

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Marsha Chechik Martin Wirsing (Eds.)

Fundamental Approaches to Software Engineering

12th International Conference, FASE 2009
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2009
York, UK, March 22-29, 2009
Proceedings

Volume Editors

Marsha Chechik
University of Toronto
Department of Computer Science
10 King's College Road, Toronto, ON, M5S 3G4, Canada
E-mail: chechik@cs.toronto.edu

Martin Wirsing
LMU Munich
Institute of Computer Science
Oettingenstr. 67, 80538 Munich, Germany
E-mail: wirsing@pst.ifi.lmu.de

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2, F.3, D.3, F.4, G.4

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

ISSN	0302-9743
ISBN-10	3-642-00592-6 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-00592-3 Springer Berlin Heidelberg New York

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

springer.com

© Springer-Verlag Berlin Heidelberg 2009
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12633227 06/3180 5 4 3 2 1 0

Foreword

ETAPS 2009 was the 12th 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 five conferences (CC, ESOP, FASE, FOSSACS, TACAS), 22 satellite workshops (ACCAT, ARSPA-WITS, Bytecode, COCV, COMPASS, FESCA, FInCo, FORMED, GaLoP, GT-VMT, HFL, LDTA, MBT, MLQA, OpenCert, PLACES, QAPL, RC, SafeCert, TAASN, TERMGRAPH, and WING), four tutorials, and seven invited lectures (excluding those that were specific to the satellite events). The five main conferences received this year 532 submissions (including 30 tool demonstration papers), 141 of which were accepted (10 tool demos), giving an overall acceptance rate of about 26%, with most of the conferences at around 25%. Congratulations therefore to all the authors who made it to the final programme! I hope that most of the other authors will still have found a way of participating in this exciting event, and that you will all continue submitting to ETAPS and contributing towards 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 and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

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

ETAPS 2009 was organised by the University of York 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 ERCIM, Microsoft Research, Rolls-Royce, Transitive, and Yorkshire Forward.

The organising team comprised:

Chair	Gerald Luetttgen
Secretariat	Ginny Wilson and Bob French
Finances	Alan Wood
Satellite Events	Jeremy Jacob and Simon O'Keefe
Publicity	Colin Runciman and Richard Paige
Website	Fiona Polack and Malihe Tabatabaie.

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

Vladimiro Sassone (Southampton, Chair), Luca de Alfaro (Santa Cruz), Roberto Amadio (Paris), Giuseppe Castagna (Paris), Marsha Chechik (Toronto), Sophia Drossopoulou (London), Hartmut Ehrig (Berlin), Javier Esparza (Munich), Jose Fiadeiro (Leicester), Andrew Gordon (MSR Cambridge), Rajiv Gupta (Arizona), Chris Hankin (London), Laurie Hendren (McGill), Mike Hinchey (NASA Goddard), Paola Inverardi (L'Aquila), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Stefan Kowalewski (Aachen), Shriram Krishnamurthi (Brown), Kim Larsen (Aalborg), Gerald Luetttgen (York), Rupak Majumdar (Los Angeles), Tiziana Margaria (Göttingen), Ugo Montanari (Pisa), Oege de Moor (Oxford), Luke Ong (Oxford), Catuscia Palamidessi (Paris), George Papadopoulos (Cyprus), Anna Philippou (Cyprus), David Rosenblum (London), Don Sannella (Edinburgh), João Saraiva (Minho), Michael Schwartzbach (Aarhus), Perdita Stevens (Edinburgh), Gabriel Taentzer (Marburg), Dániel Varró (Budapest), and Martin Wirsing (Munich).

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, and Springer for agreeing to publish the ETAPS proceedings. Finally, I would like to thank the Organising Chair of ETAPS 2009, Gerald Luetttgen, for arranging for us to hold ETAPS in the most beautiful city of York.

January 2009

Vladimiro Sassone, Chair
ETAPS Steering Committee

Preface

Software technology has become a driving factor for a rapidly growing range of products and services from all sectors of economic activity. At its core is a set of technical and scientific challenges that must be addressed in order to set the stage for the development, deployment, and application of tools and methods in support of the construction of complex software systems.

The International Conference on Fundamental Approaches to Software Engineering (FASE) – as one of the European Joint Conferences on Theory and Practice of Software (ETAPS) – focuses on those core challenges. FASE provides the software engineering research community with a forum for presenting theories, languages, methods, and tools arising from both fundamental research in the academic community and applied work in practical development contexts.

In 2009, FASE received 132 submissions: 123 regular papers and 9 tool papers. Each submission received an average of 3.1 reviews by technical experts from the Program Committee, helped by the external research community. Each paper was further discussed during a two-week “electronic” meeting. We wish to express our sincere thanks to all of the referees for the time, effort, and care taken in reviewing and discussing the submissions. The Program Committee selected 30 papers and 2 tool demonstrations – an acceptance rate of 24%. Accepted papers addressed topics such as model-driven development, modeling and specification, model analysis, testing, debugging, synthesis, security, and adaptation. The technical program was complemented by the invited lecture of Stephen Gilmore on “Scalable Analysis of Scalable Systems.”

FASE 2009 was held in York (UK) as part of the 12th edition of ETAPS. Arrangements were the responsibility of the local Organizing Committee, and the overall coordination of ETAPS was carried out by its Steering Committee. We would like to thank the Chairs of these committees, Gerald Luetzgen and Vladimiro Sassone, for the professional and friendly support with which we were provided throughout this process. The planning and coordination of the FASE series of conferences is the responsibility of EASST (European Association of Software Science and Technology). We would like to thank Reiko Heckel, as Chair of the Steering Committee of FASE in 2007, for having invited us to be Co-chairs of this 2009 edition. We wish all the best to the Co-chairs of the 2010 edition, Gaby Taentzer and David Rosenblum.

We used EasyChair for managing the paper selection process and for assembling the LNCS volume, and found this system very convenient. We are grateful to Springer for their helpful collaboration and assistance in producing this volume. As always, the real stars of the show are the authors of the papers, and especially the presenters. We would like to thank them all for having put so much effort into the papers and the presentations. As to the attendees of FASE

2009, we are sure that they were inspired by the technical and social quality of the program, and we are grateful for their participation.

January 2009

Marsha Chechik
Martin Wirsing

Organization

Programme Chairs

Marsha Chechik	University of Toronto (Canada)
Martin Wirsing	LMU Munich (Germany)

Programme Committee

Michel Bidoit	INRIA Saclay (France)
Ruth Breu	University of Innsbruck (Austria)
Jim Davies	University of Oxford (UK)
Juergen Dingel	Queen's University (Canada)
Schahram Dustdar	Vienna University of Technology (Austria)
Alexander Egyed	Johannes Kepler University Linz (Austria)
José Fiadeiro	University of Leicester (UK)
Harald C. Gall	University of Zürich (Switzerland)
Dimitra Giannakopoulou	RIACS/NASA Ames (USA)
Reiko Heckel	University of Leicester (UK)
Mats Heimdahl	University of Minnesota (USA)
Paola Inverardi	University of L'Aquila (Italy)
Alexander Knapp	University of Augsburg (Germany)
Angelika Mader	University of Twente (Netherlands)
TSE Maibaum	McMaster University (Canada)
Tiziana Margaria	University of Potsdam (Germany)
Fabio Massacci	University of Trento (Italy)
Stephan Merz	INRIA Nancy (France)
Peter Ölveczky	University of Oslo (Norway)
Richard Paige	University of York (UK)
Gregg Rothermel	University of Nebraska-Lincoln (USA)
Koushik Sen	University of California, Berkeley (USA)
Perdita Stevens	University of Edinburgh (UK)
Gabriele Taentzer	University of Marburg (Germany)
Ladan Tahvildari	University of Waterloo (Canada)
Tetsuo Tamai	University of Tokyo (Japan)
Sebastian Uchitel	University of Buenos Aires (Argentina)
Dániel Varró	Budapest University of Technology and Economics (Hungary)

External Reviewers

Aboulsamh, Mohammed	Hert, Matthias	Posse, Ernesto
Agreiter, Berthold	Holmes, Taid	Radjenovic, Alek
Arendt, Thorsten	Iris Groher	Rassadko, Natalya
Autili, Marco	Deepack Dughana	Reif, Gerald
Balogh, András	Istenes, Zoltán	Reiff-Marganec, S.
Bauer, Sebastian	Jacquemard, Florent	Rose, Louis
Berard, Beatrice	Jalbert, Nick	Ráth, István
Beszédes, Árpád	Jhala, Ranjit	Saidane, Ayda
Bielova, Nataliia	Joshi, Pallavi	Schall, Daniel
Bisztray, Dénes	Jung, Georg	Schneider, Gerardo
Bocchi, Laura	Jurack, Stefan	Schubert, Wolfgang
Boronat, Artur	Juszczyk, Lukasz	Schürr, Andy
Bouza, Amancio	Juvekar, Sudeep	Siahaan, Ida Sri Rejeki
Brooke, Phil	Jörges, Sven	Staats, Matt
Burnim, Jacob	Katt, Basel	Steffen, Bernhard
Chen, Feng	Khan, Tamim	Steffen, Martin
Chetali, Boutheina	Kolovos, Dimitrios	Stergiou, Christos
Chimiak-Opoka, Joanna	Kordon, Fabrice	Tavakoli Kolagari, R.
Choppy, Christine	Kövi, András	Tivoli, Massimo
Clavel, Manuel	Kyas, Marcel	Torrini, Paolo
Crichton, Charles	Langerak, Rom	Truong, Hong-Linh
Dalpiaz, Fabiano	Lapouchnian, Alexei	Tuosto, Emilio
de Lara, Juan	Li, Fei	Ure, Jenny
Di Benedetto, Paolo	Loew, Sarah	Voisin, Frederic
Di Ruscio, Davide	Manolescu, Ioana	Van Wyk, Eric
Donyina, Adwoa	Marché, Claude	Varró, Gergely
Drivalos, Nikos	Marincic, Jelena	Varró-Gyapay, Szilvia
Egger, Jeff	Markey, Nicolas	Vasko, Martin
Ermel, Claudia	Mehmood, Waqar	Villard, Jules
Escobar, Santiago	Memon, Mukhtiar	Voigt, Horst
Felderer, Michael	Mezei, Gergely	Wagner, Christian
Fluri, Beat	Minas, Mark	Wang, Chen-Wei
Ge, Xiaocheng	Montresor, Alberto	Weber, Michael
Ghezzi, Giacomo	Muccini, Henry	Welch, James
Giger, Emanuel	Neuhaus, Stephan	Wierse, Gerd
Grabe, Immo	Park, Chang-Seo	Wuersch, Michael
Gönczy, László	Park, Myung-Hwan	Xu, Kai
Habli, Ibrahim	Pelliccione, Patrizio	Yautsiukhin, Artsiom
Haddad, Serge	Pintér, Gergely	Zannone, Nicola
Hafner, Michael	Polack, Fiona	Zhou, Yu

Table of Contents

Scalable Analysis of Scalable Systems	1
<i>Allan Clark, Stephen Gilmore, and Mirco Tribastone</i>	

Model-Driven Development

Rewriting Logic Semantics and Verification of Model Transformations	18
<i>Artur Boronat, Reiko Heckel, and José Meseguer</i>	
Confluence in Domain-Independent Product Line Transformations	34
<i>Jon Oldevik, Øystein Haugen, and Birger Møller-Pedersen</i>	
Object Flow Definition for Refined Activity Diagrams	49
<i>Stefan Jurack, Leen Lambers, Katharina Mehner, Gabriele Taentzer, and Gerd Wierse</i>	
A Category-Theoretical Approach to the Formalisation of Version Control in MDE	64
<i>Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter</i>	

Synthesis and Adaptation

Controller Synthesis from LSC Requirements	79
<i>Hillel Kugler, Cory Plock, and Amir Pnueli</i>	
Interface Generation and Compositional Verification in JavaPathfinder	94
<i>Dimitra Giannakopoulou and Corina S. Păsăreanu</i>	
A Formal Way from Text to Code Templates	109
<i>Guido Wachsmuth</i>	
Context-Aware Adaptive Services: The PLASTIC Approach	124
<i>Marco Autili, Paolo Di Benedetto, and Paola Inverardi</i>	

Modeling

Synchronous Modeling and Validation of Priority Inheritance Schedulers	140
<i>Erwan Jahier, Nicolas Halbwachs, and Pascal Raymond</i>	

Describing and Analyzing Behaviours over Tabular Specifications Using (Dyn)Alloy	155
<i>Nazareno M. Aguirre, Marcelo F. Frias, Mariano M. Moscato, Thomas S.E. Maibaum, and Alan Wassynq</i>	

Testing and Debugging

Reducing the Costs of Bounded-Exhaustive Testing	171
<i>Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov</i>	
Logical Testing: Hoare-style Specification Meets Executable Validation	186
<i>Kathryn E. Gray and Alan Mycroft</i>	
Cross-Entropy-Based Replay of Concurrent Programs.....	201
<i>Hana Chockler, Eitan Farchi, Benny Godlin, and Sergey Novikov</i>	

Model Analysis

Control Dependence for Extended Finite State Machines	216
<i>Kelly Androutsopoulos, David Clark, Mark Harman, Zheng Li, and Laurence Tratt</i>	
Proving Consistency of Pure Methods and Model Fields	231
<i>K. Rustan M. Leino and Ronald Middelkoop</i>	
On the Implementation of @pre	246
<i>Piotr Kosiuczenko</i>	
Formal Specification and Analysis of Timing Properties in Software Systems	262
<i>Musab AlTurki, Dinakar Dhurjati, Dachuan Yu, Ajay Chander, and Hiroshi Inamura</i>	

Patterns

Formal Foundation for Pattern-Based Modelling	278
<i>Paolo Bottoni, Esther Guerra, and Juan de Lara</i>	
Problem-Oriented Documentation of Design Patterns	294
<i>Alexander Fülleborn, Klaus Meffert, and Maritta Heisel</i>	

Security

Certification of Smart-Card Applications in Common Criteria: Proving Representation Correspondences	309
<i>Iman Narasamdya and Michaël Périn</i>	

Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks	325
<i>Frank Hermann, Hartmut Ehrig, and Claudia Ermel</i>	

A Formal Connection between Security Automata and JML Annotations	340
<i>Marieke Huisman and Alejandro Tamalet</i>	

Queries and Error Handling

Algorithms for Automatically Computing the Causal Paths of Failures	355
<i>William N. Sumner and Xiangyu Zhang</i>	

Mining API Error-Handling Specifications from Source Code	370
<i>Mithun Acharya and Tao Xie</i>	

SNIFF: A Search Engine for Java Using Free-Form Queries	385
<i>Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen</i>	

Inquiry and Introspection for Non-deterministic Queries in Mobile Networks	401
<i>Vasanth Rajamani, Christine Julien, Jamie Payton, and Gruia-Catalin Roman</i>	

Tools (Demos) and Program Analysis

HOL-TESTGEN: An Interactive Test-Case Generation Framework	417
<i>Achim D. Brucker and Burkhart Wolff</i>	

CADS*: Computer-Aided Development of Self-* Systems	421
<i>Radu Calinescu and Marta Kwiatkowska</i>	

HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis	425
<i>Qichang Chen, Liqiang Wang, Zijiang Yang, and Scott D. Stoller</i>	

Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection	440
<i>Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen</i>	

Enhanced Property Specification and Verification in BLAST	456
<i>Ondřej Šerý</i>	

Finding Loop Invariants for Programs over Arrays Using a Theorem Prover	470
<i>Laura Kovács and Andrei Voronkov</i>	

Author Index	487

Scalable Analysis of Scalable Systems

Allan Clark, Stephen Gilmore, and Mirco Tribastone

The University of Edinburgh, Scotland

Abstract. We present a systematic method of analysing the scalability of large-scale systems. We construct a high-level model using the SRMC process calculus and generate variants of this using model transformation. The models are compiled into systems of ordinary differential equations and numerically integrated to predict non-functional properties such as responsiveness and scalability.

1 Introduction

Very often, our ability to build complex software systems outstrips our ability to plan and carry out rigorous analysis which predicts the behaviour of these systems under conditions of increasing load. This situation is unsatisfactory because it leads to systems being deployed in active use with no real assurance of graceful degradation of service as the user population grows. Because we cannot rely on them to provide service in times of greatest need, such systems are as unreliable in practice as ones which contain programming errors.

The crippling blow which strikes when trying to scale discrete-state models to represent user populations of significant size is the well-known problem of state-space explosion. The discrete-state representation demands memory in quantities which grow too quickly for us to be able to meet these demands for long. A bold alternative is to abandon discrete-state representations and take our models to the continuous-space world using fluid-flow analysis [1]. This allows us to represent and analyse large-scale systems with modest requirements on memory and time.

Modelling large populations of users is seldom the only difficulty which we encounter with large-scale systems. Scalable systems need to be resilient to changes in the underlying operational conditions. For this reason they are often structured with critical services replicated on several hosts in order for the system to continue to function when some of these hosts fail. It is very unusual indeed for all of the hosts to have identical performance profiles. It is instead quite common for them to be running different versions of the software services. Some will be running an older version, others the latest. Some sites will have disabled certain features, others not.

As if the above did not already give us enough challenges we also need to address the issue that large-scale systems are dynamic. Hosts providing one service may be taken down and redeployed to provide a different service. Some hosts will fail and might not be replaced if they were thought to be underused. New hosts will be sourced, purchased and brought online where the need is

In this paper we work with a process calculus which can be used for modelling problems such as these. The Sensoria Reference Markovian Calculus (SRMC), as described in [2], is a high-level modelling language which can be used for quantitative analysis of systems from the small scale to the large scale. Small-scale models in SRMC are mapped to continuous-time Markov chains and large-scale models are mapped to systems of differential equations. The SRMC language supports structured modelling via namespaces which accompany a novel mechanism for specifying uncertainty about binding. This is used to represent the inherent uncertainty about evaluation sites which is found generally in distributed computing and specifically in service-oriented computing where services are replicated across several hosts to provide scalability and robustness. Model transformations are used to capture changes in service administration leading to new hosts being commissioned or old ones being decommissioned.

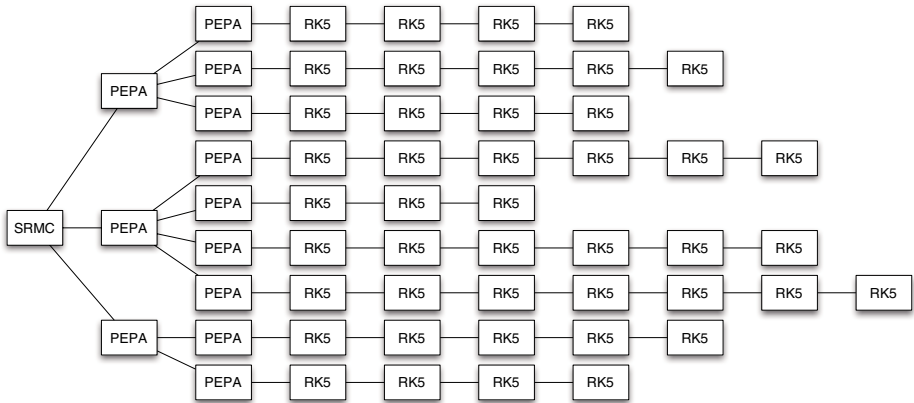


Fig. 1. The evaluation model. The single SRMC model in the illustration above gives rise to three PEPA models when possible bindings to services are considered. These three PEPA models become nine when model transformations are applied. Each of these nine models is evaluated between three and seven times in order to consider all of the possible assignments of rate values to rate parameters. This leads to forty-four systems of coupled ordinary differential equations in all and the same number of runs of a fifth-order Runge-Kutta (RK5) numerical integrator. These solve the initial value problem for each system of ODEs and give a time-series plot of the number of components of each type in the SRMC model as a function of time up to a finite time horizon. The results from these forty-four runs are combined to form the analysis results for the SRMC model.

simpler modelling language, Performance Evaluation Process Algebra (PEPA) [3,4]. Variants of each of these PEPA models are obtained by applying the transformations supplied. All models thus generated are evaluated for a range of numerical parameter values and the results from the individual runs are combined to deliver an evaluation of the model as a whole. Figure 1 illustrates the evaluation model for SRMC.

The novel contribution of the present paper is the use of model transformation to automatically generate a family of related models from a single SRMC source. In addition, this paper presents the first application of the fluid-flow analysis invented for PEPA to large-scale SRMC models. This latter innovation caused us to develop a supporting software infrastructure for aggregating the analysis results from the family of related models which we generate.

Structure of this paper: Section 2 presents our modelling concepts and introduces the ideas behind the process calculus which we use. In Section 3 we describe the features of the SRMC calculus and relate it to a simpler calculus without explicit support for dynamic binding, PEPA. In Section 4 we present the language of model transformations which we use. Our case study of a “virtual university” is presented in Section 5. The results are in Section 6. Software tool support is crucial for generating models and for managing the experimentation process; our software tools are described in Section 7. Section 8 describes related work and Section 9 presents conclusions.

2 Modelling Concepts

We are concerned here with non-functional properties of systems, specifically quantitative aspects such as performance. We investigate these properties using high-level models built from model components.

Model components are of two kinds, *behavioural* and *virtual*. A behavioural component cycles through a lifetime of timed activities offering sometimes only a single possible next activity and sometimes a choice between several alternatives. A virtual component does not perform activities but simply introduces a name into the model which we can use when querying the model later.

The kind of analysis which we will perform on the SRMC models in this paper tells us the expected number of behavioural components of every kind at all points in time. The expectation of the virtual components can be calculated from the expectations of the behavioural components by evaluating the defining expressions of the virtual components.

Because our interest is in quantitative modelling our models will contain rates and probabilities. Because we deal with large-scale systems with replicated components we also have *arrays* of behavioural components. For example, if C is a component then $C[15]$ is an array of fifteen independent copies of this component. We use these arrays to represent capacity (e.g. a pool of servers) or workload (e.g. a population of clients).

Our models will contain two kinds of numerical parameters, *certain* and *uncertain*. Certain parameters are bound to a single value: uncertain parameters

are bound to a set of possibilities. An uncertain parameter which is a probability might be bound to $\{0.4, 0.5, 0.6\}$, meaning that we should consider each of the elements of this set as a possible value for the parameter. Rates may be uncertain also, as may array sizes. Binding sites may be uncertain, and we choose one from a set of behavioural components.

We use model transformation to generate a family of related models from a single SRMC model. We think of these as plausible modifications of the original system which represent what the system might become after foreseeable reconfiguration or maintenance. Each of these generated models is “close” to the original model in the sense that they can be obtained by the application of a single transformation drawn from a set of possible transformations.

3 The Calculus

We work with the Sensoria Reference Markovian Calculus (SRMC), as described in [2]. SRMC allows modellers to structure their models using *namespaces*, separating components which have similar structure. To illustrate the SRMC language we will give an example of a simple system which consists of a process accessing one of two disks, *A* or *B*.

We first describe disk *A* which has occasional failures and has just two states, *failed* and *working*. When failed the disk must be repaired before more data can be read or written. Reads and writes are thought to be nearly equally likely: the probability that the I/O operation is a read is p_r . Failures occur somewhere between once every thousand disk operations ($p_f = 0.001$) and once every two hundred ($p_f = 0.005$). We will consider three sample values in this range. Rates λ and μ dictate the rates at which repairs and reads and writes take place. Disk *A* would be described in the SRMC syntax as shown below.

```
DiskA::{
  lambda = 0.3; mu = 1400; // rates
  p_r = { 0.4, 0.5, 0.6 }; // probability of a read
  p_f = { 0.001, 0.003, 0.005 }; // probability of failure

  Failed = (repair, lambda).Working;
  Working = (read, (1 - p_f) * p_r * mu).Working
    + (write, (1 - p_f) * (1 - p_r) * mu).Working
    + (fail, p_f * mu).Failed;
  Unavailable = [ Failed ];
};
```

This namespace has two behavioural components, **Failed** and **Working** and one virtual component **Unavailable**. The virtual component introduces the concept of “unavailability” to our model. In this case a disk is unavailable only if it has failed.

We can think of the above as a high-level schema representing nine concrete models differing only in the values assigned to the probabilities p_r and p_f . In the

first of the concrete models p_r has the value 0.4, and p_f is 0.001. In the ninth p_r is 0.6, and p_f is 0.005. In between all of the other possible assignments of values to p_r and p_f have been enumerated.

Disk B is slower than disk A . Failures occur more frequently and they take longer to repair. In addition disk B has a sleep mode which it enters to save power. The SRMC description is below.

```

DiskB::{
  lambda = 0.1; mu = 1200; // lower rates for the slower device
  p_r = { 0.4, 0.5, 0.6 }; // same probability of a read
  p_f = { 0.01, 0.03, 0.05 }; // higher probability of failure
  gamma = 0.001; delta = 0.001; // rates for sleep and wake

  Failed = (repair, lambda).Working;
  Working = (read, (1 - p_f) * p_r * mu).Working
    + (write, (1 - p_f) * (1 - p_r) * mu).Working
    + (fail, p_f * mu).Failed
    + (sleep, gamma).Offline;
  Offline = (wake, delta).Working;
  Unavailable = [ Failed + Offline ];
};

```

This too is a high-level schema representing nine concrete models. However, it introduces a different notion of unavailability. Here a disk is unavailable if it has failed or is offline. The virtual component `Unavailable` is defined to be the arithmetic sum of the number of disks which have failed plus the number of disks which are offline. Notice that the symbol “+” is overloaded in SRMC. In a behavioural component “+” denotes choice. In a virtual component “+” denotes arithmetic sum. Here we will add the expected number of failed disks and the expected number of offline disks to get the expected number of unavailable disks. Virtual components are syntactically distinguished because they consist of a defining expression enclosed in square brackets.

