their refactoring scripts; and finally, we plan to explore the application of the approach to HaRe, which is a refactoring tool developed by the authors for Haskell programs.

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
2. Roberts, D., Brant, J., Johnson, R.E.: A Refactoring Tool for Smalltalk. In: Theory and Practice of Object Systems, pp. 253–263 (1997)
3. JetBrains: JetBrains, http://www.jetbrains.com
4. Eclipse:: an open development platform, http://www.eclipse.org/
5. Li, H., Thompson, S., Reinke, C.: The Haskell Refactorer, HaRe, and its API. Electr. Notes Theor. Comput. Sci. 141(4), 29–34 (2005)
6. Li, H., et al.: Refactoring with Wrangler, updated. In: ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada (2008)
7. Lövei, L., et al.: Introducing Records by Refactoring. In: Erlang 2007: Proceedings of the 2007 SIGPLAN Workshop on Erlang Workshop. ACM (2007)
8. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Engineering 99 (2011)
9. Frenzel, L.: The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. Eclipse Magazine 5 (2006)
10. Li, H., Thompson, S.: A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, School of Computing, Univ. of Kent, UK (2011)
11. Kniesel, G., Koch, H.: Static composition of refactorings. Sci. Comput. Program. 52 (August 2004)
12. Cinnéide, M.O.: Automated Application of Design Patterns: A Refactoring Approach. PhD thesis, University of Dublin, Trinity College (2000)
13. Li, H., Thompson, S.: Incremental Clone Detection and Elimination for Erlang Programs. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 356–370. Springer, Heidelberg (2011)
14. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, Univ. of Illinois (1992)
15. Roberts, D.B.: Practical Analysis for Refactoring. PhD thesis, Univ. of Illinois (1999)
16. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. 72 (2008)

Author Index