The disk which is in use in the system is either disk A or disk B . To describe this in SRMC we introduce another namespace, `Disk` which can stand for either A or B .

```

Disk::={ DiskA, DiskB };

```

The top-level composition of components in our example here introduces a computational process which reads and writes. The disk which is used is initially in its working state.

```

System = Process::Idle <read, write> Disk::Working;

```

In all this SRMC model represents eighteen simpler concrete models. In nine of these disk A is being used. In the other nine disk B is being used. In evaluating this SRMC model we separate out two groups of models. In the first of these disk A is being used, and disk B is not represented at all—the top-level composition evaluates to `Process::Idle <read, write> DiskA::Working` and

the entire `DiskB` namespace is discarded. In the second group of models disk *B* is being used and disk *A* is not represented at all—the top-level composition becomes `Process::Idle <read, write> DiskB::Working` and the entire `DiskA` namespace is discarded.

We then perform a *parameter sweep* across the possible assignments of values to model parameters in each group. This will require us to evaluate the version of the model with the main disk nine times, and the version of the model with the spare disk nine times also. Finally, we combine the results.

It is important to understand that the model configuration is fixed during model evaluation. That is, we will investigate the behaviour of the model up to a finite time horizon and during this time interval the model configuration will not change. This ability to divide the initial SRMC model up into a collection of simpler static models is an important factor in making our analysis scale to large models.

The simpler models which are generated in the parameter sweep which is performed after resolution of binding do not have namespaces and do not have uncertain parameters. The consequence of this is that they can all be expressed in Performance Evaluation Process Algebra (PEPA) [3,4]. PEPA has both a discrete-state stochastic Markovian semantics [5] and a continuous-state sure differential equation semantics [1]. We have considered the evaluation of SRMC models using the Markovian semantics for PEPA in an earlier paper [2] and we use the differential equation semantics here because our concern is with evaluating large-scale systems with many users and many replicated services.

Once all of the separate instances of the generated PEPA models have been analysed we collate the results into what is now a database of results. This database can be used to select and display various results from results relating to a single configuration or subset of all configurations to results pertaining to the entire results space. The latter allows such queries as: “What is the worst case scenario of long term expected number of unavailable disks” and “Give a listing of all configurations which fail to satisfy a given throughput of read operations”.

4 Model Transformation

The previous section relates the process of numerically evaluating an SRMC model. This included generating PEPA models after resolving dynamic binding. However, we also wish to investigate related models, reachable by an application of a model transformation, in order to incorporate possible changes which may occur. The grammar in Figure 2 defines transformation rules. The description presented here is sufficiently general that it may be applied at either the SRMC or the PEPA level.

A rule is specified by providing a *pattern* which should match some subcomponent of the model and a corresponding replacement, which is syntactically also a pattern. A transformation rule may contain *pattern variables* denoted by a question mark followed by a name. A pattern variable will match anything

<i>rule</i>	$:= \text{pattern} \Rightarrow \text{pattern}$	rules
<i>pattern</i>	$:= ?\text{name}$	variable
	<i>name</i>	named
	<i>pattern</i> < <i>activities</i> > <i>pattern</i>	cooperation
	<i>pattern</i> [<i>size</i>] ([<i>activities</i>])	array
<i>activities</i>	$:= \langle ?\text{name}, \rangle \text{name}^*$	concrete activities
<i>size</i>	$:= ?\text{name}$	variable
	<i>integer</i>	constant
	<i>size binop size</i>	binary op
<i>binop</i>	$:= + \mid - \mid \times \mid \div$	operators

Fig. 2. The grammar for transformation rules. The names which appear in the patterns are component identifiers. The names which appear in activity sets are activity names. The names which are used in size expressions denote the integer values which are used in dimensioning arrays of components.

which may appear in the given position, so when it occurs in the place of a component then it will match any component. If the replacement refers to a pattern variable then whatever was matched against is inserted at that place in the replacement.

A list of activities may contain a pattern variable together with several other concrete activity names. If this is the case the pattern variable is set to those names which are not given concretely, however we only match the given set of activities if the concrete activities are contained within the set.

The transformation $P \langle a \rangle Q \Rightarrow P \langle a, b \rangle Q$ adds activity *b* to the cooperation set. This rule uses no pattern variables and so will only match against the cooperation $P \langle a \rangle Q$. Generally pattern variables are used as $?Q$ here in the rule $P \langle a \rangle ?Q \Rightarrow P \langle a, b \rangle ?Q$. This will match the cooperation of *P* with any component including one which is itself a cooperation or a component array. Here we match against two cooperating arrays of *P* and *Q* components, remove one *P* and add a *Q* instead: $P[?m] \langle a \rangle Q[?n] \Rightarrow P[?m - 1] \langle a \rangle Q[?n + 1]$.

This style of pattern matching is used with *redeployable components* where we wish to analyse our system with different numbers of components deployed in each role. For example, file servers may be redeployed as web servers. However the pattern more commonly abstracts over the cooperation set as in the pattern: $P[?m] \langle ?a \rangle Q[?n] \Rightarrow P[?m - 1] \langle ?a \rangle Q[?n + 1]$.

Finally a common pattern is to remove some activities from a cooperation set. The following pattern matches any component cooperating with a *P* component over the activity *b* and removes it from the cooperation set allowing the *P* component (and the other cooperating component) to perform the activity *b* independently. Note though that any other activities in the cooperation set are preserved. This is written as $P \langle ?a, b \rangle ?Q \Rightarrow P \langle ?a \rangle ?Q$.

5 Case Study

To illustrate the above ideas we consider as an example a distributed e-learning and course management system. The system is to allow students to enrol in

courses even when studying remotely. One of the quantitative issues of concern here is whether or not the system will scale well enough to cope with increased demand from a larger population of student users.

5.1 The Servers

In this example we consider a fictional virtual university which has two university sites in the University of Edinburgh and Imperial College, London. Each site has an HTTP server where students can download course materials, multimedia content, and other courseware. Each has an FTP server where students can upload project materials and coursework for assessment. The HTTP and FTP servers may fail independently and, because each is running other services as well, availability of the servers varies.

```
Edinburgh::{
  mu = 0.0001; gamma = 0.125; // rates of fail and repair
  avail = {0.6,0.7,0.8,0.9,1.0}; // availability of the server
  phi = 10.0; psi = 7.0; // rates for download and upload

  // The HTTP server
  Http::{
    Idle = (download, avail * phi).Idle
          + (fail, mu).Broken;
    Broken = (repair, gamma).Idle;
  };

  // The FTP server
  Ftp::{
    Idle = (upload, avail * psi).Idle
          + (fail, mu).Broken;
    Broken = (repair, gamma).Idle;
  };
};
```

The servers at Imperial are similar in functionality to those in Edinburgh however they differ in their performance characteristics, specifically with respect to the rates at which failures occur and the rates at which downloads and uploads occur.

```
Imperial::{
  mu = 0.006; gamma = 0.125; // failures are more likely
  avail = {0.6,0.7,0.8,0.9,1.0}; // availability is the same
  phi = 20.0; psi = 15.0; // download and upload are faster

  // The HTTP server
  Http::{
    Idle = (download, avail * phi).Idle
```



```

        + (fail, mu).Broken;
    Broken = (repair, gamma).Idle;
};

// The FTP server
Ftp::{
    Idle = (upload, avail * psi).Idle
        + (fail, mu).Broken;
    Broken = (repair, gamma).Idle;
};
};

```

Of course further servers may be added, in our results given in Section 6 we added a further server which fairly services both HTTP and FTP requests at the same rate.

5.2 The Clients

We characterise different types of user of the system. The first, Harry, connects relatively frequently, and uploads or downloads once each session.

```

Harry::{
    connect_rate = { 0.01, 0.02, 0.03 };
    disconnect_rate = 1.0;
    download_rate = { 0.01, 0.02, 0.03 };
    upload_rate = 1.0;

    Idle = (connect, connect_rate / 2).Upload
        + (connect, connect_rate / 2).Download;
    Upload = (upload, upload_rate).Disconnect;
    Download = (download, download_rate).Disconnect;
    Disconnect = (disconnect, disconnect_rate).Idle;
    Uploading = [ Upload ];
    Downloading = [ Download ];
    Inservice = [ Uploading + Downloading ];
};

```

The second type of user, Sally, connects relatively infrequently and downloads more than uploading.

```

Sally::{
    connect_rate = { 0.009, 0.0095, 0.01 };
    disconnect_rate = 0.5;
    download_rate = { 0.01, 0.02, 0.03 };
    upload_rate = 0.2;

```

```

Idle = (connect, connect_rate / 3).Upload
      + (connect, connect_rate / 3).Download1
      + (connect, connect_rate / 3).Download2 ;
Upload = (upload, upload_rate).Disconnect;
Download1 = (download, download_rate).Download2;
Download2 = (download, download_rate).Disconnect;
Disconnect = (disconnect, disconnect_rate).Idle;

Uploading = [ Upload ];
Downloading = [ Download1 + Download2 ];
Inservice = [ Uploading + Downloading ];
};

```

As with the servers further clients may be added as required, in our results we added a further client who was likely to perform either two uploads or two downloads with each connection.

5.3 The Model Configuration

The clients are either like Harry or like Sally.

```
Client ::= { Harry, Sally };
```

The HTTP server which is used is either the Edinburgh server or the Imperial server, and analogously for the FTP servers.

```
Http ::= {Edinburgh::Http, Imperial::Http};
Ftp ::= {Edinburgh::Ftp, Imperial::Ftp };
```

There are between three and six servers at each site. Each server has an allocation of twenty threads to offer. There is a very large pool of clients.

```
servers = {3, 4, 5, 6};
threads = 20;
clients = 100000;
```

The entire model consists of an array of clients uploading and downloading from an array of servers, with multiple threads on each.

```
Client::Idle[clients] <download, upload>
  ( Http::Idle[servers * threads] ||
    Ftp::Idle[servers * threads] )
```

5.4 Transformations

The transformations used in this model relate to the redeployment of a server. This means that a server currently being used as an HTTP server can be redeployed as an FTP server or vice-versa. We use this to test a particular service configuration's ability to adapt to varying client behaviours. Recall that one configuration is a set of name space choices. In our given example this means that

one particular configuration is a selection of a university to supply the HTTP server, a university to supply the FTP server and finally a client representing average client behaviour. Here we call a service configuration the part of the configuration which specifies the two servers in use.

One such configuration is *Edinburgh*, *Imperial* and *Harry*. Suppose we measure this and we find that the average number of waiting clients is acceptably low, but in the configuration *Edinburgh*, *Imperial* and *Sally* where we have changed the client behaviour the system behaves poorly. We may see that it behaves poorly because the uploading clients are not serviced fast enough. In practice if such a situation arose one response would be to redeploy one of the HTTP servers as an FTP server. For each configuration, which corresponds to a single PEPA model, we use transformations to obtain three PEPA models; the base configuration model, the model with one HTTP server redeployed as an FTP server and the model with one FTP server deployed as an HTTP server. The first transformation would tell us how the system behaves in the configuration *Edinburgh*, *Imperial* and *Sally*, with one HTTP server redeployed as an FTP server. Without this transformation we noted that the performance was poor because uploading clients were not serviced often enough, with this transformation though it may be that the system performs satisfactorily. In this case we would know that the system configuration is robust with respect to changing client behaviour and thus the system may be recommended. The transformation rule used to obtain this is:

```
Http::Idle[?m * threads] || Ftp::Idle[?n * threads] ==>
  Http::Idle[(?m - 1)*threads] || Ftp::Idle[(?n + 1)*threads]
```

and similarly for redeploying in the reverse direction.

5.5 Lazy Results

In the example case study we have used a strict semantics for results generation, that is; all the results were computed before any were viewed by the user. However an alternative semantics allows the user to generate only those results which are inspected. In the previous example scenario in which the configuration: *Edinburgh*, *Imperial* and *Sally* under-performed because the uploading clients were not serviced enough we would be unlikely to inspect the result obtained by applying the transformation which redeploys an FTP server as an HTTP server since this would only exacerbate the situation. In this case we could avoid computing all the results associated with that particular model – there are more than one set of results for each model because the model is solved several times corresponding to the varying rates used within that model instance. Lazy results can help in the development of a model since often only a small set of the results are viewed between each update of the entire model. However lazy results are limited to local observations, that is results which only depend on a subset of the entire results space. Many results such as overall average response-time or worst-case scenario depend on all of the results and in these case lazy results will behave in the same way as strict results.

6 Results

We analysed our model on a typical desktop computer. In our first run we analysed over 1000 configurations within fifteen minutes. In our second run we increased the variability of the rates involved and performed over 4500 in six hours. Figure 3 provides a selection of the graphs which were generated from the second run.

The main result metric we used was the measure of the expected number of clients currently in the process of uploading, downloading or either of the two. We term this *in service*. This gives us a measure of how many are generally competing for the resources of the servers and – in combination with the throughput of download/upload events – how long each client can expect to wait to be served.

Graph (a) plots for a single configuration the number of clients in service against time. Each line corresponds to a single permutation of the variable rates which are used with the particular configuration (*Edinburgh*, *Edinburgh* and *Harry*), without any redeployment of servers. We see that in the long run the expected number of clients waiting varies from just over thirty thousand to over eighty thousand of a total of one hundred thousand users. This shows that there is substantial variation in system performance caused only by varying the rates within a single configuration.

In graphs, (b), (c) and (d) respectively we have plotted for a single configuration the effect that redeploying a server has on the number of clients either uploading, downloading or either of the two. These three graphs all refer to the first configuration (*Edinburgh*, *Edinburgh* and *Harry*). In the graphs (b) and (c) it is shown that redeployment gives a large benefit to the recipient of the redeployment. So in the case that an HTTP server is redeployed as an FTP server the number of currently uploading clients (in graph (b)) falls almost to zero. Whereas redeployment of an FTP server reduces the number of currently downloading clients (graph (c)). In this particular configuration we note that redeploying an FTP server has only a small benefit to the number of downloaders since this is low anyway (the red/middle line in graph (c)). However redeploying an HTTP server has a large benefit in reducing the number of uploaders (the blue/bottom line in graph (b)). Finally from graph (d) we see that either redeployment increases the total number of in service clients for this particular configuration. However this is not always the case as in graph (e) in which the configuration (*Edinburgh*, *Imperial* and *Sally*) is considered the number of overall in service clients reduces when an FTP server is redeployed.

The graphs (f), (g) and (h) plot the experiment number against the total number of clients in service, uploading and downloading respectively. In each graph the experiments which correspond to an initial configuration (without any redeployment of servers) are shown in red on the left. The experiments in which the configuration has been altered by redeploying an FTP server as an HTTP server are plotted in green (in the middle) and finally those in which an HTTP server has been redeployed as an FTP server are plotted on the right in blue. From graph (f) we see that the redeployment of a server has a relatively

minor effect on the total number of clients in service. Redeploying an FTP server does worsen both the worst case scenario and the best case scenarios and in general decreases the performance by our metric – though not for all individual configurations as we have already seen from graph (e). Redeploying an HTTP server actually gives better worst and best case scenarios but performance in the general case is mildly impaired. From graphs (g) and (h) we see that the redeployment of a server generally has the intuitive effect. That is if we redeploy an FTP server the number of clients downloading decreases while the number of clients uploading increases and vice-versa. However it is encouraging to note that there are some configurations in which the redeployment of an FTP server still results in very low numbers of uploaders and conversely the redeployment of an HTTP server still results in very low numbers of downloaders.

7 Software Support

The software support which is available for SRMC has its basis in the parameter sweep developed for PEPA [6] and implemented in IPC [7]. We use the Pepato library of the PEPA Eclipse Plug-in Project [8] to compile our generated PEPA models to systems of ODEs. The Pepato library gives us access to differential equation integrators [9,10]. The model transformation engine used for our process algebra models was developed for the present work. The software used here is available for download from <http://groups.inf.ed.ac.uk/srmc>.

8 Related Work

We have considered the distributed e-learning and course management system example previously. In [2] we considered the problem of how service-level agreements can be evaluated for service-oriented systems at all. In [11,12] we considered the scalability of such a system in the absence of possible modifications as generated through model transformation.

Considering the method of evaluation rather than the example, the quantitative modelling approaches which seem similar to ours in spirit are PEPA itself, stochastic Petri nets, and PRISM model-checking. We consider each of these in turn, giving attention to the way in which parameters can be varied, the type of results which can be computed, and the query languages which are available to query models.

PEPA Eclipse Plug-in. The analysis paradigm for PEPA as supported by the PEPA Eclipse Plug-in [8] is that we have a single model with rates which can be varied using the experimenter of the Plug-in. Models can be evaluated to determine utilisation, throughput and many other criteria. Queries are expressed using state filters.

IPC. The paradigm for PEPA supported by IPC [7] is that we have a single model and rates defined in the model can be overridden at evaluation time. Models can be evaluated to determine transient or steady-state behaviour.

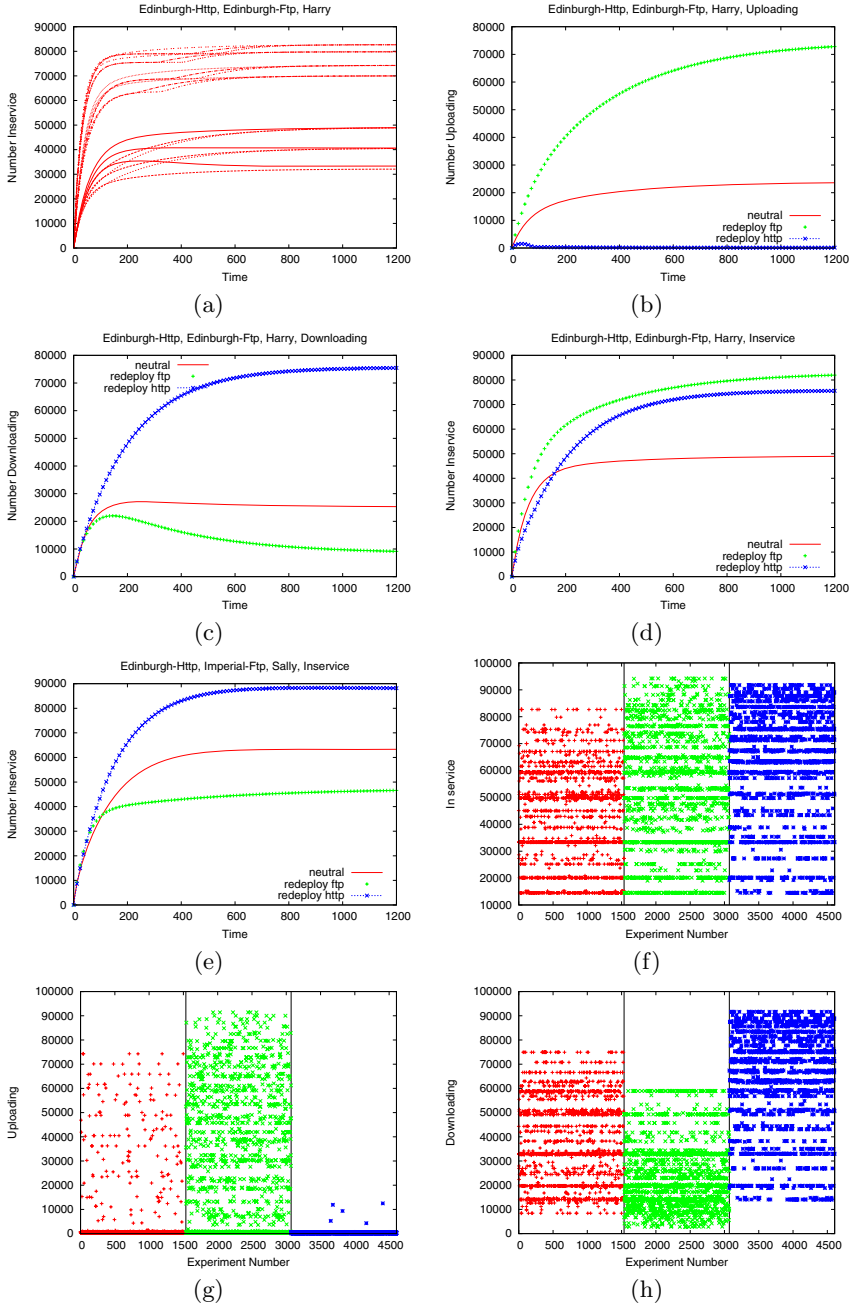


Fig. 3. Selection of generated graphs

Passage-time quantiles are computed. Queries are expressed in the language of eXtended Stochastic Probes (XSP) [13].

PIPE and PQE. Generalised Stochastic Petri net (GSPN) models are supported by the PIPE editor [14]. Here a single model with fixed rates is evaluated against many criteria. Queries are expressed as Performance Trees [15] built graphically in the Performance Query Editor (PQE) module for PIPE.

PRISM. The PRISM model-checker supports a language of reactive modules [16] together with a rich reward language. Queries are expressed in Continuous Stochastic Logic (CSL) [17]. Experimentation is supported by leaving constants in the model or the CSL formula undefined until the time of evaluation.

9 Conclusions

We have placed the emphasis here on modelling rather than models, and on evidence rather than fact. We start from a position of uncertainty about the configuration of the computational framework and consider a family of related models in an attempt to understand the sweep of possibilities. We believe that this position is a realistic one. In service-oriented computing the critical services are replicated across hosts so we have a choice of service instances, possibly modified by system administrators, performing at different rates. For these reasons, the SRMC calculus provides support for namespace selection, model transformation, and parameter sweep.

Model transformations work at the level of the PEPA model and can therefore be deployed as a means of analysing possible changes in one particular model. In this work we have used the ability to specify generic transformations in order to apply such a transformation to a large range of PEPA models generated from our SRMC model.

Collating results allows us to answer questions of a general nature about all configurations. In our case study there were few general statements that could be made because the transformations and rate variations provided substantial changes in performance. This is in itself a useful fact, that the system under consideration is sensitive to modifications in the running conditions. However we were able to ascertain worst and best case scenarios for the average number of currently downloading and currently uploading clients. Of particular note was that the number of waiting uploading clients can be all but eliminated through the redeployment of an HTTP server regardless of the initial configuration. We also saw that in general the redeployment of a server would most often benefit one kind of user (say uploaders) at the cost of denying some service capacity to the other (downloaders). This was also shown by the fact that the number of clients in service either as an uploader or a downloader was relatively less affected by redeployment of servers.

Model evaluation must be rapid to support the investigation of many alternative models. Fluid-flow analysis allows us to obtain meaningful results from a large family of models, at low computational cost. The result from a fluid-flow

analysis performed by numerically integrating a system of ordinary differential equations is precise and definitive. We need to evaluate our system of ODEs only once, not many times as would be needed for simulation models. This supports the scalability of our analysis: running many models once is feasible, running many models many times is less so.

Because fluid-flow analysis does not use a representation of the discrete state-space of the system we are not crippled by the state-space explosion problem, unlike any analysis which is based on continuous-time Markov chains. By building on the foundation provided by fluid-flow analysis and creating software tools which automate the generation of models by model transformation and the quantitative evaluation of these we hope to provide a strong basis for scalable analysis of scalable systems.

Acknowledgements. The authors are supported by the EU FET-IST Global Computing 2 project SENSORIA (“Software Engineering for Service-Oriented Overlay Computers” (IST-3-016004-IP-09)).

References

1. Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, Torino, Italy, pp. 33–43. IEEE Computer Society Press, Los Alamitos (2005)
2. Clark, A., Gilmore, S., Tribastone, M.: Service-level agreements for service-oriented computing. In: Montanari, U., Corradini, A. (eds.) Proceedings of the 19th International Workshop on Algebraic Development Techniques (WADT 2008), Pisa, Italy. LNCS. Springer, Heidelberg (2008)
3. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
4. Hillston, J.: Tuning systems: From composition to performance. The Computer Journal 48(4), 385–400 (2005); the Needham Lecture paper
5. Hillston, J.: Process algebras for quantitative analysis. In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005), pp. 239–248. IEEE Computer Society Press, Chicago (2005)
6. Clark, A., Gilmore, S.: Evaluating quality of service for service level agreements. In: Brim, L., Leucker, M. (eds.) Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems, Bonn, Germany, pp. 172–185 (August 2006)
7. Clark, A.: The ipclib PEPA Library. In: Harchol-Balter, M., Kwiatkowska, M., Telek, M. (eds.) Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST), pp. 55–56. IEEE, Los Alamitos (2007)
8. Tribastone, M.: The PEPA Plug-in Project. In: Harchol-Balter, M., Kwiatkowska, M., Telek, M. (eds.) Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST), pp. 53–54. IEEE, Los Alamitos (2007)
9. Ascher, U.M., Ruuth, S., Spiteri, R.: Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. Applied Numerical Mathematics 25(2-3), 151–167 (1997)
10. Dormand, J., Prince, P.: A family of embedded Runge-Kutta formulae. Journal of Computational and Applied Mathematics 6(1), 19–26 (1980)

11. Gilmore, S., Tribastone, M.: Evaluating the scalability of a web service-based distributed e-learning and course management system. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 214–226. Springer, Heidelberg (2006)
12. Bravetti, M., Gilmore, S., Guidi, C., Tribastone, M.: Replicating web services for scalability. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 204–221. Springer, Heidelberg (2008)
13. Clark, A., Gilmore, S.: State-aware performance analysis with eXtended Stochastic Probes. In: Thomas, N., Juiz, C. (eds.) EPEW 2008. LNCS, vol. 5261, pp. 125–140. Springer, Heidelberg (2008)
14. Bonet, P., Lladó, C., Puijaner, R., Knottenbelt, W.J.: PIPE v2.5: A Petri net tool for performance modelling. In: 23rd Latin American Conf. on Informatics (CLEI 2007) (September 2007)
15. Suto, T., Bradley, J.T., Knottenbelt, W.J.: Performance Trees: A new approach to quantitative performance specification. In: MASCOTS 2006, 14th International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 303–313 (August 2006)
16. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
17. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Software Eng.* 29(7), 1–18 (2003)

Rewriting Logic Semantics and Verification of Model Transformations

Artur Boronat¹, Reiko Heckel¹, and José Meseguer²

¹ Department of Computer Science, University of Leicester
{aboronat,reiko}@le.ac.uk

² Department of Computer Science, University of Illinois at Urbana-Champaign
meseguer@uiuc.edu

Abstract. Model transformations are used in model-driven development for mechanizing the interoperability and integration among modeling languages. Due to the graph-theoretic nature of models, the theory of graph transformation systems and its technological support provide a convenient environment for formalizing and verifying model transformations, which can then be used for defining the semantics of model-based domain-specific languages. In this paper, we present an approach for formalizing and verifying QVT-like transformations that reuses the main concepts of graph transformation systems. Specifically, we formalize model transformations as theories in rewriting logic, so that Maude’s reachability analysis and model checking features can be used for verifying them. This approach also provides a new perspective on graph transformation systems, where their formal semantics is given in rewriting logic. All the ideas presented are implemented in MOMENT2. In this way, we can define formal model transformations in the Eclipse Modeling Framework (EMF) and we can verify them in Maude. We use a model of a distributed mutual exclusion algorithm to illustrate the approach.

Keywords: Model and graph transformations, MOF, QVT, rewriting logic, reachability analysis, LTL model checking, Maude.

1 Introduction

Model transformations are used in model-driven development for mechanizing the interoperability and integration among modeling languages. Due to the graph-theoretic nature of models, the theory of graph transformation systems and its technological support provide a convenient environment for formalizing and verifying model transformations [1], which can then be used for defining the semantics of model-based domain-specific languages [2]. In this work, we provide an executable formalization of QVT-like model transformations for MOF metamodels in rewriting logic, where such transformations can be executed and analyzed by model checking of invariants and of temporal logic properties.

This work should be placed within the context of current formal and informal approaches to model transformations, and can be viewed as a contribution

to bringing such formal and informal approaches considerably closer within the MOF standard [3]. Among the various informal approaches for model transformations (see, e.g., [4]), the QVT standard [5] is probably one of the most widely accepted and has the advantage of extending MOF. However, tool support for QVT is currently only partial and the support for analyzing model transformations is still very limited. Among the formal approaches, the most widely used are based on graph transformations [6], where just a subset of the MOF modeling constructs can be directly dealt with. To the best of our knowledge, the Tiger EMF Transformation tool (EMT) [7] is the single representative tool providing termination and confluence analysis for model transformations. There is no tool support yet for model checking model transformations.

A model transformation β can be either *endogenous* to a given metamodel, so that if $M : \mathcal{M}$, then $\beta(M) : \mathcal{M}$, or *exogenous*, i.e., it can transform a model $M : \mathcal{M}$ into a corresponding model $\beta(M) : \mathcal{M}'$ in a target metamodel $\mathcal{M}'$ that need not be equal to $\mathcal{M}$ [8]. Furthermore, endogenous or exogenous transformations can be either *functional*, so that a single transformed model $\beta(M)$ is obtained, or *relational*, so that $\beta(M)$ is not unique, but belongs to a set of models in $\mathcal{M}'$ obtained from a single model M in $\mathcal{M}$. In this paper we focus on endogenous, relational transformations although it extends to the exogenous case easily.

At the semantic level we show how the algebraic semantics for MOF in MOMENT2 [9,10] is extended to a rewriting logic semantics [11] of model transformations. This rewriting logic semantics is the algebraic counterpart of graph transformation systems, in the sense that: (i) models viewed as graphs correspond to the same models viewed as terms of an algebraic data type representing a given metamodel; and (ii) graph transformation rules correspond to rules in rewrite theories over these algebraic representations. Models are represented by terms that satisfy structural semantic properties as explained in Section 3.1, which are preserved by model transformations. Conceptually, this work provides interesting new connections between graph grammars and rewriting logic, not just for standard graphs, but for graphs corresponding to models: on the one hand, the fact that node attributes in graphs often contain data belonging to sophisticated algebraic data types is seamlessly supported by the rewriting logic approach while most of other approaches leave attribute values out of the formal framework; and on the other hand, rewriting logic's crucial distinction between equations and rules can be transferred to the world of graph grammars. This can be used as a powerful state reduction technique for model checking purposes.

At a more pragmatic level, a second important contribution is bringing the model transformation approaches based on graph grammars and informal approaches extending MOF, such as QVT, significantly closer; and providing tool support for defining model transformations, executing them, and model checking their properties. To the best of our knowledge, MOMENT2 is the first formal approach and tool supporting QVT-like model transformations and their formal analysis through model checking techniques.

In Section 2 a model of a distributed mutual exclusion algorithm is presented to illustrate the definition and verification of model transformations; in Section 3

we introduce rewriting logic prerequisites and the algebraic semantics of MOF metamodels in MOMENT2; in Section 4 the semantics of the MOMENT2 model transformation language is given; in Section 5 model checking facilities are illustrated; in Section 6 we discuss related work, conclusions and future work.

2 Modeling a Distributed MUTEX Algorithm

A distributed mutual exclusion algorithm is used in operating systems and databases to ensure that a resource is never used by more than one process at a time. However, each request of a process for a resource must eventually be granted without running into a deadlock. In this section, we provide a metamodel-based version of a mutual exclusion algorithm [12].

In our approach, a model that conforms to the metamodel in Fig. 1.(a) represents the state of the system. Imagine this as the metamodel of a domain-specific language for visualizing and configuring processes and resources, where rules specify policies for how resources shall be assigned. In this metamodel, the *Ring* class contains some properties that are used for supporting *fairness* in the application of rules in Section 5. Fig. 1.(b) provides a model describing a deadlock state. Fig. 1.(c) provides an initial state model checked in Section 5.

We use the concrete syntax that was used in [12] for presenting the algorithm, where *processes* are drawn as black nodes and *resources* as light boxes. An edge from a process to a resource models a *request*. A solid edge in the opposite direction shows that the resource is currently *held by* the process. A dashed edge from a resource to a process asks the process to *release* the resource. Fig. 2 provides two sets of rules describing: (i) the mutual exclusion algorithm (ME) and (ii) the distributed deadlock detection mechanism (DDD).

Mutual exclusion (ME). The system state is a cyclic list of processes linked by the *next* reference. For each resource there is a *token*, represented by an edge with a white flag, which is passed from process to process along the ring. If a process wants to use a resource, it waits for the corresponding token. *Mutual*

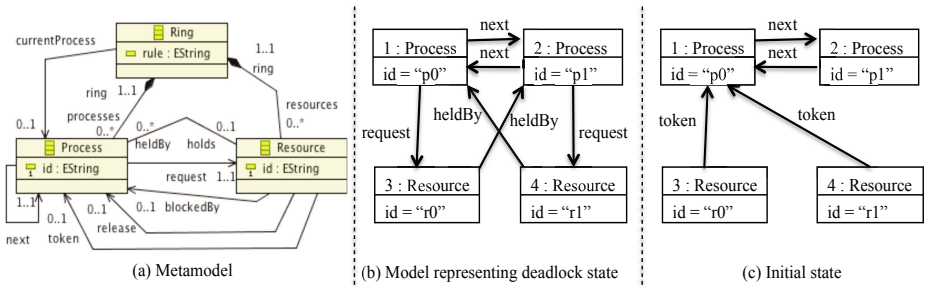


Fig. 1. (a) Metamodel $\mathcal{M}$. (b) Model representing a deadlock state. (c) Model representing an initial state for model checking purposes.

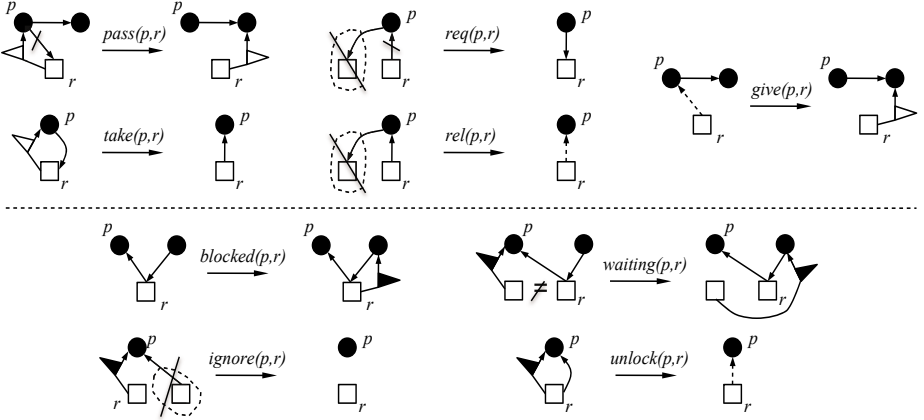


Fig. 2. Algorithms for mutual exclusion and deadlock detection

exclusion is ensured because there is only one token for each resource in the system. Among the ME rules, $pass(p, r)$ describes that a process having the token may pass it to the next process in the ring, provided that it does not have a request on the corresponding resource. This negative application condition is visualized by the crossed-out request edge from the process to the resource. If a process wants to use a resource, it may generate a request. This is modeled by the rule $req(p, r)$, which is only applicable if the process does not have any requests yet, and if the particular resource is not used already by this process. If a process receives a token and there is a request for the resource, the process will choose the rule $take(p, r)$ replacing the token and the request by a *heldBy* edge from the resource to the process. When it has finished its task, the process may release its resource and give the token to the next process using $rel(p, r)$ and $give(p, r)$. This will happen only when there are no pending requests for r , which is modeled by a negative application condition at $rel(p, r)$.

Distributed deadlock detection. In a model representing a state, a deadlock is represented as a cycle of *request* and *heldBy* edges. The *distributed deadlock detection* uses *blocked* messages, represented by edges with a black flag from a resource to a process, in order to detect cyclic dependencies. The $blocked(p, r)$ rule detects when a process requests a resource already held by another process and the rule $unlock(p, r)$ ensures that the resource will be released at some point.

We use the union of these two algorithms to show how relational in-place model transformations can be defined and model checked in MOMENT2. On the one hand, we model check the safety property *MUTEX-safe*: *a resource cannot be held by two different processes*, by means of reachability analysis of an invariant defined with a model pattern. On the other hand, we verify the liveness property *MUTEX-live*: *each request of a resource by a process will eventually be granted*, by means of model checking of LTL properties.

3 Preliminaries: Rewriting Logic and MOMENT2

The key point of rewriting logic is to provide a general and flexible logical framework for concurrent systems, which are specified as rewrite theories so that their concurrent computation exactly corresponds to deduction by rewriting [11]. Specifically we specify a concurrent system as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ where: (i) (Σ, E) is an *equational theory*, in some variant of equational logic, that specifies the *system states* as elements of the initial algebra $T_{(\Sigma, E)}$ associated to (Σ, E) (see, e.g., [13]), and where (ii) R is a set of *rewrite rules* that describe in a parametric way all the one-step *concurrent transitions* in the system.

The underlying equational logic can be unsorted, many-sorted, order-sorted, or the more general *membership equational logic* (MEL) [13], which is the variant we adopt in this paper. It has the advantage of supporting expressive sorts, subsorts, and partiality. Its atomic sentences include not only equations $t = t'$, but also memberships $t : s$, stating that term t has sort s . For our purposes in this paper MEL has the additional advantage, as we further explain in Section 3.1, of providing the algebraic semantics for MOF metamodels on which the MOMENT2 tool is based. That is, given a MOF metamodel $\mathcal{M}$, its algebraic semantics is a MEL theory $\mathbb{A}(\mathcal{M})$. The models M conformant with $\mathcal{M}$ then appear as elements of the initial algebra $T_{\mathbb{A}(\mathcal{M})}$. This gives us what we might call the *static semantics* of models in $\mathcal{M}$. The point of considering model transformations is that they can specify a *dynamic semantics* for models, in which each model M is now viewed as a *state* (for example, a dynamic software architecture configuration), and model transformations become *state transitions*. It is precisely in this passage from the static to the dynamic semantics of models that rewriting logic is particularly helpful: as we explain in Section 4 a model transformation specifying dynamic model changes for models of a metamodel $\mathcal{M}$ can be precisely characterized as a *rewrite theory* extending the MEL theory $\mathbb{A}(\mathcal{M})$ that specifies the static semantics.

All this is of more than theoretical interest, because there are several high-performance implementations of rewriting logic. In particular, the Maude implementation [14] supports not only execution, but also verification of invariants and model checking of LTL properties, and has various other formal tools for verification purposes. This is heavily exploited in the MOMENT2 tool, where, as we further explain in Section 5, Maude is used as the underlying engine to model check invariants and LTL properties of model transformations.

We assume rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup A, R)$, with E , A , and R finite and where A is a set of axioms, so that both the equations E and the rules R are applied *modulo* the axioms A . That is, we rewrite not just terms t but rather A -equivalence classes $[t]_A$. The axioms A are for example very important in the algebraic semantics $\mathbb{A}(\mathcal{M})$ of a metamodel $\mathcal{M}$, because the axioms A of associativity, commutativity, and identity of set union exactly capture the graph-theoretic nature of a model M in $\mathcal{M}$ as a *set* of objects linked by mutual references. Then, rewriting M modulo such axioms exactly corresponds to graph rewriting, a correspondence discussed in [15] and systematically exploited in MOMENT2 as we explain in this paper.

Furthermore, we assume the following executability properties of $\mathcal{R} = (\Sigma, E \cup A, R)$: (i) the equations E are *confluent and terminating* modulo A , and (ii) the rules R are *coherent* with E modulo A . We refer to [14] for a detailed description of these properties and give here only an intuitive description of them. Confluence and termination of E mean that the rules E can be applied from left to right to always obtain a *unique* (modulo A) canonical form for any term. Coherence of R with E intuitively means that the strategy of first simplifying a term with the equations E to canonical form and then rewriting it with R is *complete*: no reachable states are missed by imposing this strategy. For tool support in Maude to verify confluence, termination, and coherence see [14].

3.1 MOMENT2: MOF and Models

MOMENT2 is based on a reflective, algebraic, executable specification of the MOF and OCL standards [10,16] in MEL. Let $\llbracket \text{MOF} \rrbracket$ denote the set of all MOF metamodels $\mathcal{M}$, and let SpecMEL denote the set of all MEL specifications. The algebraic semantics of a MOF metamodel $\mathcal{M}$ is defined as a function $\mathbb{A} : \llbracket \text{MOF} \rrbracket \longrightarrow \text{SpecMEL} : \mathcal{M} \mapsto \mathbb{A}(\mathcal{M})$. $\mathbb{A}(\mathcal{M})$ provides a sort *Model*, whose carrier $T_{\mathbb{A}(\mathcal{M}), \text{Model}}$ is defined by a membership axiom

$$M : \text{Model} \text{ if } \text{wellFormed}(M) = \text{true}$$

that ensures that a model M is semantically well-formed: identifiers are unique, objects are instances of object types of $\mathcal{M}$, there are no dangling edges, and the containment hierarchy, defined by means of composition associations, is preserved. $T_{\mathbb{A}(\mathcal{M}), \text{Model}}$ defines the set of terms that represents models M conforming to the metamodel $\mathcal{M}$, denoted $M : \mathcal{M}$. The MOF metamodels $\mathcal{M}$ that we consider involve the following relational constraints: inheritance relations, and composition and association relations with multiplicities (cardinalities, order and uniqueness). When a model M conforms to a metamodel $\mathcal{M}$, M implicitly satisfies these constraints by means of the aforementioned membership. For the metamodel $\mathcal{M}$ in Fig. 1(a), the model depicted in Fig. 1(b) can be defined as a term of sort *Model* in the $\mathbb{A}(\mathcal{M})$ theory as follows:

```
<< < '1 : Process | id = "p0", next = '2, request = '3, holds = '4 >
  < '2 : Process | id = "p1", next = '1, request = '4, holds = '3 >
  < '3 : Resource | id = "r0", heldBy = Set{ '2 } >
  < '4 : Resource | id = "r1", heldBy = Set{ '1 } > >>
```

where the set of four tuples is formed by an associative and commutative union operator with empty syntax (juxtaposition), and each tuple $\langle \text{Oid} : \text{ClassName} \mid \text{Properties} \rangle$ represents an *object* that is typed with a specific object type of the corresponding metamodel. Objects are defined with *properties* of two kinds: *attributes*, typed with simple data types, and *references*, typed with object identifier types. Each property is defined by a pair (**name** = **value**). All the constructors that are used in the previous term are defined in the signature of the $\mathbb{A}(\mathcal{M})$ theory. The representation of models as algebraic terms is automatically generated by MOMENT2 from models in the Eclipse Modeling Framework (EMF) [17]. A detailed definition of the mapping $\mathbb{A}$ can be found in [9,18].

4 Rewriting Logic Semantics of Model Transformations

MOMENT2's model transformation language supports the standards MOF, OCL and QVT: (i) MOF for specifying object types, (ii) OCL for manipulating attribute values, and (iii) QVT for specifying model patterns. In this paper we focus on endogenous, relational transformations, where the source and target metamodels are the same. A pair $(\mathcal{M}, \mathcal{T})$, of a MOF metamodel $\mathcal{M}$ and a MOMENT2 model transformation $\mathcal{T}$, represents a model transformation, whose *semantics* is formally defined by a rewrite theory $\mathbb{R}(\mathcal{M}, \mathcal{T})$ given by a semantic function $\mathbb{R} : \text{SpecTransf} \rightarrow \text{SpecRL} : (\mathcal{M}, \mathcal{T}) \mapsto \mathbb{R}(\mathcal{M}, \mathcal{T})$ such that $\mathbb{A}(\mathcal{M}) \subseteq \mathbb{R}(\mathcal{M}, \mathcal{T})$. The nondeterministic outcomes of a transformation $(\mathcal{M}, \mathcal{T})$ applied to an input model $M : \mathcal{M}$ are obtained by rewriting M with the rules in $\mathbb{R}(\mathcal{M}, \mathcal{T})$.

In this section, we describe: (i) the concrete syntax of our model transformation language, (ii) the semantics of an *admissible* model transformation $(\mathcal{M}, \mathcal{T})$ as a rewrite theory $\mathbb{R}(\mathcal{M}, \mathcal{T})$, and (iii) a notion of *consistent* model transformations that will be helpful for verification purposes. The MOMENT2 QVT syntax to specify $\mathcal{T}$ and the rewriting logic semantics $\mathbb{R}(\mathcal{M}, \mathcal{T})$ are illustrated by using the graph production rule $rel(p, r)$ in Figure 3 as a running example.

4.1 QVT-Based Syntax for Model Transformations

Transformation. An in-place, relational transformation is specified as a pair $(\mathcal{M}, \mathcal{T})$, which is declared by providing a *label*, a *domain* and a set of *model rewrites*. The QVT notion of domain corresponds to a model that is used in the transformation. The transformation declaration of the example is defined as **transformation** `mutexAlgorithm`(`model` : `mutex`) { .. }, where `mutexAlgorithm` corresponds to the name of the model transformation, `model` corresponds to the domain variable, and `mutex` corresponds to an identifier for the metamodel.

Model Equation/Rewrite. Model transformation rules can be defined either as *equations* or as *rewrites* by the respective keywords `eq` or `rl`, respectively. The corresponding semantics is that they are respectively interpreted as equations in MEL or as rewrites in rewriting logic as explained in Section 3. A model transformation rule always consists of three elements: a label, a left-hand side (LHS) pattern and a right-hand side (RHS) pattern, where the LHS and RHS patterns correspond to collections of object template patterns in the QVT terminology, or to graph patterns in the graph transformation terminology. Optionally, we can add a set of (possibly conditional) negative application conditions (NACs) to each model transformation rule and a global condition with the **when** clause. A complete definition of the model transformation language is provided in [19].

QVT Model Patterns. In a model equation/rewrite, model patterns are represented by using the QVT syntax for *object patterns* (*object templates* in the QVT specification). A model pattern is a collection of object patterns that are applied over a specific domain model. In our running example, we only have one domain

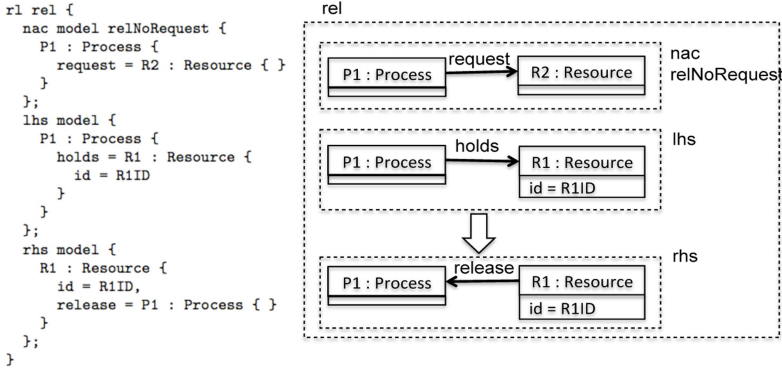


Fig. 3. Model rewrite $rel(p,r)$: textual format in MOMENT2 and graphical representation

called *model*. The LHS model pattern of the $rel(p,r)$ model rewrite in Fig. 3 can be applied over a model that conforms to the metamodel in Fig. 1.(a). This is due to the transformation declaration above. In this model pattern, an object of type **Process** points to an object of type **Resource** through a **holds** reference. We also match the value of attribute **id** of the **Resource** object with the variable **R1ID** to illustrate how attributes can be included in model patterns. This model pattern is shown as a graph pattern in Fig. 3.

Conditions. In MOMENT2 a user can define rules with conditional and negative application conditions, which can be defined as boolean OCL expressions. In addition, OCL expressions can be used to query data in the NAC of a model equation/rewrite and to manipulate data in its RHS. That is, OCL expressions can be used to manipulate the shared variables that are matched either in a LHS pattern or in a NAC of a model transformation rule. See [9] for a detailed algebraic specification of OCL and [16] for some of its applications.

4.2 Rewriting Logic Semantics of Model Transformations

Let $SpecTransf$ denote the set of pairs $(\mathcal{M}, \mathcal{T})$ where $\mathcal{M}$ is a metamodel and $\mathcal{T}$ is a MOMENT2 model transformation definition. The rewriting logic semantics is given by the function $\mathbb{R} : SpecTransf \longrightarrow SpecRL : (\mathcal{M}, \mathcal{T}) \mapsto \mathbb{R}(\mathcal{M}, \mathcal{T})$, where the rewrite theory $\mathbb{R}(\mathcal{M}, \mathcal{T})$ is such that $\mathbb{A}(\mathcal{M}) \subseteq \mathbb{R}(\mathcal{M}, \mathcal{T})$. However, not all model transformation specifications $(\mathcal{M}, \mathcal{T})$ correspond to valid model transformations, because, as explained in Section 3, the associated rewrite theory $\mathbb{R}(\mathcal{M}, \mathcal{T})$ should satisfy reasonable executability requirements such as confluence and termination of equations, and coherence between equations and rewrites. We call a model transformation $(\mathcal{M}, \mathcal{T})$ *admissible* iff $\mathbb{R}(\mathcal{M}, \mathcal{T})$ satisfies such executability requirements. In what follows we illustrate how the model rewrite $rel(p,r)$ in Fig. 3, already specified in Section 4.1, is translated by $\mathbb{R}$ into a conditional rewrite rule in the corresponding theory $\mathbb{R}(\mathcal{M}, \mathcal{T})$.

Model equations/rewrites. In our example rewrite theory $\mathbb{R}(\mathcal{M}, \mathcal{T})$, there is an operator `op mutexAlgorithm` : `Model` $\rightarrow$ `Model`, corresponding to the transformation's name. The $\mathbb{R}$ function maps each model equation/rewrite in an admissible model transformation $(\mathcal{M}, \mathcal{T})$ to equations/rewrites in the theory $\mathbb{R}(\mathcal{M}, \mathcal{T})$. In the running example, model equations are mapped into equations that define the above operator `mutexAlgorithm`. Model rewrites are mapped to rewrites where this operator is the top symbol of the term that is rewritten. Within the mapping of a model equation/rewrite, the following steps are considered: (i) the LHS model pattern becomes a term with variables $t(X_1 \dots X_n)$ that is used to match a model M , such that $M : \mathcal{M}$, (ii) the RHS of a model transformation rule becomes a sequence of operators that uses the variables $X_1 \dots X_n$ to perform atomic changes over a model M , and (iii) the `when` and `such that` clauses become boolean expressions used as conditions in the resulting equation/rewrite.

LHS Model Pattern. A collection of object templates that constitutes the LHS of a model equation/rewrite in a model transformation $\mathcal{T}$ is mapped to a term with variables. For example, the model pattern

```
lhs model { P1 : Process { holds = R1 : Resource { id = R1ID } } };
```

is mapped to the term with variables

```
< P10ID:Oid : Process | holds = R10ID:Oid, P1PS:PropertySet >
< R10ID:Oid : Resource | id = R1ID:String, R1PS:PropertySet > .
```

RHS Model Pattern. The RHS model pattern of a given model equation/rewrite is mapped to a sequence of equationally defined operators that perform atomic changes in a model. These changes correspond to the usual modeling primitives: to create a *new* object, to *destroy* an existing object, to *set/unset* attributes, or to *update/remove* references. These operators manipulate models in a consistent way so that *dangling edges* and *orphan objects* are automatically removed [19].

For the QVT expression of the RHS model pattern in the model rewrite $rel(p, q)$ in Fig. 3 the corresponding RHS term is $[[M] \text{ remove}(P10ID:Oid, "holds", R10ID:Oid)] \text{ update}(R10ID:Oid, "release", P10ID:Oid)$, where M corresponds to the model that is matched in the LHS model pattern, the operator $[M] \text{ op}$ represents the application of an atomic change `op` over the model M , and the variables that correspond to object identifiers are matched in the LHS model pattern, as shown above. In this example, the `holds` reference is removed from the object identified by `P10ID:Oid` and the `release` reference is updated in the object with identifier `R10ID:Oid`. A complete definition of the mapping is presented in [19] considering all possible combinations of atomic changes.

NACs and conditions. NACs in model transformation rules are compiled into equationally defined boolean functions as detailed in [19]. These functions are used in the conditions of the equations and rules that are generated from model transformation rules checking whether a model pattern matching fails or not. *Such that* expressions are compiled into conditions for the equations that define

the corresponding NAC function. *When* expressions are compiled into conditions for the equations or rules that are generated for model transformation rules. *When* and *SuchThat* clauses are constructed using OCL expressions that are compiled into terms as explained in [9].

4.3 Consistent Model Transformations

The dynamic semantics of model-based systems can be given by means of model transformations $(\mathcal{M}, \mathcal{T})$, where the set of system states is a subset of the model type $\llbracket \mathcal{M} \rrbracket$ and transitions are defined by model rewrites in $\mathcal{T}$. The semantic mapping $\mathbb{R}$, when applied to admissible model transformations, plays a crucial role to model check invariants and LTL properties. It ensures that a model transformation β , defined by $\mathbb{R}(\mathcal{M}, \mathcal{T})$, always preserves the constraints of a metamodel $\mathcal{M}$ (types and reference multiplicities — cardinalities, order, uniqueness), when it is applied to a well-formed model $M : \mathcal{M}$, i.e., $\beta(M) : \mathcal{M}$.

Theorem 1. *An admissible model transformation $(\mathcal{M}, \mathcal{T})$ represents a model transformation β in $\mathbb{R}(\mathcal{M}, \mathcal{T})$ that preserves the metamodel conformance relation, i.e., that is consistent w.r.t. the constraints in $\mathcal{M}$.*

Proof (Sketch). Since $(\mathcal{M}, \mathcal{T})$ is assumed admissible, model equations in $\mathcal{T}$ are ground confluent¹ and terminating modulo structural axioms (associativity, commutativity and identity), model equations and rewrites are coherent and there are no free variables in the LHS, NAC and when expressions.

$\mathbb{R}$ maps model equations/rewrites in $(\mathcal{M}, \mathcal{T})$ to ordinary equations/rewrites in $\mathbb{R}(\mathcal{M}, \mathcal{T})$, where the RHS is given as a sequence of atomic changes performed by means of equationally defined operators, namely, *new*, *destroy*, *set*, *unset*, *update* and *remove*. In [19], we prove that the equations that define these operators are confluent and terminating. Such operators are completely defined by equations for admissible model transformations $(\mathcal{M}, \mathcal{T})$, so that these operators will never appear in the normal form that represents a model $M : \mathcal{M}$. Furthermore, the atomic changes that are performed by means of these operators for admissible model transformations $(\mathcal{M}, \mathcal{T})$ only produce well-formed models $M' : \mathcal{M}$. This key consistency property is proved by structural induction in [19]. $\square$

5 Dynamic Analysis in MOMENT2

Model Checking Model-Based Invariants. In a model-based system defined by an endogenous, relational model transformation $(\mathcal{M}, \mathcal{T})$, we are interested in checking *invariants*, that is, predicates that hold of a given initial state $M : \mathcal{M}$ and of all models $M' : \mathcal{M}$ reachable from M by means of state transitions produced by model rewrites. We can use Maude's breadth-first search for

¹ Rewrite steps with model equations make specific choices of names for the objects in the model and the choice might be different depending on the rewrite steps chosen. Graph-theoretically, different models may thereby represent the same abstract model up to renaming. We consider confluence of equations up to name isomorphism [19].

this purpose by searching for a reachable state *violating* the invariant. Even if the number of reachable states is infinite, due to the breadth-first nature of the search, this gives a *semidecision* procedure for invariant violations. But if the reachable states are finite it becomes a *decision procedure*. We can define the *negation* of an invariant predicate by means of a pair (P, C) , where P is a model pattern describing potentially “bad” states, and C is a boolean predicate imposing additional semantic restrictions on the model pattern P . The compilation process then transforms the pair (P, C) into a corresponding pair (t, C') , where t is the pattern term with variables corresponding to P , and C' is a boolean condition involving the variables of t that expresses condition C at the term level. The original invariant $I_{\neg(t, C')}$ that (t, C') negates is then the set-theoretic *complement* of the set of states that are instances of t and satisfy C' .

Given an initial state M of sort *Model* in the theory $\mathbb{R}(\mathcal{M}, \mathcal{T})$, we write $\mathbb{R}(\mathcal{M}, \mathcal{T}), M \models I_{\neg(t, C')}$ to denote that all states reachable from M satisfy the invariant $I_{\neg(t, C')}$. This is the case if and only if no state reachable from M is an instance of t that satisfies C' , which can be semidecided by breadth-first search for an infinite number of reachable states, and can be decided by failure of such search if the set of reachable states is finite.

MOMENT2 uses Maude’s search command for model checking invariant violations of the form (P, C) , with QVT model patterns P and boolean OCL expressions C . In our mutual exclusion algorithm, we can verify that a resource will never be held by two different processes with the search command:

```
search [1, unbounded] =>* domain model {
  P1 : Process { holds = R1 : Resource{} }
  P2 : Process { holds = R1 : Resource{} } }
```

where the pair $[1, \text{unbounded}]$ indicates that one single solution is searched and that all possible states are traversed, and $=>^*$ states that we are applying zero, one, or more model rewrites. In [19], we explain the complete syntax of the command and its compilation to Maude’s search command.

Model Checking LTL Properties. Given an admissible model transformation specified by $(\mathcal{M}, \mathcal{T})$ with semantics $\mathbb{R}(\mathcal{M}, \mathcal{T})$, we can verify Linear Temporal Logic properties over such a model transformation by using Maude’s LTL model checker. In MOMENT2, we enable the use of model predicates in LTL formulas by defining a set $\mathcal{D}$ of model predicates for $(\mathcal{M}, \mathcal{T})$ by means of: (i) possibly parametric model predicate symbols $\mathcal{P}$, and (ii) model-based equations $E_{\mathcal{P}}$ defining the satisfaction relation of a predicate in a model. As an example, the satisfaction of a parametric model predicate $\text{requests}(P, R)$, where P and R are string parameters, can be defined by means of the equation

```
domain model {P1 : Process{id = P, request = R1 : Resource{id = R}}}
|= requests( P, R ) = true
```

stating that the predicate $\text{requests}(P, R)$ is satisfied in a state M , when there is a process with id P that requests a resource with id R in M . Predicate parameters can be either constants or variables that are matched with the model pattern.

Let $SpecPred$ denote the set of specifications of sets of predicates $\mathcal{D}$ that can be defined for an admissible model transformation $(\mathcal{M}, \mathcal{T})$. We define a function $\mathbb{A} : SpecPred \rightarrow SpecMEL$ that maps a set $\mathcal{D}$ of model predicates to a MEL theory defining the state predicates, which are used as propositions in LTL formulae as explained in [14]. For the aforementioned equation, the following equation is generated:

```
eq mutexAlgorithm(<< OC:ObjectCollection
  < POID:Oid : Process | id = P:String,
    request = ROID:Oid, PPS:PropertySet >
  < ROID:Oid : Resource | id = R:String, RPS:PropertySet >
>>) |= requests(P:String, R:String) = true .
```

The following equation defines the satisfaction of the state predicate $heldBy(R, P)$, specifying when a resource with id R is held by a process with id P :

```
domain model {P1 : Process{id = P, holds = R1 : Resource{id = R}}}
|= heldBy( R, P ) = true
```

A model transition system defined by a meaningful model transformation $(\mathcal{M}, \mathcal{T})$ and a set $\mathcal{D}$ of model predicates for $(\mathcal{M}, \mathcal{T})$ is formally defined as a rewrite theory as follows: (i) $(\mathcal{M}, \mathcal{T})$ is mapped to the rewrite theory $\mathbb{R}(\mathcal{M}, \mathcal{T})$, and (ii) $\mathcal{D}$ for $(\mathcal{M}, \mathcal{T})$ is mapped to the equational theory $\mathbb{A}(\mathcal{D})$ that extends $\mathbb{R}(\mathcal{M}, \mathcal{T})$ with model predicate equations. The rewrite theory $\mathbb{R}(\mathcal{M}, \mathcal{T}) \cup \mathbb{A}(\mathcal{D})$ defines a Kripke structure $\mathcal{K}(\mathbb{R}(\mathcal{M}, \mathcal{T}) \cup \mathbb{A}(\mathcal{D}))$ as explained in [14], where the transition relation corresponds to one-step model rewrites in $\mathbb{R}(\mathcal{M}, \mathcal{T})$.

In this way, we enable the use of model predicates as propositions in LTL formulas in Maude's model checker. In the example, we want to verify that each process that requests a resource is always eventually served. This can be specified with the LTL formula $\Box (requests("p0", "r0") \rightarrow \Diamond heldBy("r0", "p0"))$. This property can be model checked in Maude by means of the command

```
red modelCheck( mutexAlgorithm(model),
  [] (requests("p0", "r0") -> <> heldBy("r0", "p0")) ) .
```

By using the model in Fig. 1.(c) as initial state, Maude's model checker found a path in which the property is violated. In particular, the two processes $P0$ and $P1$ request the resource $R0$ and the *pass* rule is henceforth applied over the resource $R1$. This is due to the fact that the rules are not applied in a *fair* way, i.e., all rules are not equally applied in all possible paths. This problem can be solved by forcing the application of the *take* rule. In the metamodel in Fig. 1.(a), we have added the properties *rule* and *currentProcess* to the *Ring* class. The *rule* property indicates which rule has been applied and the *currentProcess* property indicates which process has been activated by the application of a rule. In the model transformation, we can modify the model rules so that each rule updates the *rule* and *currentProcess* in the *Ring* object as in the *take* rule:

```
r1 take {lhs model{RG : Ring{..}};
  rhs model{RG : Ring{rule = "take", currentProcess = P1 : Process{..}};
```

where we only show the elements that are needed to ensure fairness. Two more model predicates are needed to check when the *take* rewrite can be applied (*enabled-take*) and when it has already been applied (*take*):

```
domain model {
  rg : Ring {rule = "take", currentProcess = P1 : Process{id = P}}
} |= enabled-take( P ) = false
domain model { } |= enabled-take( P ) = true [owise]
domain model {rg : Ring{rule = "take",
  currentProcess = P1 : Process{id = P}}
} |= take( P ) = true
```

The *MUTEX-live* property can be model checked for the modified system by encoding the fairness of the *take* rule in the LTL formula as follows:

```
([] <> enabled-take( "p0" ) -> [] <> take( "p0" )) ->
[] (requests("p0", "r0") -> <> heldBy("r0", "p0"))
```

6 Related Work, Conclusions and Future Work

This work has brought closer the algebraic and graph-based approaches to models and model transformations. In this sense, it continues a line of work on algebraic approaches to graphs and graph rewriting including [1,20,21,22,15]. The closest papers to the approach presented here are [22], and particularly [15], which makes explicit the relationship of graph rewriting with rewriting logic. However, our approach has some novel features not treated in previous work, which express the kinds of graph used in model-based software engineering. Specifically, we support different kinds of references, namely, ordinary references, and containment references, which have a different semantic treatment.

Several approaches based on model-checking provide automated procedures for formal verification of model-based systems. On the one hand, model-checking tools with specific support for graph transformations are particularly interesting due to the graph-theoretic nature of models. GROOVE [23] is a graph-based analysis tool that provides model checking for LTS whose states are graphs. Augur [24] is an analysis tool based on the translation of graph transformations to Petri nets and the application of Petri net analysis techniques. On the other hand, generic model checkers have been used, such as SPIN, in approaches such as CheckVML [25]. In [26], Maude is used as a programming language for encoding Atom3 visual graph transformations [27] as Maude system modules so that the state space of a graph transformation system can be examined.

On the other hand, there are fewer approaches that enable the verification of QVT-like model transformations, as opposed to graph transformations. A notion of *graph with containments* is used in [7] to encode model transformations as graph transformations, and to analyse termination and confluence of model transformations in the algebraic graph transformation environment AGG [28]. Another approach [29], based on Alloy, provides the analysis of the consistency of model transformations w.r.t. the metamodel conformance relation, which is

ensured by construction in MOMENT2. Our approach focuses on the formalization of QVT-like model transformations in rewriting logic by reusing the theory of graph transformation systems. In this way, MOMENT2 enhances the application of Maude's techniques for reachability analysis and LTL model checking to model transformations by considering MOF modeling primitives, such as containment relationships, not directly supported in graph-based approaches. At the same time, our approach provides the rewriting logic semantics of graph transformations so that the computational semantics of graph rewriting is given by term rewriting, by considering production rules either as equations or as rewrites. This semantics is directly supported by the MOMENT2 tool [30], an Eclipse plugin in which model transformations between EMF metamodels can be conveniently defined, executed, and efficiently model checked.

Much work remains ahead, including: a closer study of the expressiveness of the MOMENT2 model transformation language and its applications for defining the formal semantics of popular transformation languages such as ATL [4] or QVT [5], and for defining graph-theoretic notions, such as graph constraints and related logics [31] and triple-graph grammars [32]; a further study of the rule/equation distinction for model transformation rules and symmetry reduction techniques for state reduction purposes in model checking; and the formal analysis of MOF domain-specific languages such as real-time DSLs.

Acknowledgments. We cordially thank Francisco Durán for his kind help with the CRC and MTT Maude tools. We also thank the anonymous reviewers for their helpful comments and suggestions. This work has been partially supported by the NSF Grant IIS-07-20482, by the project META TIN2006-15175-C05-01, and by the project SENSORIA, IST-2005-016004.

References

1. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*. Springer, Heidelberg (2006)
2. Ehrig, H., Montanari, U., Kreowski, H.J., Rozenberg, G., Kreowski, H.J.: *Handbook of Graph Grammars and Computing by Graph Transformations*, vol. 3. World Scientific Publishing Company, Singapore (1999)
3. OMG: Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01) (2006)
4. ATLAS Group: ATL web site (2008), <http://www.eclipse.org/m2m/at1/>
5. OMG: MOF 2.0 QVT final adopted specification (ptc/07-07-07) (2007)
6. Ehrig, H., Engels, G., Kreowski, H.J.: *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 2. World Scientific Publishing Company, Singapore (1999)
7. Biermann, E., Ermel, C., Taentzer, G.: *Precise Semantics of EMF Model Transformations by Graph Transformation*. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) *MODELS 2008*. LNCS, vol. 5301. Springer, Heidelberg (2008)
8. Mens, T., Gorp, P.V.: *A taxonomy of model transformation*. *Electr. Notes Theor. Comput. Sci.* 152, 125–142 (2006)
9. Boronat, A.: *MOMENT: a formal framework for MOdel manageMENT*. PhD in Computer Science, Universitat Politècnica de València (UPV), Spain (2007), http://www.cs.le.ac.uk/~aboronat/papers/2007_thesis_ArturBoronat.pdf

10. Boronat, A., Meseguer, J.: An Algebraic Semantics for MOF. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 377–391. Springer, Heidelberg (2008)
11. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(1), 73–155 (1992)
12. Heckel, R.: Compositional verification of reactive systems specified by graph transformation. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 138–153. Springer, Heidelberg (1998)
13. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
14. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
15. Meseguer, J.: Rewriting logic as a semantic framework for concurrency: a progress report. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 331–372. Springer, Heidelberg (1996)
16. Boronat, A., Meseguer, J.: Algebraic Semantics of OCL-constrained Metamodel Specifications. Technical Report UIUCDCS-R-2008-2995, UIUC (2008), <http://www.cs.uiuc.edu/research/techreports.php?report=UIUCDCS-R-2008-2995>
17. Eclipse Organization: The Eclipse Modeling Framework (2007), <http://www.eclipse.org/emf/>
18. Boronat, A., Meseguer, J.: An algebraic semantics for MOF. Technical Report CS-08-005, University of Leicester (2008), <http://www.cs.le.ac.uk/people/aboronat/papers/boMe-mof-apps.pdf>
19. Boronat, A., Heckel, R., Meseguer, J.: Rewriting Logic Semantics and Verification of Model Transformations. Technical Report CS-08-004, University of Leicester (2008), <http://www.cs.le.ac.uk/people/aboronat/papers/boHeMe-rl-mt.pdf>
20. Bauderon, M., Courcelle, B.: Graph expressions and graph rewriting. *Math. Systems Theory* 20, 83–127 (1987)
21. Corradini, A., Montanari, U.: An algebra of graphs and graph rewriting. In: Curien, P.-L., Pitt, D.H., Pitts, A.M., Poigné, A., Rydeheard, D.E., Abramsky, S. (eds.) CTCS 1991. LNCS, vol. 530, pp. 236–260. Springer, Heidelberg (1991)
22. Raoult, J.C., Voisin, F.: Set-theoretic graph rewriting. In: Ehrig, H., Schneider, H.-J. (eds.) Dagstuhl Seminar 1993. LNCS, vol. 776, pp. 312–325. Springer, Heidelberg (1994)
23. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
24. König, B., Kozioura, V.: AUGUR 2—a new version of a tool for the analysis of graph transformation systems. *ENTCS*, vol. 211, pp. 201–210. Elsevier, Amsterdam (2008)
25. Schmidt, Á., Varró, D.: CheckVML: A Tool for Model Checking Visual Modeling Languages. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 92–95. Springer, Heidelberg (2003)
26. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with maude. In: SLE (2008)
27. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-modelling and graph grammars for multi-paradigm modelling in AToM³. *Software and System Modeling* 3(3), 194–209 (2004)

28. AGG Homepage (2008), <http://tfs.cs.tu-berlin.de/agg/>
29. Anastakis, K., Bordbar, B., Küster, J.M.: Analysis of Model Transformations via Alloy. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002. Springer, Heidelberg (2008)
30. MOMENT2 (2008),
<http://www.cs.le.ac.uk/people/aboronat/tools/moment2>
31. Orejas, F., Ehrig, H., Prange, U.: A logic of graph constraints. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 179–198. Springer, Heidelberg (2008)
32. 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)

Confluence in Domain-Independent Product Line Transformations

Jon Oldevik^{1,2}, Øystein Haugen^{2,1}, and Birger Møller-Pedersen¹

¹ University of Oslo, Department of Informatics, Oslo, Norway

² SINTEF Information and Communication Technology, Oslo, Norway
jonold@ifi.uio.no, oystein.haugen@sintef.no, birger@ifi.uio.no

Abstract. Flexible models for product line variability allow representing variability within any kind of domain-specific model. We show how complex variabilities represented by one variability modelling approach are implemented by general, domain-independent model transformations. We analyse the confluence and consistency characteristics of these transformations, show when multiple variabilities may be in conflict, and define the criteria for confluence of multiple variability transformations.

1 Introduction

Software product line engineering (SPLE) has been recognized as a valuable approach for achieving reuse and configurability when building software. A main concept in SPLE are feature diagrams, which are used to describe commonalities and variabilities of a product line. A resolution process is then used to resolve each variability and transform the product line to a specific product. Examples of approaches supporting product line specifications are the Orthogonal Variability Model (OVM) defined by Pohl et al. [1], cardinality-based feature modelling by Czarnecki et al. [2], and the variability model defined by Haugen et al. [3].

The real value of feature diagrams appear when features are related to assets of product line system models, which we denote the *base model*. It can be represented by modelling or programming elements. A resolution of a variable feature can then be directly related to a transformation of corresponding elements of that base model. When the relationship between the feature diagram and the actual product line is rigorously specified, the complete resolution process can be supported by automated model transformations. However, the feature diagram may specify variabilities that are in conflict, which can result in invalid transformations or require specific ordering of resolutions.

The variability model defined by Haugen et al. [3] allows value-based or structural variability to be described on any kind of domain model. Based on this variability model and its links to product line model elements, we implement general model transformations that can transform any product line model in any language defined by a MOF-metamodel [4] to a product configuration. A product line may define variabilities with conflicts e.g. in that they may manipulate overlapping parts of the product line. We analyse confluence and conflict

properties of the product line model and define criteria for determining confluence and detecting model conflicts between multiple variability transformations.

Outline. The remainder of this paper is structured as follows: Section 2 outlines the example we use to illustrate the product line transformations. Section 3 gives the background and basis for variability specification. Section 4 describes how the domain-independent transformations have been realised, and Section 5 addresses confluence characteristics of these transformations. Section 6 presents related work, and finally Section 7 gives some concluding remarks.

2 Motivating Example

We will use a domain-specific language (DSL) representing train lines and stations to illustrate our approach. In fact, we will use a subset of a larger DSL for train stations, which is in use by ABB Norway for defining train line and station topologies. This example was also used by the authors in [3], and its practical usage was detailed by Svendsen et al. in [5]. Fig. 1 shows the metamodel of the train station DSL subset that we will use here.

Two example models in this DSL are the *SingleTrack* and *SwitchedTrack* stations, which are illustrated by the annotated concrete syntax in Fig. 2. The *SingleTrack* (to the left) is defined by four *NormalEndPoint* objects and three *SimpleLine* objects. The lines connect to the endpoints by their *start* and *end* reference properties. The *SwitchedTrack* (to the right) defines two remote switches, which are special kinds of line segments that allow switching between train lines.

We want to reuse these station models to configure new and existing station models. In our illustrating example, we show how to modify an existing single track station to become a switched track station by using a variability model. The result is a new switched track, resulting from replacing objects in the single track base model with objects in the switched track model (Fig. 3).

In the course of this paper, we will show how variabilities specified according to [3] on the example models in the train station DSL are implemented by model transformations to create new model variants in the DSL. We will also analyse conflict and confluence characteristics of these transformations when several variabilities are involved. The variability illustrated here is a kind of

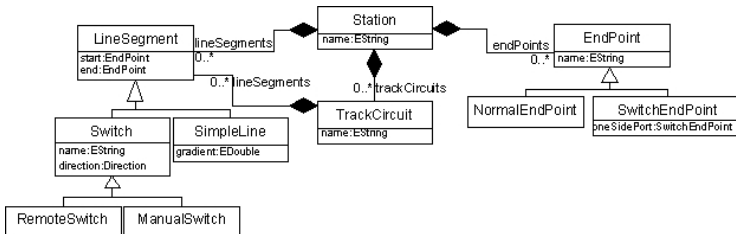


Fig. 1. TrainStation DSL Metamodel

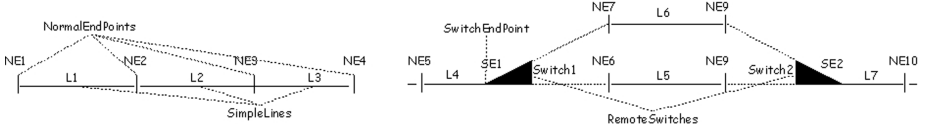


Fig. 2. SingleTrack (left) and SwitchedTrack (right) Stations

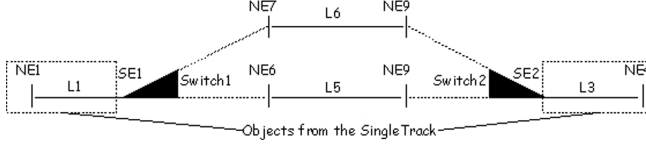


Fig. 3. Modified SingleTrack to SwitchedTrack

alternative variability, where parts of the original base model structure (SingleTrack) represent one alternative and parts of the SwitchedTrack represent the other.

3 Background on Product Line Variability

Product lines are often described in terms of common and variable features. A feature is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family [6]. Common features are shared among all members of a product line, and variable features are those that may vary among product line members. A product line will also often define a common product line architecture (PLA) that is shared by all products in the product line.

Feature Modelling was introduced by Kang et al. [7] in the *Feature Oriented Domain Analysis* (FODA) method, which comprehensively described process and techniques for product line development. Many variants of feature modelling have evolved from this. Notable recent ones are the Orthogonal Variability Model (OVM) by Pohl et al. [1], cardinality-based feature modelling by Czarnecki et al. [2], the conceptual reference model for variability by Bayer et al. [8]. The variability metamodel (hereafter referred to as the *VarModel*) defined by Haugen et al. [3] was based on [8] and adds more general mechanisms for specifying variable fragments of models in *any kind of DSL*. In this paper, we show how the VarModel approach can be implemented by model transformation, and we define important theoretical properties for these transformations with respect to *confluence* and *conflict*. We further define the criteria for checking confluence when several variabilities are involved. To get a basic understanding of the concepts in the VarModel, we describe its central concept for representing variability, namely *substitution*.

Substitution can be viewed as an advanced kind of assignment operator for model elements. Three kinds of substitution are defined:

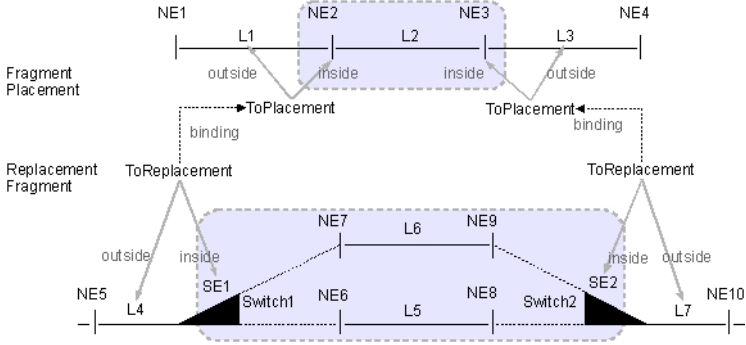


Fig. 5. Fragment Placement and Replacement Fragment

Fig. 5 illustrates a VarModel superimposed on the two station models. The fragment placement logically isolates a set of model elements in the base model, where the edges are defined by boundary elements. In this example, the model elements defining the *inner boundary* of the fragment placement have incoming references from the objects outside the boundary (e.g. the L1 SimpleLine element references the NE2 endpoint element). The placement boundary elements are therefore of type *ToPlacement*. If the reference instead had been from the inner boundary element (NE2) to the outer element (L1), the placement boundary element should have been of type *FromPlacement*. The replacement fragments are defined correspondingly. The full scope of a fragment is found by traversing the links from/to boundary objects with cut-off at any outer boundary object. The shaded parts of 5 shows the model elements that are part of the fragment models.

The selection of an alternative (a replacement fragment) is done by resolutions within the variability model, which is also a part of the metamodel in [3]. It defines resolution elements for different substitutions. For brevity, we cannot go into further detail on the VarModel, but for full detail, the interested reader is referred to [3].

4 The Domain-Independent Product Line Transformation

The modification of the product line base model is controlled by the variability model, and specifically, by resolved variabilities. The resolution process typically involves human decision making, and it adds resolved substitutions to the variability model. The *resolved variability model* is then processed by the product line transformation, which applies the resolutions to the product line domain model. The transformation obtains base model references through the variability model through its *base* reference. Each resolved variability will result in a transformation on the product line base model.

We implemented the transformations using the MOFScript language [9], which has been extended with model to model transformation capabilities and reflection

support at the metamodel (ecore) level. MOFScript is an open source tool originally developed for model to text transformations based on EMF (Eclipse Modeling Framework) models. Its recent extensions allowed us to specify transformations that implement feature substitutions for any kind of base models.

4.1 Generating the Product Model Using the Variability Transformation

The variability transformation processes a resolved VarModel, containing *resolved* substitutions, and generates a product model *from* the domain-specific product line *base model* by applying each resolved substitution. The result of the transformation is based on copies of the original base model(s). We copy the base models referenced by the VarModel and define correspondence links between the original base models and the copies. Correspondence links are defined by iterating all features of all objects in the base models and adding links between the base model and the copied model elements. Then, all variation elements of the VarModel are processed. The transformation must handle the three different kinds of resolved substitutions of the VarModel (*Value-*, *Reference-*, and *FragmentSubstitution*). We will go through each of these in detail.

The Resolved Value and Reference Substitutions. The resolved Value Substitution modifies a value for an element in the base model. A resolved Reference Substitution is similar, but modifies a reference in the base model. In our example, a value substitution could be to modify the value of the *gradient* property of a SimpleLine. Fig. 6 (left hand side) illustrates this value substitution in the *SingleTrain* model. The property *val* of the *Resolved Value* object defines the value to be set on the property identified by the *attr* property of the value substitution. The right hand side illustrates a reference substitution on the same model; it modifies the *start* property of the *L1* line to point to the *N4* element instead of the *N1* element, hence modifying the model structure.

The setting of model element references is done much in the same way as setting values, by a reflexive call on the model element using the feature

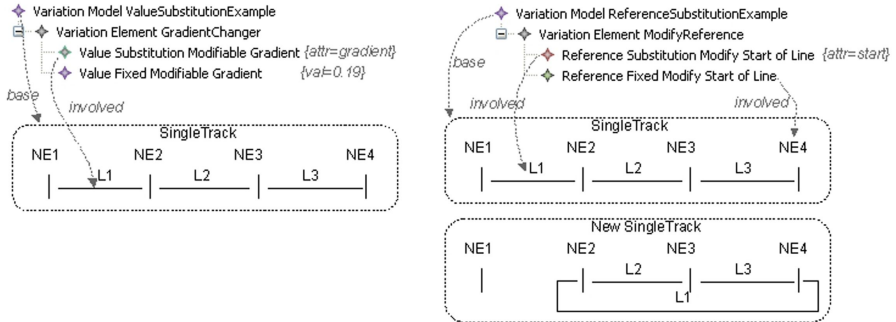


Fig. 6. Value (left) and Reference (right) Substitutions

information given by the substitution. When references are set, the transformation must check if the receiving feature is a collection or a single element and handle the assignment correspondingly. In addition, if references are moved across base models, model elements will be moved from one base model to another.

The Resolved Fragment Substitution. As previously described, the resolved Fragment Substitution may be used to handle optionality, alternatives, and repetition variability of fragments covering parts of a model. Here, we will focus in detail on how alternatives, as illustrated by our original example (Fig. 5), are handled. In this example, we have two base models; one that defines a generic single track line and one that defines a double track with switches.

The transformation of the example model iterates all boundary elements contained in the replacement fragment. For each replacement boundary element (*rbe*):

- If *rbe* is an ingoing boundary element (ToReplacement), the *outside* model element defined by the fragment placement in *rbe.binding* is modified by changing its reference (defined by *rbe.binding.attr*) to point to the *inside* element defined by the replacement fragment (*rbe.inside*) (SE1 from Fig. 5).
- If, on the other hand, *rbe* is an outgoing boundary element (FromReplacement), the *inside* model element defined by the *rbe.inside* is modified by setting its value to reference the model element referenced by the outside of the bound fragment placement (*rbe.binding.outside*).

The corresponding references between the fragment placement inside and outside elements will be removed (*rbe.binding.inside/outside*). Fig. 7 Step a) illustrates this first step. The model elements identified by the fragment placement are deleted according to the following algorithm: for each element *E* in

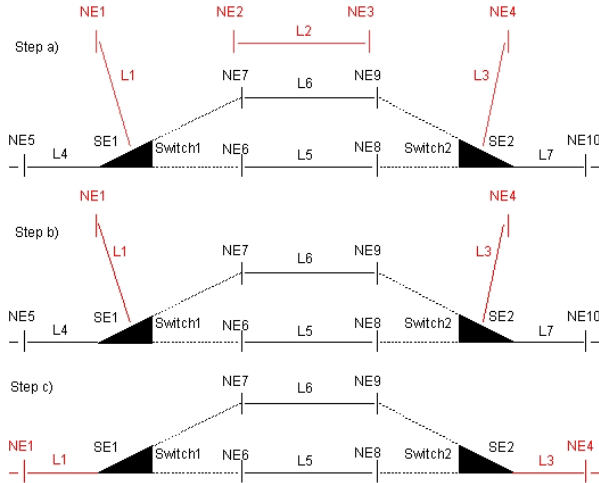


Fig. 7. Steps in the Fragment Substitution

ToPlacement.inside $\cup$ *FromPlacement.inside*, delete E and any element in the transitive closure of all references to and from E , but cut off at any element in the set of *ToPlacement.outside* $\cup$ *FromPlacement.outside* (Fig. 7 Step b).

Since the model elements identified by the fragment placement and replacements may live in different model containers (which is the case in our example), the elements of the replacement fragment are copied into the base model containing the fragment placement. The algorithm is similar to that for deletion of the fragment placement: all model elements in the transitive closure of all references to and from *ToReplacement.inside* $\cup$ *FromReplacement.inside*, but cut off by any element in the set of *ToReplacement.outside* $\cup$ *FromReplacement.outside*.

The two algorithms for deleting and relocating objects are vulnerable to outgoing and incoming references not captured by the boundary elements. An important invariant for the fragment substitutions is that the fragments are defined completely, i.e. that all references going into or out from a fragment are described by boundary elements; a precise representation of boundary elements for references entering/exiting the fragments is required to avoid deletion or copying of unintended model elements.

5 Confluence of Variability Transformations

A product line may have many dependencies between its features. Some of these are explicitly designed in the product line by relationships or feature constraints. There may also be known ordering constraints that govern the order by which features may be resolved. Conversely, there may be ordering constraints that are *not* known in advance, which are there due to constructs in the underlying product line model. We can use confluence analysis to determine if multiple supposedly independent feature substitutions *are* independent or not, when there are no constraints in place.

In term rewriting systems, confluence describes that terms in a system can be rewritten in more than one way and still yield the same result [10]. Within graph transformation theory, confluence can be used to show that a graph transformation with different paths have a unique normal form [11,12]. We will use confluence to reason about the effects of changing the order of transformations represented by a variability model, i.e. if we can expect the same result if the ordering of variability substitutions is altered. We look at the three kinds of substitutions and analyse their confluence characteristics.

5.1 Value and Reference Substitution

A value substitution only modifies the value of properties within an object. It does not change the structure of a model. As such, a value substitution can never conflict with a substitution modifying the model structure (reference or structure substitution).

However, two value substitutions will conflict with each other if they modify the same property of an object, with the result that only the last transformation

overwrite the result of the first one. The governing constraint for value substitution is: *two value substitutions should not be overlapping both with respect to their involved model element and the referenced attribute.*

Reference substitutions may, similarly to value substitutions, interfere with each other if two or more address the same reference of the same model element. The governing constraint for reference substitutions is: *two reference substitutions should not be overlapping both with respect to their involved model element and the referenced property.*

Reference substitutions may also be in conflict with fragment substitutions, as we will address in the next Section.

5.2 Fragment Substitutions

Fragment substitutions are more challenging regarding conflicts as they manipulate model element structures that are larger part of the models.

Overlapping Fragments. We define a model fragment (MF) within a base model (M) in terms of three disjunct sets of model elements from the base model, where these elements are specified by boundary element within the fragment MF : $MF = \{E_{int}, BE_{int}, BE_{ext}\}$, where E_{int} represents elements internal to the model fragment without being part of a boundary element, BE_{int} represents elements specified as the inner boundary of the MF , and BE_{ext} represents elements specified as the outer boundary of MF . Specifically, if MF is a fragment placement, BE_{int} is defined by the set of model elements in *placementBE.inside*, and BE_{ext} by *placementBE.outside*, where *placementBE* is the *PlacementBoundaryElements* contained in the fragment placement. For a replacement fragment, the fragment model is defined in the same way by the *ReplacementBoundaryElements*. The set of model elements in E_{int} is defined by the traversal algorithm as the elements internal to the fragment model. Fig. 8 illustrates the model element sets in a fragment model, where the objects (f1, f2, etc.) have uni-directional references to each other.

The set of model elements outside the fragment (MF) is defined by $(M \setminus MF) \cup BE_{ext}$. We call model fragments that share a set of model elements *overlapping*.

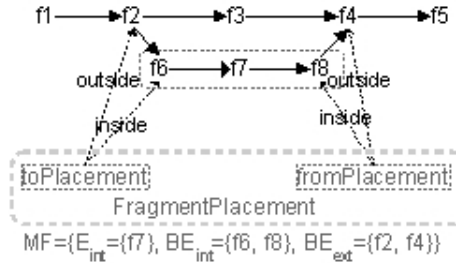


Fig. 8. The Element Sets in a Fragment Model

Definition 1. Two model fragments MF_1 and MF_2 are overlapping if: $(MF_1.E_{int} \cup MF_1.BE_{int}) \cap (MF_2.E_{int} \cup MF_2.BE_{int}) \neq \emptyset$. We say that the two model fragments MF_1 and MF_2 have an external boundary overlap if $MF_1.BE_{ext} \cap (MF_2.E_{int} \cup MF_2.BE_{int}) \neq \emptyset$.

Definition 2. A model fragment MF_1 is a model fragment of another model fragment MF_2 if all model elements in MF_1 are also in MF_2 : $Fragment(MF_1, MF_2) \equiv o \in MF_1 \Rightarrow o \in MF_2$.

Definition 3. We call MF_1 a proper model fragment of MF_2 if it is a model fragment of MF_2 and MF_1 only contains elements internal to MF_2 : $MF_1 \subseteq MF_2.E_{int}$.

A model fragment may be defined as a proper model fragment of another to impose variability on the model fragment itself. For a fragment placement, it makes little sense to have another model fragment inside, as the outer fragment will be replaced anyway. However, within a replacement fragment, it may make sense to define proper fragment placements. Replacement fragments may be defined with any level of overlapping. Since their elements are copied and replace elements defined by a fragment placement, no such overlap will result in a modification or conflict between the replacement fragments.

A replacement fragment cannot be overlapped by a fragment placement unless the latter is a proper model fragment of the other. In that case, the fragment placement provides an internal variability space for the replacement fragment.

Such a proper model fragment will in principle result in confluence with respect to the ordering of the two associated substitutions. In the current implementation, however, these fragments must reside in the same physical model. If all fragments are contained within the same physical model space, proper model fragments are confluent with respect to the ordering of the substitutions. Fig. 9 illustrates a transformation involving proper model fragments.

In this example, the fragment placement $F2$ is a proper model fragment of the replacement fragment $A1$. In addition, we have one more fragment placement ($F1$) and replacement fragment ($A2$). In the first transformation path, the proper model fragment $F2$ is replaced with $A2$. The result still have the replacement fragment $A1$, which is used as replacement for $F1$. In the second path, the replacement fragment $A1$ is first used as replacement for $F1$. The intermediate result still contains the proper model fragment $F2$, which is then replaced by $A1$. The results of the two paths are identical.

A proper model fragment (pmf) defined as part of another replacement fragment (rf) implies (by definition 3) that $pmf \subseteq rf.E_{int}$. A replacement of pmf inside rf with another replacement fragment, $rf2$, will change the structure of rf by replacing the involved objects in the $rf.E_{int}$ set, obtaining $rf.E'_{int}$. It will however never touch the $rf.EB_{int}$ or $rf.BE_{ext}$ elements. When the rf fragment is used as replacement for another fragment placement, the set of $rf.E'_{int}$ elements will be copied as part of rf . If instead the original rf that contains pmf is used in the replacement, the replacee will contain all the elements of the fragment placement pmf , which subsequently can be replaced with replacement fragment $rf2$.

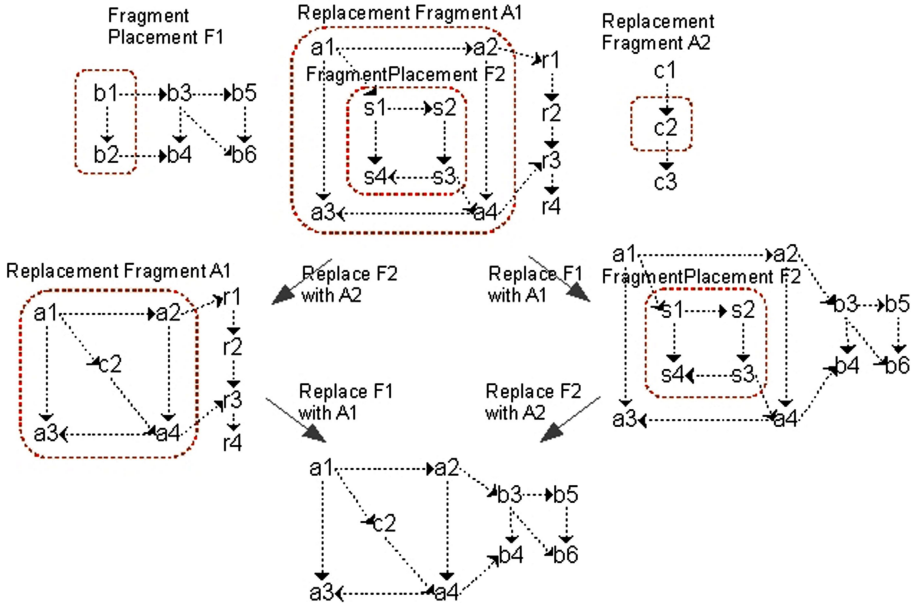


Fig. 9. Confluence of Proper Model Fragments

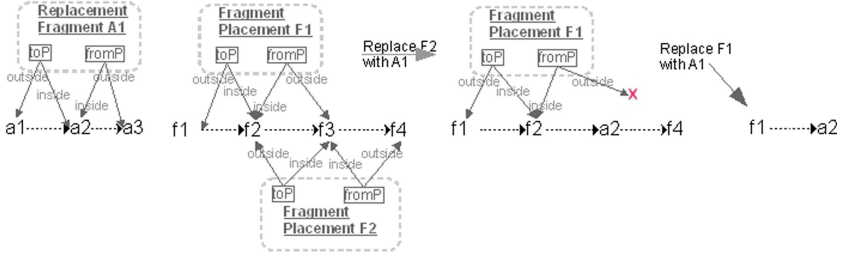


Fig. 10. Conflicting Model Fragments

Otherwise, overlaps between model fragments will not only result in non-confluence among substitutions, but also invalidate any order of transformations. This may happen even if the fragments only have an external boundary overlap, i.e. that model elements associated with the $(E_{int} \cup BE_{int})$ set of one fragment are the same as those associated with the BE_{ext} elements of another.

Fig. 10 illustrates what may happen in the case of an external boundary overlap. The fragment placement $F1$ is defined by the following base model elements: $E_{int} = \{\emptyset\}$, $BE_{int} = \{f2\}$, $BE_{ext} = \{f1, f3\}$, while the fragment placement $F2$ is defined by these base model elements: $E_{int} = \{\emptyset\}$, $BE_{int} = \{f3\}$, $BE_{ext} = \{f2, f4\}$. In this example, the fragment placement $F2$ is replaced by the replacement fragment $A1$. This transformation replaces the *replacement* fragment element $a2$ for the *placement* element $f3$ within our model copy. In the model copy, the

$f3$ is deleted, and although our original base model keeps the element and the fragment placement points to this, the mapping to the model copy of that element is lost. The cut off at $F1.fromPlacement.outside$ for the traversal of the $F1$ fragment placement has now been invalidated; when $F1$ now is replaced by the replacement fragment $A1$, the deletion of the fragment placement elements will never reach the original $f3$ cut off boundary, hence deleting model elements $a2$ and $f4$ by following the references. The resulting model contains only the $f1$ and $a2$ elements.

Conflicts Between Reference Substitutions and Fragment Substitutions. If a reference substitution operates within the model element set identified by a fragment, it may result in unwanted effects: if a reference internal to the fragment is modified to point to the outside of the fragment or an external reference is modified to point inside the fragment, this will break the traversal cut off of the fragment and include unintended model elements for deletion or addition through an extended traversal path that is not captured by a boundary element. This is similar for elements outside of the fragment domain boundary, which should not be allowed to modify a reference to point inside the fragment.

In the case of a fragment placement, this will result in deletion of unwanted objects. In the case of the replacement fragment, it will result in copying unintended objects.

For a model M and a model fragment MF , where $MF \subseteq M$ and $MF = \{E_{int}, BE_{int}, BE_{ext}\}$, the set of model elements outside MF , MF_{out} is defined by $M \setminus MF$. There should be no reference substitution that modifies an element in $(E_{int} \cup BE_{int})$ to reference an element in MF_{out} . There should equally be no reference substitution that modifies an element in MF_{out} to reference an element in $(E_{int} \cup BE_{int})$. Finally, there should be no reference substitution that modifies a reference going from BE_{ext} to BE_{int} to an element in MF_{out} . All these will lead to inconsistency and non-confluence with respect to the two substitutions.

5.3 Confluence Checking of Fragment Substitutions

To see how we can analyse confluence of fragment substitutions, we consider a model M and a VarModel VM with two fragment substitutions $fs1$ and $fs2$, where $fs1$ and $fs2$ each contains one fragment placement and one replacement fragment, $fp1$ and $rf1$, and $fp2$ and $rf2$, respectively.

Confluence of fragment placements can be determined by analysing two properties: model element overlap and completeness of a fragment substitution. Model element overlap of two fragment placements, $fp1$ and $fp2$, can be checked by the contents of their respective E_{int} , BE_{int} , and BE_{ext} sets, basically by checking if there are overlaps in the object sets.

An overlap may only exist in two fragment placements in the BE_{ext} sets. The rest of the object sets must be disjoint:

$$- (fp1.E_{int} \cup fp1.BE_{int}) \cap (fp2.E_{int} \cup fp2.BE_{int}) = \emptyset \wedge (fp1.BE_{ext}) \cap (fp2.E_{int} \cup fp2.BE_{int}) = \emptyset \wedge (fp2.BE_{ext}) \cap (fp1.E_{int} \cup fp1.BE_{int}) = \emptyset.$$

Any other overlap between fragment placements will result in deletion of boundary objects for a fragment placement when the other one is replaced, resulting in an inconsistent model. If fragment placements adhere to this constraint, the respective transformations will be confluent. The replacement fragments do not influence the confluence property.

In addition to set overlap, each fragment substitution must be complete in the sense that it captures *all* incoming and outgoing reference on the boundary of the fragment. For a fragment placement $fp1$, this means that all references going in and out of the fragment are captured by boundary elements: f

- $\forall e \in fp1.E_{int}, \forall e2 \in M \mid \text{hasReference}(e, e2) \Rightarrow e2 \in (fp1.E_{int} \cup fp1.BE_{int})$
- $\forall e \in fp1.BE_{int}, \forall e2 \in M \mid \text{hasReference}(e, e2) \Rightarrow e2 \in (fp1.E_{int} \cup fp1.BE_{int} \cup fp1.BE_{ext})$

The $\text{hasReference}(e, e2)$ operation returns true if there is a reference in either direction between elements e and $e2$. The constraints defined here gives a simple way of checking confluence between fragment substitutions. We have implemented a model checking algorithm to check the fragment set overlapping property.

6 Related Work

In [13], Czarnecki and Antkiewicz describe an approach for mapping features to models and generating model instances (configurations). They use model templates with feature annotations that are matched and evaluated against a feature configuration. The approach is generally applicable to any model domain based on MOF, which is the same for our approach. The implementation of their approach, however, seems dependent on the domain, which is avoided in our approach. The model templates are expressed in the base language and defines the product line in terms of base language elements and presence conditions as annotations to this model. Hence, the variability specifications are embedded into the base language, which is not the case in our approach.

Significant work on confluence has been done in term rewriting and graph transformation theory. Heckel et al. [12] define confluence properties for typed attributed graph transformation systems, which are based on well established theory of parallel independence of graph transformations. Parallel independence requires that two transformations only share elements that are preserved by both steps, i.e. one transformation cannot delete elements used by another. This property has been used to show commutativity and confluence of transformations. The transformations defined in this paper can be mapped to graph transformations of general MOF 2.0 object graphs, allowing confluence properties to be analysed using graph transformation techniques such as critical pair analysis. The transformations described by substitutions are, however, very limited in that each substitution is never recursive, which makes the described analysis quite feasible.

The work by Batory et al. [14] on stepwise refinement provides an approach for composition of features based on hierarchical equations and shows that it can be applied for code and non-code artifacts, given that composition operators are

defined for the artifact type. The execution of each variability substitution can be viewed as one step in such a refinement, where the composition operator is defined by the transformation implementation semantics.

7 Conclusions and Future Work

We have described our solution to domain-independent product line transformations, which was implemented as a model transformation in MOFScript [9]. We addressed how different kinds of variability defined by the variability model are handled by model transformations. Based on the variability transformation, we analysed confluence and conflict characteristics when several points of variation are involved. We defined the criteria for confluence between different kinds of substitutions and specified how confluence between substitutions can be checked. We showed that fragment substitutions that are proper fragment models of replacement fragments resulted in confluent transformations. In general, we saw that fragments should have disjunct domains, i.e., they should not be overlapping in order to be confluent.

The variability transformations require further work in some areas. Specifically, we have not addressed recursively configurable model fragments, to handle structure substitutions that produce duplicates (repetitions), where each duplicate can be individually and recursively configured. There are, however, still limitations in the variability model that prevent this. The transformations have been tested on relatively small examples in simple DSLs, such as example train station model instances; we need to test the scalability of the approach on more elaborate models and metamodels.

Acknowledgments. This work has been carried out in the context of the SWAT project (Semantics-preserving Weaving - Advancing the Technology), funded by the Norwegian Research Council (project number 167172/V30). It has also partly been conducted within the MoSiS project (ITEA 2 - i06035). MoSiS is a project within the ITEA2 - Eureka framework. Information included in this document reflects only the authors views.

References

1. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
2. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models, pp. 266–283. Springer, Heidelberg (2004)
3. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In: Software Product Line Conference (SPLC) (2008)
4. Object Management Group (OMG): Meta Object Facility (MOF) Core Specification. OMG Available Specification formal/06-01-01, Object Management Group, OMG (2006)

5. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.J., Haugen, O.: The Future of Train Signaling. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) *MODELS 2008*. LNCS, vol. 5301, pp. 128–142. Springer, Heidelberg (2008)
6. Czarnecki, K., Eisenecker, U.: *Generative Programming - Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York (2000)
7. Kang, K., Cohen, S., Hess, J., Novak, W., Petersen, S.: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, CMU/SEI-90-TR-21. Technical report, Software Engineering Institute (SEI) (1990)
8. Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J., Widen, T.: Consolidated Product Line Variability Modeling. In: *Software Product Lines, Research Issues in Engineering and Management*. Springer, Heidelberg (2006)
9. Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., Berre, A.: Toward Standardised Model to Text Transformations. In: Hartman, A., Kreische, D. (eds.) *ECMDA-FA 2005*. LNCS, vol. 3748, pp. 239–253. Springer, Heidelberg (2005)
10. Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) *Computational Problems in Abstract Algebra* (1970)
11. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation. Part I: Basic Concepts and Double Pushout Approach. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Foundations, vol. 1, pp. 163–245 (1997)
12. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: *Proceedings of the First International Conference on Graph Transformation*, pp. 161–176 (2002)
13. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) *GPCE 2005*. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
14. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-wise Refinement. In: *ICSE 2003: Proceedings of the 25th International Conference on Software Engineering*, pp. 187–197. IEEE Computer Society, Washington (2003)

Object Flow Definition for Refined Activity Diagrams

Stefan Jurack¹, Leen Lambers², Katharina Mehner³, Gabriele Taentzer¹,
and Gerd Wierse¹

¹ Philipps-Universität Marburg, Germany

{sjurack, taentzer, gwierse}@mathematik.uni-marburg.de

² Technische Universität Berlin, Germany

leen@cs.tu-berlin.de

³ Siemens, Corporate Technology, Germany

katharina.mehner@siemens.com

Abstract. Activity diagrams are a well-known means to model the control flow of system behavior. Their expressiveness can be enhanced by using their object flow notation. In addition, we refine activities by pairs of pre- and post-conditions formulated by interrelated object diagrams. To define a clear semantics for refined activity diagrams with object flow, we use a graph transformation approach. Control flow is formalized by sets of transformation rule sequences, while object flow is described by partial dependencies between transformation rules. This approach is illustrated by a simple service-based on-line university calendar.

1 Introduction

UML2 activity diagrams are a well-known means to model the control flow of system behavior. Their expressiveness can be enhanced by using their object flow notation. Currently, it is an open problem how to formalize coherent object flow for activity diagrams. In this paper we aim at providing a precise semantics for refined activity diagrams with coherent object flow. We use graph transformation as semantic domain, since it supports the integration of structural and behavioral aspects and provides different analysis facilities.

In [1], sufficient criteria for the consistency of *refined* activity diagrams were provided, where interrelated object diagrams are used to specify pre- and post conditions of single activities. All conditions refer to a domain class model. This refinement serves as a basis for consistency analysis. The refinement of activities by pre- and post-conditions was first introduced in [2] to analyze inconsistencies between individual activities refining use cases. Pre- and post conditions are formalized as graph transformation rules. Mehner et.al. extend the consistency analysis in [3] where also the control flow is taken into account. In [4], a similar approach for consistent integration of life sequence charts (LSCs) with graph transformation, applied to service composition modeling, was developed. The formalization based on graph transformation is used to analyze rule sequences. In addition, data flow is modeled textually by name equality for input and output variables.

In this paper, we extend refined activity diagrams by object flow. We introduce partial rule dependencies to formalize the semantics of object flow. Based on the consistency

notion of refined activity diagrams in [1], we define consistency-related properties of refined activity diagrams with object flow.

We illustrate our approach with an example from model-driven development of a service-based web university calendar. In particular the behavior modeling of individual services still lacks advanced support for precise modeling and subsequent consistency analysis. Activity diagrams are an adequate means for modeling individual services, and the use of object flow and pre-/post-conditions can define service behavior more precisely.

This paper is organized as follows. Section 2 introduces the syntax and semantics of refined activity diagrams with object flow informally. Section 3 introduces algebraic graph transformations and the new notion of partial rule dependency. Section 4 presents the semantics and consistency notion of refined activity diagrams and extends it for object flow. Sections 5 and 6 contain related work and concluding remarks.

2 Introduction to Refined Activity Diagrams with Object Flow

This section introduces refined activity diagrams with object flow and illustrates this modeling approach by a small example for a service-based web university calendar. In this example, we model services by activity diagrams with object flow where each activity is refined by pre- and post-conditions, and guards are refined by patterns.

2.1 Domain Model

Our example application manages course parts that are lectures, laboratories, and exercises where a lecture may offer a laboratory and an exercise. Each course part is held by a lecturer and can be located in a room. An appropriate class diagram is presented in Fig. 1. From an abstract class *Object*¹, three classes are derived: *Room*, *Lecturer* and *CoursePart*. The latter is abstract and is specialized by three further classes: *Laboratory*, *Exercise* and *Lecture*. Day and time information for course parts are realized by enumerations *Day* and *Time*.

2.2 Activity Diagrams with Object Flow

We use UML2 activity diagrams with object flow [5] to model services of the university calendar. Three services, *AddLecture*, *AddExercise*, and *AddLaboratory*, are shown exemplarily in Fig. 2.

Web applications usually contain a number of services. A service provides a clearly defined logical unit of functionality based on data entities. While a basic service might be realized by one activity only, more complex services might contain a number of different activities. Defining services by the means of hierarchical activity diagrams opens up the possibility to call services from other ones. The usage of other services is depicted by placing a complex activity as representation of the used service into the control flow. The invocation of a complex activity is indicated by placing a rake-style symbol within the activity node. Our example service *AddLecture* uses two other

¹ Italic class names in diagrams indicate abstract classes.

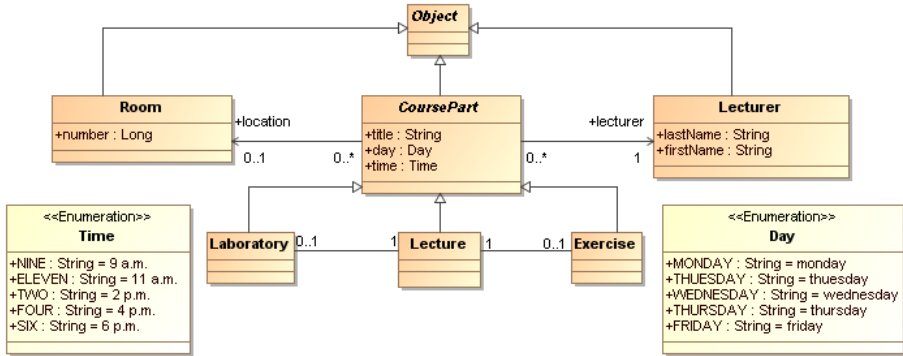


Fig. 1. Domain Class Diagram

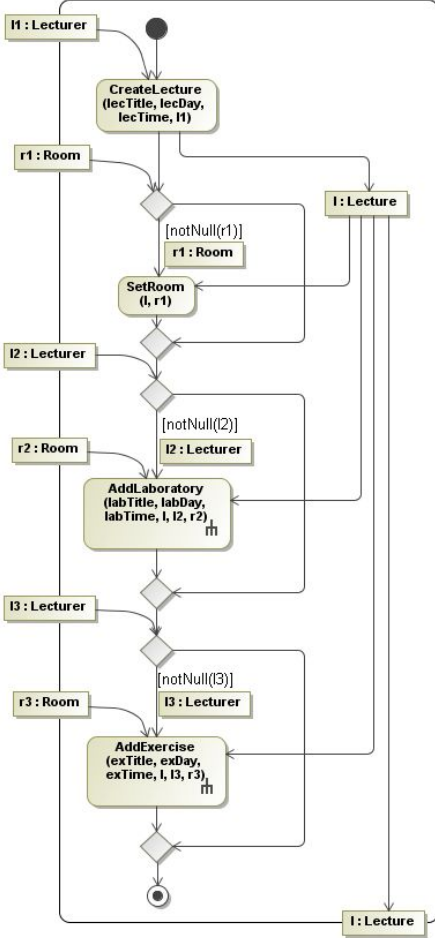
services. Accordingly, the complex activities modeling used services *AddLaboratory* and *AddExercise* are refined by corresponding activity diagrams (cf. Section 4).

UML2 provides several object flow notations. The preference for a notation depends on different aspects, e.g. the amount of information, potential ambiguities, and the equality of control and object flow. For example, if object and control flow overlap, related objects may be depicted next to transitions as shown above activity *SetRoom* in Fig. 2. Otherwise an object node with separate object flow edges has to be used as shown for lecture *l*. However, it is desirable to keep the object flow description as simple as possible without leaving out important information. Each object may be named and its identity is expressed by equal names within an activity diagram. E.g. in activity diagram *AddLecture* both lecturer nodes named *l2* depict the same object. Please note that in our approach, an object may flow along multiple outgoing edges i.e. object flows, whereas in UML2 one object serves one object flow exclusively.

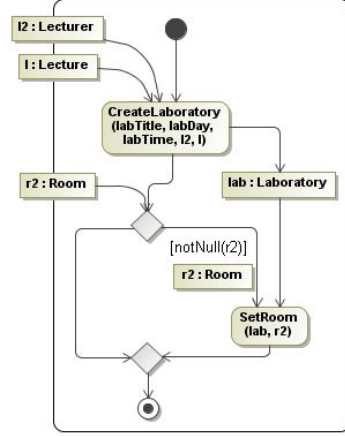
Objects passed from outside to an activity diagram can be drawn on the diagram boundary in order to show parameters flowing into certain activities. Objects passed out of the diagram itself, may be depicted as boundary objects as well. Consider Fig. 2: Objects of types *Lecturer* and *Room* are passed to the activity diagram *AddLecture*, while a newly created object of type *Lecture* is passed out of this diagram.

In Fig. 2, service *AddLecture* uses two other services *AddLaboratory* and *AddExercise*. Once a lecture has been created and its attributes have been set, a related laboratory or exercise might be created additionally. At first, a new lecture is created in activity *CreateLecture*, its attributes are set and it is linked to lecturer *l1*. If, moreover, room *r1* is not null, activity *SetRoom* is used to link this room *r1* to the lecture newly created. If a lecturer is given for a laboratory, the complex activity *AddLaboratory* is used to add a laboratory to the lecture. Therefore, *AddLecture* has to pass the newly created lecture *l*, lecturer *l2*, and room *r2* to the activity. In diagram *AddLaboratory* a new laboratory is created by the first activity *CreateLaboratory*. In the same step, this laboratory is linked to lecture *l* and to lecturer *l2*. Furthermore, the laboratory's attributes are set. In the next activity, the laboratory's location is set to room *r2*, provided that *r2* is given. If a lecturer for a related exercise is given, *AddExercise* is used by *AddLecture* analogously.

AddLecture(in String lecTitle, in Day lecDay, in Time lecTime, in String labTitle, in Day labDay, in Time labTime, in String exTitle, in Day exDay, in Time exTime, in Lecturer I1, in Room r1, in Lecturer I2, in Room r2, in Lecturer I3, in Room r3, out Lecture I)



AddLaboratory(in String labTitle, in Day labDay, in Time labTime, in Lecture I, in Lecturer I2, in Room r2)



AddExercise(in String exTitle, in Day exDay, in Time exTime, in Lecture I, in Lecturer I3, in Room r3)

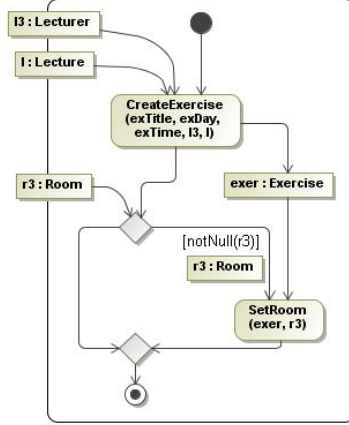


Fig. 2. Activity Diagrams of Services *AddLecture* and *AddLaboratory*

Since our activity diagrams model services, we equip each of them with a name and a comma-separated list of *parameters*. The semantics follow the programming concept of parameter passing between operations, i.e. an activity diagram models an operation consisting of a signature and a body. The signature of an activity diagram consists of its name and a list of attribute and object parameters. While object parameters have a type occurring in the domain model, attribute parameters have primitive types in most cases. This signature is an extension of UML2 made by our approach. Please note that all attributes and boundary objects used within the activity diagram are arguments which correspond to the signature. In addition, each parameter declaration has to be enriched with keyword *in*, *out*, or *inout*. This qualification defines the object flow direction. E.g.

lecturer *l* has to be passed to diagram *AddLecture* and is therefore marked *in*. Vice versa, the newly created lecture *l* is passed out of the diagram and is therefore marked by *out*. Parameter objects marked by *inout* are both input and output objects.

2.3 Refined Activities

Activities are used to model specific changes of the current system snapshot i.e. object structure. We propose to refine activities by pre- and post-conditions specifying snapshots before and after the activity respectively. We refine activities separately by pairs of object diagrams which are typed over the domain model. Figure 3 shows object diagrams refining activities of our example (cf. Fig. 2) where pre-conditions are depicted on the left and post-conditions on the right. Objects and links with equal names on both sides express identity and preservation. Objects and links occurring on the left-hand side only will be deleted, while objects and links occurring in the right-hand side only will be created. Conditions on non-existence of patterns are depicted in red dashed outline.

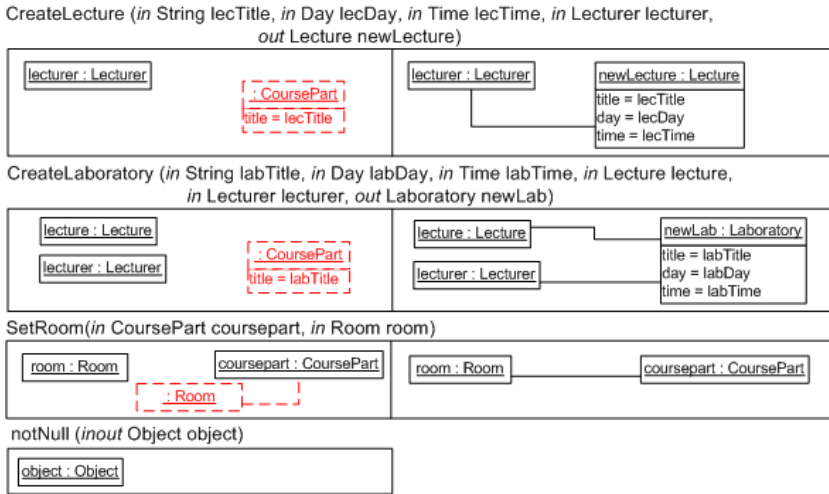


Fig. 3. Refined Activities by Pre- and Post-Conditions

Each pair of conditions exhibits a signature according to the inscription of its refined activity, i.e. it consists of a name (the activity name) and a list of typed parameters qualified with keyword *in*, *out* or *inout*. Parameters can be distinguished into object and attribute parameters, analogously to their usage in activity diagrams. While the former ones are matched to objects, the latter ones are used as attribute values. Keyword *in* requires the occurrence of the related object (if object parameter) on the left-hand side. The object may be used in a read, edit, or delete operation. Keyword *out* declares a returned object and requires its presence on the right-hand side. It may be used for a create or select operation. *Inout* declares an object to be given and returned as well, thus requires the given object on both sides which explicitly guarantees its non-deletion. Attribute parameters must be input parameters. If occurring in pre-conditions, attribute

parameter values restrict the matching of objects, occurring in post-conditions they are used to assign attribute values. Object parameter types must be respected by condition checking, i.e. by pattern matchings. Parameters may be matched, if they are matched with equally typed or sub-typed values only. Analogously, this must hold for attribute types. Note that arrays and collection-like types are not supported by our approach yet.

The first pair of conditions in Fig. 3 refines activity *CreateLecture*. The pre-condition requires the existence of a lecturer in the current system snapshot, otherwise the activity cannot be applied. Also, it requires the non-existence of a *CoursePart* instance (which could be of concrete type *Lecture*, *Exercise*, or *Laboratory*) with a *title* equal to given attribute parameter *lecTitle*. If both conditions hold, the activity is applicable and creates a *Lecture* instance associated with the given *Lecturer* instance and the lecture is returned. The refinement of activity *CreateLaboratory* shown as second pair in Fig. 3 is quite similarly, but it requires two given objects to exist and the creation of an object of type *Laboratory*. Since the conditions of *CreateExercise* are analogous to those of *CreateLaboratory*, they are left out. The refinement of activity *SetRoom* is shown as third pair. It requires two object parameters, one instance of type *Room* and one of type *CoursePart*, and it forbids the *CoursePart* instance to have a room already. No object but a link between the given course part and the new room is created here. Please note, that *CoursePart* is an abstract type. Thus instances of its concrete sub-classes can be used here only. The last condition in Fig. 3 refines guard *notNull*. Since guards do not perform model-changing transformations but rather check for existence in the system snapshot, we just define a guard pattern here. Note that we disallow non-existence conditions in guard patterns. Else-guards are predefined by negated guard patterns i.e. it is checked for non-existence of the corresponding guard pattern.

3 Formalization by Graph Transformation

The UML variant presented in the previous section can be equipped with a graph transformation semantics. We start with presenting the theory of graph transformation as in [6] and extend it by new concepts. While class diagrams are formalized by type graphs, activities with pre- and post-conditions are mapped to graph rules. The object flow is formalized by a new concept called *partial rule dependencies*. This semantics definition serves as a basis for validating the consistency of refined activity diagrams with object flow.

3.1 Graphs and Graph Transformation

Graphs are often used as abstract representation of visual models, e.g. UML models. When formalizing object-oriented models, graphs occur at two levels: the type level (defined by a meta-model) and the instance level. This idea is described by the concept of *typed attributed graphs*, where a fixed *type graph* *TG* serves as an abstract representation of the meta-model (without constraints). Node types can be structured by an inheritance hierarchy and may be abstract in the sense that they cannot be instantiated. Multiplicities and other annotations have to be expressed by additional graph constraints. Attribute types are formally described by data type algebras. Instances of

the type graph are *object graphs* equipped with a structure-preserving mapping to the type graph. Attribute values are given by a concrete data algebra.

Graph transformation is the rule-based modification of graphs. A *rule* is defined by $p = (L \xleftarrow{l} K \xrightarrow{r} R, I, O, NACs)$ where L is the left-hand side (LHS) of the rule representing the pre-condition and R is the right-hand side (RHS) describing the post-condition. l and r are two injective graph morphisms, i.e. functions on nodes and edges which are structure and type-preserving. They specify a partial mapping $r \circ l^{-1}$ from L to R . $L \setminus l(K)$ defines the graph part that is to be deleted, and $R \setminus r(K)$ defines the graph part to be created. The types of newly created nodes have to be non-abstract. Elements in K are mapped in a type preserving way. All graphs of a rule are attributed by the same algebra being a term algebra with variables. Some of these variables are considered to be rule parameters. Input parameters can be nodes or variables, thus $I = I_N \cup I_V$, whereas output parameters can be nodes only, i.e. $O = O_N$ with $I \subseteq L$ and $O \subseteq R$.

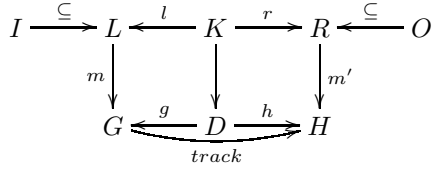
NACs is a set of negative application conditions, each defined by an injective graph morphism $n : L \rightarrow N$ where $N \setminus n(L)$ defines a forbidden graph part. n allows to refine node types, i.e. a node of a more abstract type is allowed to be mapped to a node with a finer type according to the inheritance hierarchy.

Example 1 (Example rules). Figure 3 shows example graph rules. Each pre-condition forms an LHS with one negative application condition and each post-condition describes an RHS. Identifiers given by names indicate the mapping between left- and right-hand sides. The solid parts of a pre-condition indicate the LHS L , while the dashed ones prohibit a certain graph part and represent $N \setminus n(L)$ of the NAC. Input and output parameters are listed on top of each pair of conditions, formally in the head of each rule.

A *graph transformation step* $G \xRightarrow{p,m} H$ between two instance graphs G and H is defined by first finding a match $m : L \rightarrow G$ of the left-hand side L of rule p into the current instance graph G such that m is an injective type-refining graph morphism. Match m has to fulfill the *dangling condition*, i.e. nodes may be deleted only, if all adjacent edges are mentioned in the LHS. Moreover, each NAC has to be fulfilled, i.e. m satisfies a *NAC*, if for each $n \in NACs$ there does not exist an injective type-refining morphism $o : N \rightarrow G$ such that $o \circ n = m$. Input parameters are instantiated by concrete values being nodes of the instance graph and data type values. Thus, parameter instantiation provides a partial match.

In the second step, graph H is constructed by a double-pushout construction (see [6]). Roughly spoken, the construction is performed in two passes: (1) build a graph D which contains all those elements of G not deleted; (2) construct H as a union of D and all elements of R to be created. To focus on the preserved part of a graph transformation step, we define a partial graph morphism $track : G \rightarrow H$ by $track = g^{-1} \circ h$. Graph $dom(track)$ is the subgraph of G where $track$ is defined, i.e. the domain of $track$. (See also [7] for a first definition of track morphism.) Morphisms $g : D \rightarrow G$ and $h : D \rightarrow H$ are constructed by a double-pushout as shown below. Morphism g^{-1} is always well-defined, since l is injective and the pushout construction preserves injectivity, thus g is also injective. Furthermore, a so-called co-match $m' : R \rightarrow H$ is defined by the double-pushout construction. Output parameters point to a certain part

of this co-match. Output parameters are useful for pointing to specific nodes which can be used in further transformation steps then.

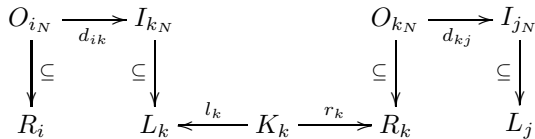


A *graph transformation (sequence)* $t = G_0 \xrightarrow{p_1, m_1} G_1 \dots G_{n-1} \xrightarrow{p_n, m_n} G_n$ consists of zero or more graph transformation steps. Track morphism $track_{0,n}$ of sequence t is simply the composition of track morphisms $track_{n-1,n} \circ \dots \circ track_{0,1}$ of its steps. For $n = 0$, $track_{0,0} = id_{G_0}$. A set of graph rules P , together with a type graph TG , is called a *graph transformation system (GTS)* $GTS = (TG, P)$. A GTS may show two kinds of non-determinism: Given a graph, (1) several rules can be applicable, and (2) for each rule several matches can exist. There are techniques to restrict both kinds of choices. The choice of rules can be restricted by the definition of control flow while the choice of matches can be restricted by passing partial matches. The tool AGG (Attributed Graph Grammar System) [8] can be used to specify and analyze graph transformation systems.

3.2 Partial Rule Dependencies

To restrict the choice of matches for rules, we introduce the concept of *partial rule dependencies* which may relate output parameter nodes of one rule to input parameter nodes of a (not necessarily direct) subsequent rule in a given rule sequence². We say that rule sequences are dependency-compatible, if the transitive closure of all dependencies between each two rules is well-defined.

Definition 1 (partial and joint rule dependencies). Given a GTS (T, P) and a rule sequence $s : p_1, \dots, p_n$ with $p_1, \dots, p_n \in P$. A partial rule dependency between rules p_i and p_j with $1 \leq i < j \leq n$ is defined by an injective partial morphism $d_{ij} : O_{i_N} \rightarrow I_{j_N}$ from output parameter nodes of p_i to input parameter nodes of p_j . If d_{ij} is the empty morphism, no rule dependency is defined. For each pair of rules p_i and p_j in s , its *closure* $closure_{ij}$ is defined as follows: (1) d_{ij} belongs to $closure_{ij}$ and (2) for all d_{ik} , d_{kj} , and rules p_k with $i < k < j$ add $d_{kj} \circ r_k \circ l_k^{-1} \circ d_{ik}$ to $closure_{ij}$.



Rule sequence s is *dependency-compatible*, if for all closures $closure_{ij}$ the following holds: (1) For all $d \in closure_{ij}$: $type(x)$ has to be finer or coarser than $type(d(x))$

² Note that rule sequences differ from transformation sequences in not providing graphs to which rules are applied.

for all $x \in O_{i_N}$ wrt. the type inheritance relation defined by type graph T . (2) Each two dependencies d and d' in closure_{ij} are weakly commutative, i.e. $d(x) = d'(x)$ for all $x \in \text{dom}(d) \cap \text{dom}(d')$.

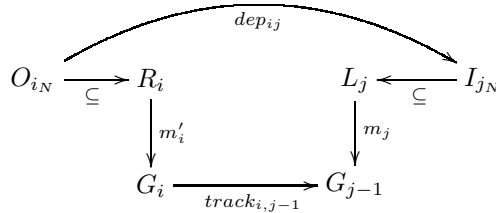
If rule sequence s is dependency-compatible, we can define a joint dependency of a closure. Given closure_{ij} we define the joint dependency $\text{dep}_{ij} : O_{i_N} \rightarrow I_{j_N}$ as follows: (1) $\text{dom}(\text{dep}_{ij}) = \bigcup_{d \in \text{closure}_{ij}} \text{dom}(d)$ and (2) $\text{dep}_{ij}(y) = d(y)$ if $y \in \text{dom}(d)$ for $d \in \text{closure}_{ij}$.

Example 2 (partial rule dependencies). Considering the rules in Fig. 3, we compose rule sequence $s = \text{CreateLecture}, \text{SetRoom}, \text{CreateLaboratory}, \text{SetRoom}$. As first step, we define partial rule dependencies taking input and output parameters into account: $d_{12}(\text{newLecture}) = \text{coursepart}$, $d_{23} = d_{34} = d_{24} = d_{14} = \emptyset$, $d_{13}(\text{newLecture}) = \text{lecture}$. All dependencies are type-compatible, since either the types of mapped nodes are equal or in hierarchy, e.g. $\text{type}(\text{newLecture}) = \text{Lecture}$ is finer than $\text{type}(d_{12}(\text{newLecture})) = \text{type}(\text{coursepart}) = \text{Coursepart}$ (see Fig. 1). None of the closures contains more than one non-empty partial dependency. Thus, partial rule dependencies are not really composed from each other in this example, e.g. $\text{dep}_{13} = d_{13}$.

If coursepart were an *inout* parameter of rule SetRoom , closure_{13} could look more interesting: With $d_{23}(\text{coursepart}) = \text{lecture}$ we would have $\text{closure}_{13} = \{d_{13}, d_{23} \circ r_2 \circ l_2^{-1} \circ d_{12}\}$ with $\text{dep}_{13}(\text{newLecture}) = \text{lecture}$.

If we enlarged the rule sequence by rule CreateLaboratory and defined $d_{34}(\text{newLab}) = \text{coursepart}$ as well as $d_{45}(\text{coursepart}) = \text{lecture}$, then dep_{35} would have to map newLab to lecture which would not be type-compatible.

Definition 2 (application of dependency-compatible rule sequences). A dependency-compatible rule sequence $s : p_1, \dots, p_n$ is applicable to some graph G_0 , if there is a graph transformation sequence $G_0 \xrightarrow{p_1, m_1} G_1 \dots G_{n-1} \xrightarrow{p_n, m_n} G_n$ such that $m_j \circ \text{dep}_{ij}$ and $\text{track}_{i,j-1} \circ m'_i(O_{i_N})$ are weakly commutative, with $\text{track}_{i,j-1}$ being the track morphism from G_i to G_{j-1} and m'_i being the co-match of rule p_i for $1 \leq i < j \leq n$.



Partial rule dependencies are defined independently of causal dependencies. Causal dependencies between rules can be analyzed by the critical pair analysis (CPA) [6]. The only kind of causal dependencies we are interested in here are *produce/use*-dependencies where the application of one rule produces an element needed by the match of a second rule. If two rules are not causally dependent on each other, the corresponding joint dependency which is defined explicitly must not introduce any *produce/use*-dependency. If some partial dependency is defined, it has to correspond with at least one *produce/use* dependency.

4 Object Flow: Semantics Definition and Properties

In this section, we first specify well-structured refined activity diagrams, refine their activities by graph rules and their guards by graph patterns, and define their semantics and consistency based on graph transformation. Thereafter, this approach is extended to refined activity diagrams with object flow.

From now on, we assume that an activity diagram does not contain any complex activities and that each complex activity has been flattened before, i.e. it has been replaced by its refining activity diagram. During this potentially recursive process, each object which goes in to or comes out from a complex activity is glued with the corresponding boundary object of the refining activity diagram, i.e. the boundary and boundary objects disappear.

4.1 Refined Activity Diagrams

As in [9,1], we restrict our considerations to well-structured activity diagrams. The building blocks are simple activities, sequences, fork-joins, decision-merge structures, and loops only.

Definition 3 (well-structured activity diagram). *A well-structured activity diagram A consists of a start activity s , an activity block B , and an end activity e such that there is a transition between s and B and another one between B and e . An activity block is defined as follows:*

- Empty: *An empty activity block is not depicted.*
- Simple: *A simple activity is an activity block.*
- Sequence: *A sequence of two activity blocks A and B connected by a transition from A to B form an activity block.*
- Decision/Merge: *A decision activity which is followed by two guarded transitions leading to one activity block each and where each block is followed by a transition both heading to a common merge activity form an activity block. One transition is explicitly guarded, called the if-guard, while the other transition carries a predefined guard "else" which equals the negated if-guard.*
- Loop: *A decision activity is followed by a guarded transition. This guard is called loop-guard. The transition leads to an activity block with an outgoing transition to the same decision activity as above. Considering this decision activity again, its incoming transition from outside becomes the incoming transition of the new block. Its outgoing transition to outside becomes the outgoing transition of the new block. This transition is guarded by "else". The whole construct forms an activity block.*
- Fork/Join: *A fork activity followed by two branches with one activity block each followed by a join activity form an activity block.*

To be able to define object flow to be coherent with control flow we define a control flow relation as prerequisite. Because of potential loops it is not a partial order.

Definition 4 (control flow relation). *The control flow relation CFR_A of an activity diagram A contains pairs (x,y) where x, y are activities such that the following holds:*

- Pair $(x, y) \in CFR_A$, if x is directly connected via a transition with y .
- If $(a, b) \in CFR_A$ and $(b, c) \in CFR_A$, then also $(a, c) \in CFR_A$.

An if- or loop-guard is equipped with a graph pattern which describes an existence condition on graphs. A guard pattern can be interpreted as identical rule (i.e. a rule where the left and the right-hand sides are equal). Guard pattern g is fulfilled by a graph G , if its corresponding rule p_g is applicable to G . After rule p_g has been performed, the guarded alternative is executed. Otherwise, rule $\bar{p}_g$ which formalizes "else" for given guard g , is applicable to G and the second alternative is performed.

Definition 5 (guard pattern, guard rule and negated guard rule). A guard pattern g is defined by a typed graph being attributed over a term algebra with variables. Its guard rule p_g is defined by $(g \xleftarrow{id_g} g \xrightarrow{id_g} g, I, O, \emptyset)$. Its negated guard rule $\bar{p}_g$ is defined by $(\emptyset \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset, \emptyset, \emptyset, \{n : \emptyset \rightarrow g\})$.

Lemma 1. Given a guard pattern g and a graph G . Rule p_g is applicable to G , iff rule $\bar{p}_g$ is non-applicable to G .

Proof. See [10].

Definition 6 (refined activity diagram). A refined activity diagram RA is a well-structured activity diagram such that each simple activity occurring in RA is equipped with a graph transformation rule. Each if- or loop-guard occurring in RA is equipped with a guard pattern. We also say that an activity is refined by a transformation rule where decision activities are refined by guard rules deduced from guard patterns which refine guards.

Definition 7 (semantics of refined activity diagrams). Given an activity block B of a refined activity diagram RA , its corresponding set of rule sequences S_B is defined as follows.

- If B is empty, $S_B = \emptyset$.
- If B consists of a simple activity a refined by rule p_a , $S_B = \{p_a\}$.
- If B is a sequence of X and Y , $S_B := S_X \text{ seq } S_Y = \{s_x s_y | s_x \in S_X \wedge s_y \in S_Y\}$
- If B is a decision block on X and Y with guard pattern g refining its if-guard, $S_B = (\{p_g\} \text{ seq } S_X) \cup (\{\bar{p}_g\} \text{ seq } S_Y)$
- If B is a loop block on X with guard pattern g refining its loop-guard, $S_B := \text{loop}(g, S_X) = \bigcup_{i \in I} S_X^i$ where $S_X^0 = \{\bar{p}_g\}$, $S_X^1 = \{p_g\} \text{ seq } S_X \text{ seq } \{\bar{p}_g\}$, $S_X^2 = \{p_g\} \text{ seq } S_X \text{ seq } S_X^1$ and $S_X^i = \{p_g\} \text{ seq } S_X \text{ seq } S_X^{i-1}$ for $i > 2$. $S_B(n) = S_X^n$ denotes the semantics of loop block B with exactly n loop executions.
- If B is a fork block on X and Y , $S_B := S_X || S_Y = \bigcup s_x || s_y$ with $s_x \in S_X \wedge s_y \in S_Y$ where $s_x || \lambda = \{s_x\}$, $\lambda || s_y = \{s_y\}$, and $p_x s'_x || p_y s'_y = \{p_x\} \text{ seq } (s'_x || p_y s'_y) \cup \{p_y\} \text{ seq } (p_x s'_x || s'_y)$.

The semantics $Sem(RA)$ of a refined activity diagram RA consisting of a start activity s , an activity block B , and an end activity e is defined as the set of rule sequences S_B generated by the main activity block B . If RA contains k guarded loops, $Sem_{n_1, \dots, n_k}(RA) \subseteq Sem(RA)$ denotes a restricted semantics where the semantics of each guarded loop $B_j \in A$ for $1 \leq j \leq k$ is $S_{B_j}(n_j)$.

Now, we are ready to check the control flow consistency of activity diagrams. To do so, we consider snapshots of the system, i.e. object models which are formalized as graphs by mapping objects to graph nodes and object links to graph edges. In the following definitions for consistency-related properties, we directly use graphs as abstract syntax representation of object models.

Activity diagrams are *consistent*, if there is a set $\mathcal{S}$ of model graphs such that each rule sequence in the diagram semantics is applicable to some of these graphs. If the diagram contains guarded loops, we use the restricted semantics for diagrams (as defined above) which checks for each guarded loop, if a predefined number of loop executions is feasible. $\mathcal{S}$ is *without junk*, if each of its model graphs represents a potential snapshot of the system to which an activity sequence in A can be applied.

Definition 8 (completeness). *A set $\mathcal{S}$ of graphs is complete wrt. to a refined activity diagram RA , if for all rule sequences s in $Sem(RA)$ there is a graph G in $\mathcal{S}$ such that s is applicable to G . If RA contains k guarded loops, a set $\mathcal{S}$ of graphs is quasi-complete wrt. to RA , if for all rule sequences s in $Sem_{n_1, \dots, n_k}(RA)$ there is a graph G in $\mathcal{S}$ such that s is applicable to G . Set $\mathcal{S}$ is without junk, if for each graph in $\mathcal{S}$ at least one applicable rule sequence in $Sem(RA)$ (resp. $Sem_{n_1, \dots, n_k}(RA)$) exists.*

Definition 9 (consistent activity diagram (without object flow)). *A refined activity diagram RA is consistent, if there is a set $\mathcal{S}$ of graphs which is complete wrt. RA . If RA contains k guarded loops, RA is quasi-consistent, if there is a set $\mathcal{S}$ of graphs which is quasi-complete wrt. RA .*

4.2 Refined Activity Diagrams with Object Flow

In the following, we define refined activity diagrams by partial rule dependencies which formalize object flows and enrich its semantics.

Definition 10 (well-structured activity diagram with coherent object flow). *A well-structured activity diagram $A_{OF} = (A, Obj, OFR, I, O)$ with coherent object flow is a well-structured activity diagram A (as given in Def. 3) equipped with a set of object nodes Obj , an object flow relation OFR for A and Obj , input parameter set I , and output parameter set O , defined as follows:*

- *Input parameters can be object nodes or values, i.e. $I = I_N \cup I_V$ with $I_N \subseteq Obj$. Output parameters may only be object nodes only, i.e. $O = O_N$ with $O \subseteq Obj$.*
- *Object flow relation OFR contains triples (x, o, y) where x and y are simple or decision activities and $o \in Obj$. In addition, there is a special tag `null` not used as activity name which is used to define object flow from and to parameter objects, i.e. triples $(null, o, y)$ and $(x, o, null)$ can also be in OFR where $o \in I_N$ or $o \in O_N$, resp. For each object o in I_N (resp. in O_N), there is a triple $(null, o, y)$ (resp. $(x, o, null)$) in OFR . For each other object $o \in Obj$, there has to be a triple $(x, o, y) \in OFR$.*
- *OFR is coherent with control flow relation CFR_A of A (see Def. 4), i.e. for all $(x, o, y) \in OFR$ with $x, y \neq null$ there is $(x, y) \in CFR_A$.*

Please note that OFR contains a triple for each pair of object flows sharing an object and Obj is not allowed to contain objects not involved in object flow.

Definition 11 (refined activity diagram with object flow). A refined activity diagram RA_{OF} with object flow is a well-structured activity diagram $A_{OF} = (A, Obj, OFR, I, O)$ with coherent object flow such that each simple activity x occurring in A_{OF} is refined by a graph transformation rule p_x . Each decision activity $x \in A_{OF}$ has an if- or loop-guard which is equipped with a guard pattern g . Its guard rule p_g is also denoted by p_x . Let O_{p_x} be the output parameter set of p_x and I_{p_y} the input parameter set of p_y . OFR has to be coherent with refining rules which is defined as follows:

- For all $(x, o, y) \in OFR$ where $x \neq null$, an output object parameter exists in O_{p_x} which is called $src(x, o, y)$. If $y \neq null$, an input object parameter exists in I_{p_y} , called $tgt(x, o, y)$.
- For all triples $(x, o, y), (x, o, y')$ (resp. $(x, o, y), (x', o, y)$) in OFR we have $src(x, o, y) = src(x, o, y')$ (resp. $tgt(x, o, y) = tgt(x', o, y)$).
- For each two activities x and y and the set of all $(x, o, y) \in OFR$, the set of all pairs $(src(x, o, y), tgt(x, o, y))$ defines an injective mapping.
- For all triples $(x, o, null), (x, o', null)$ (resp. $(null, o, y), (null, o', y)$) in OFR with $o \neq o'$ we have $src(x, o, null) \neq src(x, o', null)$ (resp. $tgt(null, o, y) \neq tgt(null, o', y)$).

Definition 12 (semantics of refined activity diagrams with object flow). The semantics $Sem(RA_{OF})$ of an activity diagram RA_{OF} with object flow being a refined activity diagram of $A_{OF} = (A, Obj, OFR, I, O)$ is equal to $Sem(RA)$, the semantics of refined activity diagram RA without object flow, where in addition partial rule dependencies (see Def. 1) are defined as follows:

For each pair of rules (p_i, p_j) in a rule sequence $s : p_1, \dots, p_n$ of $Sem(RA)$ with $1 \leq i < j \leq n$, partial rule dependency d_{ij} is defined as follows: Let x (resp. y) be the activity that is refined by rule p_i (resp. p_j) in sequence s , then the partial rule dependency d_{ij} between p_i and p_j consists of all pairs $(src(x, o, y), tgt(x, o, y))$ such that $(x, o, y) \in OFR$ where src and tgt are given by Def. 11.

RA_{OF} is called dependency-compatible, if all rule sequences in $Sem(RA_{OF})$ are dependency-compatible, as defined in Def. 2.

Definition 13 (completeness of refined activity diagrams with object flow). A set $\mathcal{S}$ of graphs is complete wrt. a dependency-compatible refined activity diagram RA_{OF} , if for all dependency-compatible rule sequences s in RA_{OF} there is a graph G in $\mathcal{S}$ such that s is applicable to G in the sense of Def. 2.

Properties quasi-completeness and consistency of refined activity diagrams without object flow can be extended to those with object flow accordingly.

Example 3 (Semantics of activity diagrams). The semantics of the flattened activity diagram *AddLecture* in Figure 2 consists of a number of rule sequences. For listing some of them, we use the following acronyms: NN=NotNull, CLec=CreateLecture, CLab=CreateLaboratory, CEx=CreateExercise, and SR=SetRoom: $Sem(RA_{OF}) \supseteq \{(CLec, \overline{NN}, \overline{NN}, \overline{NN}), (CLec, NN, SR, \overline{NN}, \overline{NN}), (CLec, \overline{NN}, NN, CLab, \overline{NN}, \overline{NN}),$

$(CLec, \overline{NN}, NN, CLab, NN, SR, \overline{NN}), (CLec, NN, SR, NN, CLab, \overline{NN}, \overline{NN}),$
 $(CLec, \overline{NN}, NN, CLab, NN, SR, NN, CEx, NN, SR),$
 $(CLec, NN, SR, NN, CLab, NN, SR, NN, CEx, NN, SR)\}$

As partly shown in Example 2, the object flow in our example can be formalized by partial rule dependencies. All rule sequences given above are dependency-compatible.

5 Related Work

This paper is rooted in formal semantics and analysis of activity diagrams as well as graph transformation approaches. While a lot of research has been done on semantics and validation of activity diagrams (see e.g. [11,12,9]), few works exist on the analysis of object flow in activity diagrams such as [13] and [14]. For example, [14] adds data flow semantics to activity diagrams by means of colored petri nets. Objects which are passed between activities have attribute value checks and method calls. Colored Petri nets provide validation like reachability of certain states and quantitative analyses as matching of time bounds. In contrast, we define a semantics for activity diagrams with object flow where activities may be refined by interrelated object diagrams which has not been done before (to the best of our knowledge).

Fujaba [15], VMTS [16], and GReAT [17] are graph transformation tools for specifying and applying object rules along a control flow specified by activity diagrams. Fujaba's story diagrams integrate activity diagrams with object rules. Compared to our approach, object flow is not depicted separately, but represented by equal names in activities. Furthermore, rules are not separated from activities. Rules used at different places have to be specified several times. We define object rules independently of activities and can apply them more than once with different arguments. VMTS and GReAT support controlled rule application with explicit control flow in a similar way and some kind of object flow. All three approaches are implemented, but do not provide a formal semantics comprising activity refinement and object flow.

6 Conclusion

In this paper, we have defined refined activity diagrams with object flow where each activity is refined by a set of interrelated object diagrams in addition, describing the pre- and post-conditions of an activity. Pre-conditions can also include non-existence conditions on object patterns. We have formalized the semantics of well-structured refined activity diagrams with coherent object flow using algebraic graph transformation where activity-refining object diagrams are defined by transformation rules. In addition, we have introduced the notion of partial dependencies between rules formalizing object flow between refined activities. To prepare a notion of consistency we define the applicability of rule sequences with partial rule dependencies.

In this paper, we have applied the approach to service modeling. Our example demonstrates how service behavior can be modeled precisely and how the coherence of its object flow can be checked. We expect that domains such as work flow design and aspect-oriented modeling can benefit from the application of our concepts as well. In future, we want to use the formal semantics given by graph transformation to prove the

consistency of refined activity diagrams with object flow along sufficient criteria easy to check. We expect that the graph transformation environment AGG can do a good job to support automatic checks.

References

1. Lambers, L., Jurack, S., Mehner, K., Taentzer, G.: Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 341–355. Springer, Heidelberg (2008)
2. Hausmann, J., Heckel, R., Taentzer, G.: Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In: Proc. of Int. Conference on Software Engineering 2002, Orlando, USA (2002)
3. Mehner, K., Monga, M., Taentzer, G.: Interaction Analysis in Aspect-Oriented Models. In: International Conference on Requirements Engineering RE 2006 (2006)
4. Lambers, L., Mariani, L., Ehrig, H., Pezzè, M.: A formal Framework for Developing Adaptable Service-Based Applications. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 392–406. Springer, Heidelberg (2008)
5. UML: Unified Modeling Language (2008), <http://www.uml.org>
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
7. Plump, D.: Evaluation of Funtional Expressions by Hypergraph Rewriting. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik (1993)
8. AGG: AGG Homepage, <http://tfs.cs.tu-berlin.de/agg>
9. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities Using Dynamic Meta Modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 76–90. Springer, Heidelberg (2007)
10. Jurack, S., Lambers, L., Mehner, K., Taentzer, G., Wierse, G.: Object Flow Definition for Refined Activity Diagrams - Long Verison. Technical Report 2009-1, Technische Universität Berlin (2009)
11. Eshuis, R., Wieringa, R.: Tool support for Verifying UML Activity Diagrams. IEEE Trans. on Software Eng. 7(30) (2004)
12. Störrle, H., Hausmann, J.H.: Towards a Formal Semantics of UML 2.0 Activities. In: Software Engineering 2005. LNI P-64, Gesellschaft f. Informatik, pp. 117–128 (2005)
13. Barros, J.P., Gomes, L.: Actions as Activities and Activities as Petri nets. In: Workshop on Critical Systems Development with UML. In: 20–24 workshop at 6. Int. Conf. on the Unified Modeling Language (UML 2003), San Francisco, U.S.A (2003)
14. Störrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. Electronic Notes in Theoretical Computer Science, vol. 117 (2003)
15. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
16. Visual Modeling and Transformation System (2008), <http://vmts.aut.bme.hu/>
17. GReAT - Graph Rewriting and Transformation (2008), <http://www.isis.vanderbilt.edu/tools/GReAT>

A Category-Theoretical Approach to the Formalisation of Version Control in MDE

Adrian Rutle¹, Alessandro Rossini², Yngve Lamo¹, and Uwe Wolter²

¹ Bergen University College, P.O. Box 7030, 5020 Bergen, Norway
{aru,yla}@hib.no

² University of Bergen, P.O. Box 7803, 5020 Bergen, Norway
{rossini,wolter}@ii.uib.no

Abstract. In Model-Driven Engineering (MDE) models are the primary artefacts of the software development process. Similar to other software artefacts, models undergo a complex evolution during their life cycles. Version control is one of the key techniques which enables developers to tackle this complexity. Traditional version control systems are based on the *copy-modify-merge* paradigm which is not fully exploited in MDE because of the lack of model-specific techniques. In this paper we give a formalisation of the copy-modify-merge paradigm in MDE. In particular, we analyse how common models and merge models can be defined by means of category-theoretical constructions. Moreover, we show how the properties of those constructions can be used to identify model differences and conflicting modifications.

1 Introduction and Motivation

Since the beginning of computer science, raising the abstraction level of software systems has been a continuous process. One of the latest steps in this direction has lead to the usage of modelling languages in software development processes. Software models are indeed abstract representations of software systems which are used to tackle the complexity of present-day software by enabling developers to reason at a higher level of abstraction. In Model-Driven Engineering (MDE) models are first-class entities of the software development process and undergo a complex evolution during their life-cycles. As a consequence, the need for techniques and tools to support model evolution activities such as version control is increasingly growing.

Present-day MDE tools offer a limited support for version control of models. Typically, the problem is addressed using a *lock-modify-unlock* paradigm, where a repository allows only one developer to work on an artefact at a time. This approach is workable if the developers know who is planning to do what at any given time and can communicate with each other quickly. However, if the development group becomes too large or too spread out, dealing with locking issues might become a hassle.

On the contrary, traditional version control systems such as Subversion enable efficient concurrent development of source code. These systems are based on the *copy-modify-merge* paradigm. In this approach each developer accesses a repository and creates a local working copy – a snapshot of the repository’s files and directories. Then, the developers modify their local copies simultaneously and independently. Finally, the local modifications are merged into the repository. The version control system assists with

the merging by detecting conflicting changes. When a conflict is detected, the system requires manual intervention of the developer.

Unfortunately, traditional version control systems are focused on the management of text-based files, such as source code. That is, difference calculation, conflict detection, and source code merge are based on a per-line textual comparison. Since the structure of models is graph-based rather than text- or tree-based [1], the existing techniques are not suitable for MDE.

During the last years, research has lead to various outcomes related to model evolution: [2] for the difference calculation, [3] for the difference representation, [4] for the conflict detection, and [5] for syntactic software merging that exploits the graph-based structure(s) of programs, to cite a few. However, the proposed solutions are not formalised enough to enable automatic reasoning about model evolution. For example, operations such as *add*, *delete*, *rename* and *move* are given different semantics in different works/tools. In addition, concepts such as *synchronisation*, *commit* and *merge* are only defined semiformaly. Moreover, the terminology is not precise and unique, e.g. the terms “create”, “add” and “insert” are frequently used to refer to the same operations.

Our claim is that the adoption of the copy-modify-merge paradigm is necessary to enable effective version control in MDE. This adoption requires formal techniques which are targeting graph-based structures. The goal of this paper is the formalisation of the copy-modify-merge paradigm in MDE. In particular, we show that common models and merge models can be defined as pullback and pushout constructions, respectively. For our analysis we use the Diagram Predicate Framework (DPF)¹ [6,7,8] which provides a formal approach to modelling based on category theory – the mathematics of graph-based structures. In addition, DPF enables us to define a language to represent model differences and a logic to detect conflicting modifications.

The rest of the paper is structured as follow. Section 2 provides a brief introduction to DPF. Then Section 3 outlines a motivating example, and gives the formalisation of the concepts of version control. In Section 4 the state-of-the-art of research in version control is summarised. Finally, in Section 5 some concluding remarks and ideas for future work are given.

2 Diagram Predicate Framework

Diagram Predicate Framework (DPF), is a diagrammatic formalism for the definition and reasoning about modelling languages, (meta)models and model transformations. The formalism is based on category theory and first order logic; it combines the mathematical rigour – which is necessary to enable automatic reasoning – with the intuitiveness of diagrammatic notations [9]. DPF’s usage in the formalisation of concepts in (meta)modelling and model transformations are discussed in [8] and [7,10], respectively. This section includes only a short description of the basic concepts of DPF such as *signatures*, *constraints* and *diagrammatic specifications*.

In DPF, software models are represented by diagrammatic specifications². These diagrammatic specifications are structures which consist of a graph and a set of constraints.

¹ Formerly named Diagrammatic Predicate Logic (DPL).

² In this paper the terms *model* and *diagrammatic specification* are used interchangeably.

The graph represents the structure of the model. Predicates from a predefined diagrammatic signature are used to add constraints to the graph [6]. Each modelling language L corresponds to a diagrammatic signature Σ_L and a metamodel MM_L . L -models are represented by Σ_L -specifications where the underlying graphs are instances of the metamodel MM_L [8]. Signatures, constraints and diagrammatic specifications are defined as follows:

Definition 1 (Signature). A (diagrammatic predicate) signature $\Sigma := (\Pi, \alpha)$ is an abstract structure consisting of a collection of predicate symbols Π with a mapping that assigns an arity (graph) $\alpha(p)$ to each predicate symbol $p \in \Pi$.

Definition 2 (Constraint). A constraint (p, δ) in a graph G is given by a predicate symbol p and a graph homomorphism $\delta : \alpha(p) \rightarrow G$, where $\alpha(p)$ is the arity of p .

Definition 3 (Diagrammatic Specification). A Σ -specification $S := (G(S), S(\Pi))$, is given by a graph $G(S)$ and a set $S(\Pi)$ of constraints (p, δ) in $G(S)$ with $p \in \Pi$.

Table 1 shows a sample signature $\Sigma = (\Pi, \alpha)$ which consists of a collection of useful predicates such as [cover], [key] etc. The first column of the table shows the names

Table 1. A sample signature Σ

Π	α	Proposed visualisation	Intended semantics
[total]	$1 \xrightarrow{x} 2$		$\forall a \in A : f(a) \geq 1$
[key]	$1 \xrightarrow{x} 2$		$\forall a, a' \in A : a \neq a' \text{ implies } f(a) \neq f(a')$
[single-valued]	$1 \xrightarrow{x} 2$		$\forall a \in A : f(a) \leq 1$
[cover]	$1 \xrightarrow{x} 2$		$\forall b \in B : \exists a \in A \mid b \in f(a)$
[isA]	$1 \xrightarrow{x} 2$		$f = \phi \text{ where } \phi : B \rightarrow A \text{ is a persistent extension and } \phi \text{ is its reduct.}$
[containment]	$1 \xrightarrow{x} 2$		$\forall b \in B, \exists a \in A \mid b \in f(a) \text{ and } \forall g : X \rightarrow B, \forall x \in X \text{ if } b \in g(x) \text{ then } f = g, X = A \text{ and } a = x$
[inverse]	$1 \begin{matrix} \xrightarrow{x} \\ \xleftarrow{y} \end{matrix} 2$		$\forall a \in A, \forall b \in B : b \in f(a) \text{ iff } a \in g(b)$
[jointly-key]	$1 \xrightarrow{x} 2$ $\downarrow y$ 3		$\forall a, a' \in A : a \neq a' \text{ implies } f(a) \neq f(a') \text{ or } g(a) \neq g(a')$

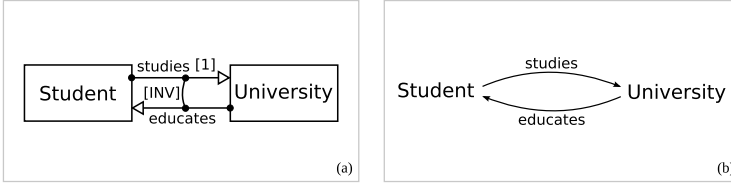


Fig. 1. A Diagrammatic Specification: (a) $S = (G(S), S(II))$, (b) its graph $G(S)$

of the predicates. The second and the third columns show the arities and a possible visualisation of these predicates, respectively. In the fourth column, the intended semantic of each predicate is specified. These predicates in Table 1 allow for specifying some useful properties and constraints that a modeller would define for a structural model. In addition, the signature can be extended with custom-defined predicates. Typically in structural models, model elements are interpreted as sets and arrows as multivalued functions $f : A \rightarrow \wp(B)$, i.e. an arrow without constraints stands for an arbitrary multivalued function. For example, in UML class diagrams the intended semantics of an association between two classes is that the instances of those two classes have a many-to-many relationship.

Fig. 1a shows an example of a Σ -specification $S = (G(S), S(II))$. S specifies the structural model of a simple information system for universities. $G(S)$ in Fig. 1b is the graph of S without any constraints on it. In S , every university educates *one or more* students; this is forced by the constraint $([total], \delta_1)$ on the arrow *educates* (see Table 2). Moreover, every student studies at *exactly one* university; this is forced by the constraint $([single-valued], \delta_2)$ on the arrow *studies*. Another property of S is that the functions *studies* and *educates* are inverse of each other, i.e. $\forall u \in University : u = studies(educates(u))$ and $\forall s \in Student : s \in educates(studies(s))$. This is forced by the constraint $([inverse], \delta_4)$ on *studies* and *educates*.

Table 2. Diagrams $(p, \delta) \in S(II)$

(p, δ)	$\alpha(p)$	$\delta(\alpha(p))$
$([total], \delta_1)$	$1 \xrightarrow{x} 2$	$University \xrightarrow{educates} Student$
$([single-valued], \delta_2)$	$1 \xrightarrow{x} 2$	$Student \xrightarrow{studies} University$
$([cover], \delta_3)$	$1 \xrightarrow{x} 2$	$University \xrightarrow{educates} Student$
$([inverse], \delta_4)$	$1 \begin{array}{c} \xrightarrow{x} \\ \xleftarrow{y} \end{array} 2$	$Student \begin{array}{c} \xrightarrow{studies} \\ \xleftarrow{educates} \end{array} University$

3 Version Control in MDE

The problem of version control in MDE is formalised in terms of category-theoretical constructs. It should be noted that our reasoning is applicable both at model and meta-model levels.

First we start with an example to present a usual scenario of concurrent development in MDE. In our examples we use diagrammatic specifications defined by means of DPF. The example is obviously simplified and only the details which are relevant for our discussion are presented. Then, common models, merge models and their computations are analysed in the subsequent sections.

Suppose that two software developers, Alice and Bob, use a version control system based on the copy-modify-merge paradigm. The scenario is depicted in Fig. 2, while an overview of the models in the example is shown in Fig. 3.

Alice checks out a local copy of the model V_1 (Fig. 1) from the repository and modifies it to V_{1A} , where 1 is a version number and A stands for Alice. In particular, she adds the node `PhDStudent` as an extension of `Student`, together with the arrow `enrols`. This modification takes place in the *evolution step* e_{1A} . Since the model in the repository may have been updated in the mean time, she needs to synchronise her model with the repository in order to integrate her local copy with other developers' modifications. This is done in the *synchronisation* s_{1A} . However, no modifications of the model V_1 has taken place in the repository while Alice was working on it. Therefore, the synchronisation completes without changing the local copy V_{1A} . Finally, Alice commits the local copy, which will be labelled V_2 in the repository (Fig. 3a). This is done in the *commit* c_{1A} .

Afterwards, Bob checks out a local copy of the model V_2 from the same repository and modifies it to V_{2B} . In particular, he takes into consideration also *Postdoc* as a different type of student; to avoid the pollution of extensions in the model he deletes the `PhDStudent` node, and refactors the model by adding a new node `Enrolment`. Then, he synchronises his model with the repository. Again, the synchronisation completes without changing the local copy V_{2B} . Finally, Bob commits the local copy, which will be labelled V_3 in the repository (Fig. 3b).

Alice continues working on her local copy, which is still V_2 and is not synchronised with the repository which contains Bob's modifications. She adds a node `Project`

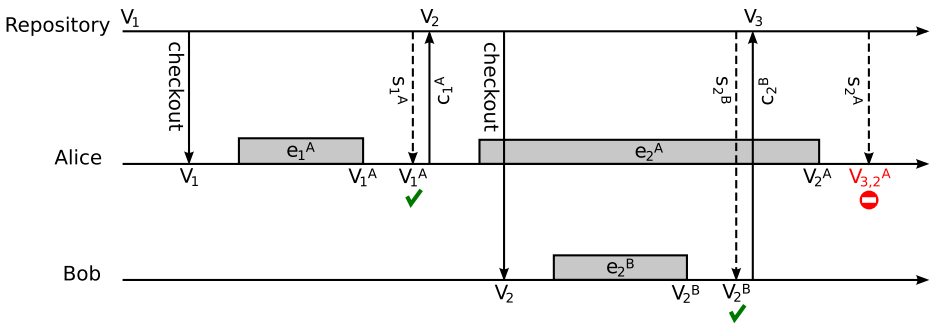


Fig. 2. The timeline of the example

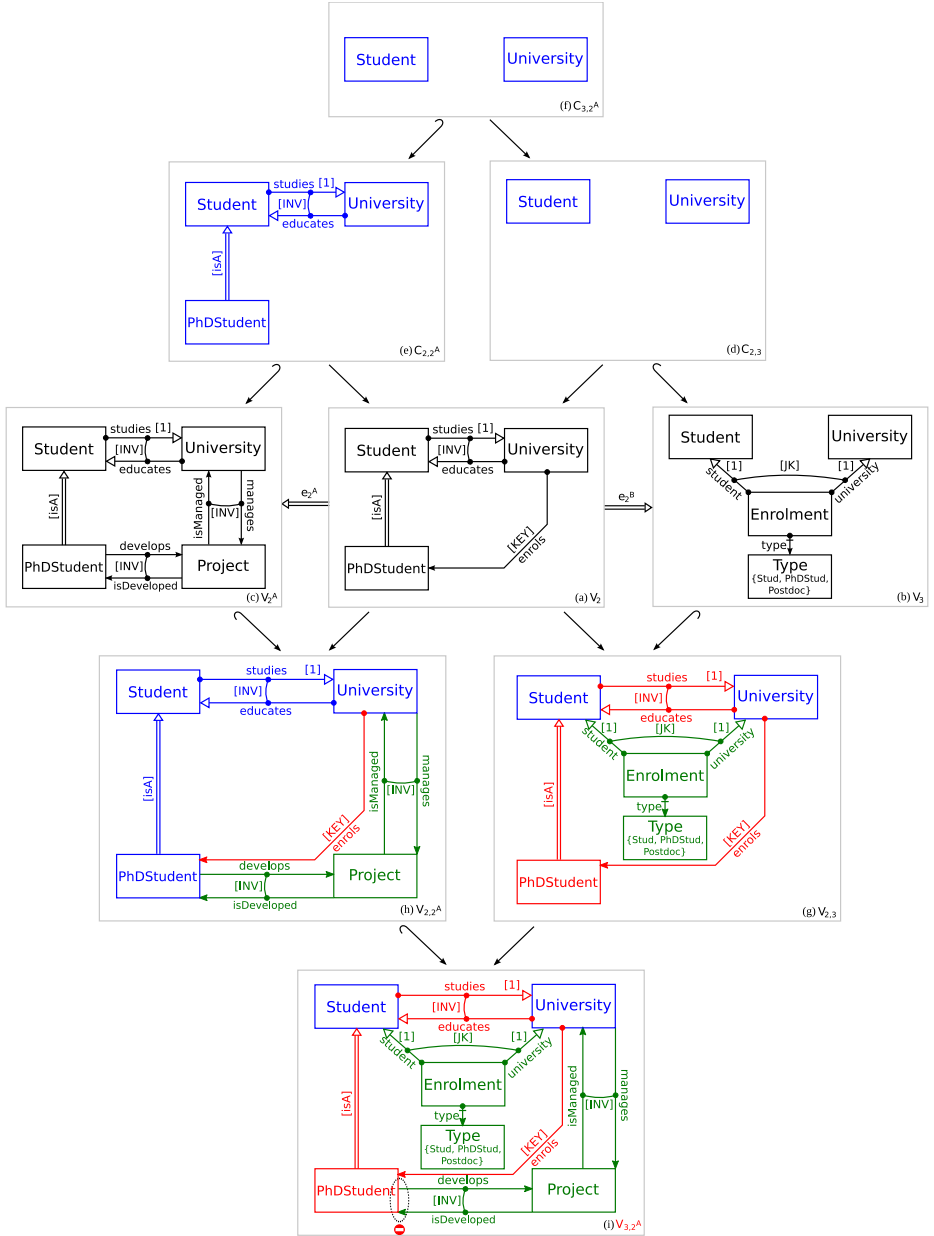


Fig. 3. The models of the example

(Fig. 3c). She synchronises her model with the repository where the last model is V_3 . Hence, the synchronisation computes the *merge model* $V_{3,2^A}$ (Fig. 3i). Now, the version control system reports a conflict in the merge model which forbids the commit c_{2^A} . This is because the node `PhDStudent` has been deleted by Bob, but Alice has added some arrows from/to it. The resolution of the conflict requires the manual intervention of Alice, who must review the model and decide to adapt it to Bob’s modifications, or, adapt Bob’s modifications to her own model.

3.1 Common Model

When Alice changes her local copy from V_2 to V_{2^A} , her development environment must keep track over what is common between the two models. The identification of what is common is the same as the identification of what is not modified, which should be feasible to implement in any tool.

Every two model elements which correspond to each other can be identified in a *common model*. For example, the model $C_{2,2^A}$ (Fig. 3e) is a common model of the models V_2 and V_{2^A} . The usage of a common model makes the construction of merge models at synchronisation step easy (explained in Sec. 3.2, 3.3). In some frameworks, what is common between two models is defined implicitly by stating that structurally equivalent elements imply that the elements are equal (*soft-linking*). This approach has the benefit of being general, but its current implementations are too resource greedy to be used in production environment. In other frameworks, elements with equal identifiers are seen as equal elements (*hard-linking*). Unfortunately, this approach is tool-dependent, since the element identification is different for every environment. Our claim is that “recording” which elements are kept unmodified during an evolution step addresses the problems of the soft- and hard-linking approaches. That is, these equalities are specified explicitly in common models as in the following definition (see Fig. 4).

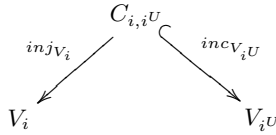


Fig. 4. Common model

Definition 4 (Common Model). A model C_{i,i^U} together with the injective morphism inj_{V_i} and the inclusion morphism $inc_{V_i^U}$ is a common model for V_i and V_{i^U} .

Note that we support renaming operations by allowing arbitrary injective morphisms inj_{V_i} . We decided, however, that the common model contains always the most recent names by requiring that the $inc_{V_i^U}$ are inclusions.

In order to find the common model between two models which are not subsequent versions of each other, i.e. for which we do not have a direct common model, we can construct the common model by the composition of the common models of their intermediate models. For example, the model $C_{3,2^A}$ (Fig. 3f) is the common model of the

models V_3 and V_{2A} . We call this common model for the *composition of commons* or the *normal form*. A possible way to compute this common model is as follows (see Fig. 5):

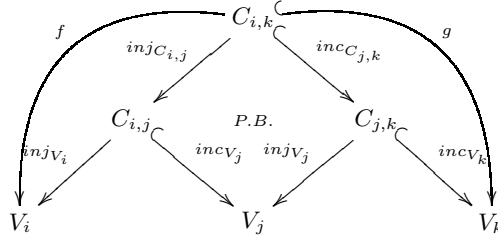


Fig. 5. Common models: $C_{i,j}$ and $C_{j,k}$; and the composition: $C_{i,k}$

Definition 5 (Composition of Commons). Given the diagrams $V_i \xleftarrow{inj_{V_i}} C_{i,j} \xrightarrow{inc_{V_j}} V_j$ and $V_j \xleftarrow{inj_{V_j}} C_{j,k} \xrightarrow{inc_{V_k}} V_k$ the common model for V_i and V_k is $C_{i,k}$ with the two morphisms f and g where $f = inj_{C_{i,j}}; inj_{V_i}$, $g = inc_{C_{j,k}}; inc_{V_k}$, and, $C_{i,k}$ is a pull-back ($C_{i,k}$, $inj_{C_{i,j}} : C_{i,k} \rightarrow C_{i,j}$, $inc_{C_{j,k}} : C_{i,k} \rightarrow C_{j,k}$) of the diagram $C_{i,j} \xrightarrow{inc_{V_j}} V_j \xleftarrow{inj_{V_j}} C_{j,k}$ such that $inc_{C_{j,k}}$ is an inclusion.

3.2 Merge Model

Recall that when Alice wanted to commit her local copy V_{2A} to the repository, she had to first synchronise it with the repository. In the synchronisation s_{2A} , a merge model $V_{3,2A}$ was created (Fig. 3i). The merge model must contain the information which is needed to distinguish which model elements come from which model. Since this is exactly one of the properties of pushout, we use pushout construction to compute merge models, as stated in the next definition (see Fig. 6).

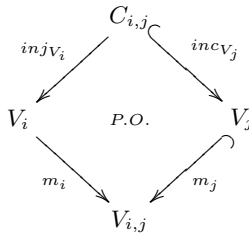


Fig. 6. Merge model

Definition 6 (Merge Model). Given the models V_i , V_j and $C_{i,j}$, the merge model $V_{i,j}$ is the pushout ($V_{i,j}$, $m_i : V_i \rightarrow V_{i,j}$, $m_j : V_j \rightarrow V_{i,j}$) of the diagram

$$V_i \xleftarrow{inj_{V_i}} C_{i,j} \xrightarrow{inc_{V_j}} V_j \text{ such that } m_j \text{ is an inclusion.}$$

The properties of the pushout are then used to decorate merge models such that added, deleted, moved, and renamed elements are distinguished (explained in Sec. 3.4).

3.3 Synchronisation and Commit

Fig. 7 outlines synchronisation and commit operations in the copy-modify-merge paradigm. These operations are defined as follows. In Fig. 7 and in the following definitions and propositions, U stands for “username”.

Definition 7 (Synchronisation). *Given the local copy V_{iU} , the last model in the repository V_j and their merge model $V_{j,iU}$, the synchronisation $s_{iU} : (V_{iU}, V_j) \rightarrow V_{jU}$ is an operation which generates a synchronised local copy V_{jU} such that*

$$V_{jU} := \begin{cases} V_{iU} & \text{if } i = j; \\ V_{j,iU} & \text{if } i < j, \text{ and } V_{j,iU} \notin \mathcal{C}^U \end{cases} \quad \text{with } \mathcal{C}^U \text{ the set of conflicting merge models.}$$

Definition 8 (Commit). *Given the synchronisation $s_{iU} : (V_{iU}, V_j) \rightarrow V_{jU}$, the commit $c_{iU} : V_{jU} \Rightarrow V_{j+1}$ is an operation which adds the model V_{jU} to the repository as V_{j+1} .*

Whenever a local copy V_{iU} is synchronised with a model V_j from the repository, if the version numbers are the same, i.e. $i = j$, then a *synchronised* local copy V_{jU} will be created such that $V_{jU} = V_{iU}$. However, if $i < j$, then a merge model $V_{j,iU}$ will be created such that $V_{jU} = V_{j,iU}$, only if $V_{j,iU}$ is not in a conflict state (explained in Sec. 3.4), i.e. $V_{j,iU} \notin \mathcal{C}^U$. Finally, the commit operation will add the synchronised local copy V_{jU} to the repository and will label it V_{j+1} . The next procedure explains the details of our approach to the synchronisation and commit operation (see Fig. 7).

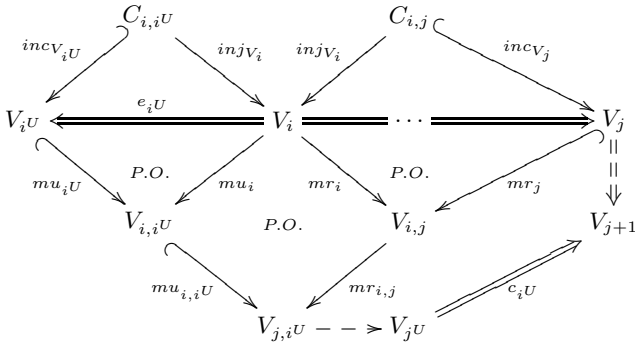


Fig. 7. Synchronisation and Commit

Procedure 9 (Synchronisation Procedure). *Given the models V_{iU} , V_i , $C_{i,iU}$ and V_j , where $i < j$, the synchronisation $s_{iU} : (V_{iU}, V_j) \rightarrow V_{jU}$ is computed as follows:*

1. compute the merge model $(V_{i,iU}, mu_{iU}, mu_i)$ as a pushout of

$$V_{iU} \xleftarrow{inc_{V_{iU}}} C_{i,iU} \xrightarrow{inj_{V_i}} V_i$$

2. compute the common model $(C_{i,j}, inj_{V_i}, inc_{V_j})$ as a pullback of

$$V_i \xrightarrow{mr_i} V_{i,j} \xleftarrow{mr_j} V_j$$

3. compute the merge model $(V_{i,j}, mr_i, mr_j)$ as a pushout of

$$V_i \xleftarrow{inj_{V_i}} C_{i,j} \xrightarrow{inc_{V_j}} V_j$$

4. compute the merge model $(V_{j,i^U}, mu_{i,i^U}, mr_{i,j})$ as a pushout of

$$V_{i,i^U} \xleftarrow{mu_i} V_i \xrightarrow{mr_i} V_{i,j}$$

5. $V_{j^U} := V_{j,i^U}$ only if $V_{j,i^U} \notin \mathcal{C}^U$

3.4 Difference and Conflict

As mentioned, during a synchronisation operation $s_{i^U} : (V_{i^U}, V_j) \rightarrow V_{j^U}$ where $i < j$, the merge model V_{j,i^U} may contain conflicts. To detect these conflicts, we need a way to identify the differences between V_{i^U} and V_j , i.e. the modifications which has occurred in the evolution step(s). Difference identification in the merge model V_{j,i^U} can be done by distinguishing common elements, V_{i^U} -elements and V_j -elements from each other. However, since this is one of the properties of merge models, we already have all the information we need to identify the differences and, we only need a language to represent these differences. Moreover, since software models are graph-based structures, we need a diagrammatic language for this purpose. The language must enable tagging model elements as *common*, *added*, *deleted*, *renamed* and *moved*. We use DPF to define such a diagrammatic language, Δ , for the representation of model differences.

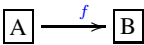
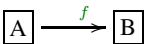
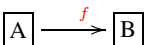
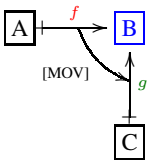
The language Δ is represented by the signature $\Sigma_\Delta = (\Pi_\Delta, \alpha_\Delta)$ which consists of five predicates: [common], [add], [delete], [rename], and [move] (see Table 3). The merge models will be *decorated* by predicates from the signature Σ_Δ in addition to the predicates from the signature which represents the modelling language.

Each of [common], [add] and [delete] has two arities: 1 and $1 \xrightarrow{x} 2$. That is, each of these predicates can be used to tag either a node or an arrow. For example, Bob has added the node `Enrolment` and the arrows `student` and `university` in the model V_3 (Fig. 3b). These added elements are coloured green, i.e. tagged as *added*, in the merge model $V_{2,3}$ (Fig. 3g) since the visualisation of the predicate [add] in Σ_Δ is green.

For the predicate [rename], when an element $A \in V_i$ is renamed to $B \in V_{i^U}$, the common model C_{i,i^U} will contain B with $inj_{V_i}(B) = A$ and $inc_{V_{i^U}}(B) = B$. Moreover, the visualisation will be $\boxed{B}$ [REN: $A \mapsto B$] in the merge model V_{i,i^U} . The morphism inj_{V_i} is injective in order to allow for this renaming. Moreover, the morphism $inc_{V_{i^U}}$ is inclusion so that the common- and the merge models always contains the new name. However, when V_{j,i^U} is a merge model for $j > i$, then the visualisation will be $\boxed{C}$ [REN: $C \mapsto Y$], where $C \in V_i$ is the old name, and $Y \in V_j$ or $Y \in V_{i^U}$ is the new name. This is due to the commutative property of pushouts. Fig. 8 shows an example of renaming, where `Employee` is renamed in an evolution step e_{1^A} to `Person`.

In general, the predicate [move] is used when the source of the reference to a contained model element is changed from a container to another. In object oriented models, e.g. in class diagrams, this operation is usually used in two cases; when an attribute or a method of a class is moved to another class, and, when a class is moved from a package to another. An example of the usage of a move operation is shown in Fig. 8, where the attribute `salary` is moved to the *objectified* relationship `Employment`.

Table 3. The signature Σ_Δ . $A, B, C, f, g \in V_{i,j}$. $V_{i,j}$ and $C_{i,j}$ are the merge and common models, respectively, of V_i and V_j , with $i < j$.

Π_Δ	α_Δ	Proposed visualisation	Intended semantics
$[\text{common}]^n$	1		$A \in V_i$ and $A \in V_j$
$[\text{common}]^a$	$1 \xrightarrow{x} 2$		$f \in V_i$ and $f \in V_j$
$[\text{add}]^n$	1		$A \notin V_i$ and $A \in V_j$
$[\text{add}]^a$	$1 \xrightarrow{x} 2$		$f \notin V_i$ and $f \in V_j$
$[\text{delete}]^n$	1		$A \in V_i$ and $A \notin V_j$
$[\text{delete}]^a$	$1 \xrightarrow{x} 2$		$f \in V_i$ and $f \notin V_j$
$[\text{rename}]$	1		$\exists C \in C_{i,j} : \text{inj}_{V_i}(C) = X$ and $\text{inc}_{V_j}(C) = Y$ where $\text{inj}_{V_i} : C_{i,j} \rightarrow V_i$ and $\text{inc}_{V_j} : C_{i,j} \rightarrow V_j$
$[\text{move}]$	$1 \xrightarrow{x} 2$ $\quad \quad \quad \uparrow$ $\quad \quad \quad y$ $\quad \quad \quad 3$		$f \in V_i$ and $f \notin V_j$ and $g \in V_j$ and $g \notin V_i$ and $B \in C_{i,j}$ and both f and g are containment arrows as defined in Table 1

The synchronisation procedure we have developed uses Σ_Δ for two main purposes:

- to *reduce* the decorated merge model V_{j,i^U} according to the rules in Table 4, e.g. if a model element is tagged with both $[\text{common}]$ and $[\text{delete}]$, it will be tagged only with $[\text{delete}]$ in V_{j,i^U} .
- to obtain the synchronised local copy V_{j^U} from V_{j,i^U} by *interpreting* the predicates as operations, e.g. if a model element is tagged with the predicate $[\text{delete}]$, it will not exist in V_{j^U} (see Table 4).

If the reduced merge model V_{j,i^U} contains the predicate $[\text{conflict}]$, then $V_{j,i^U} \in \mathcal{C}^U$, i.e. it is in a state of conflict. Although conflicts are context-dependent, we have recognised some situations where syntactic conflicts will arise. The definition of new rules/conflicting situations is also allowed in DPF. The following is a summary of the concurrent modifications which we identify as conflicts:

- adding structure to an element which has been deleted
- renaming an element which has been renamed
- moving an element which has been moved

In Table 4, the predicates in $V_{j,i^U}(\Pi_\Delta)$ are written in the form $([p], \delta_x^v)$ with v a version number and $p \in \Pi_\Delta$, where v is used to distinguish between predicates which come from V_{i,i^U} and $V_{i,j}$ (see Def. 2). For example, $[\text{common}](X)$ in the first column is an abbreviation for $(([\text{common}]^n, \delta_x^{i^U}) : 1 \mapsto X) \in V_{j,i^U}(\Pi_\Delta)$ for $x \in \mathbb{N}$. That

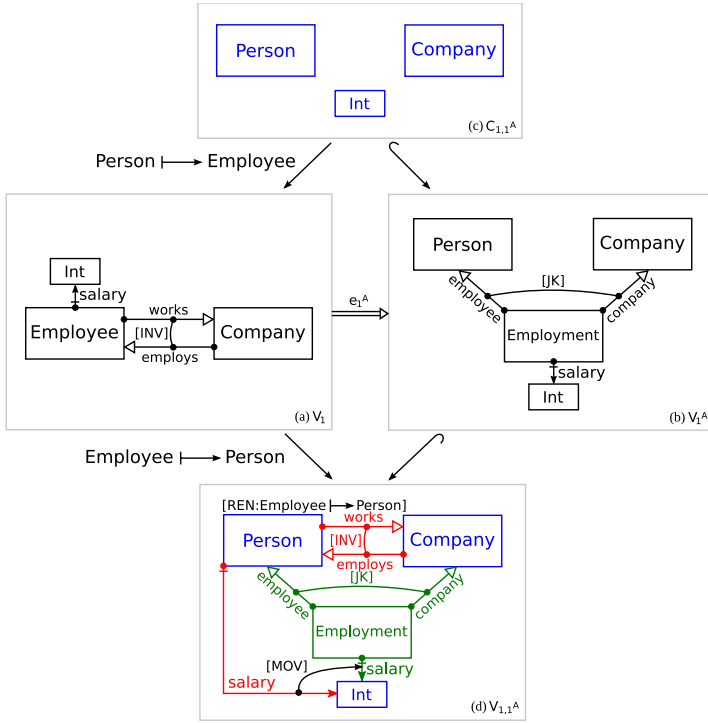


Fig. 8. Examples of the predicates [move] and [rename]

Table 4. A subset of the rules used for the reduction of $V_{j,iU}$ and to obtain V_{jU} from $V_{j,iU}$. $X, f \in V_{j,iU}$

$V_{i,iU}(\Pi_\Delta)$	$V_{i,j}(\Pi_\Delta)$	$V_{j,iU}(\Pi_\Delta)$	$\text{In } V_{jU}$
[common] (X)	[common] (X)	[common] (X)	remains
[delete] (X)	[delete] (X)	[delete] (X)	deleted
[common] (X)	[delete] (X)	[delete] (X)	deleted
[delete] (X)	[common] (X)	[delete] (X)	deleted
[move] (X)	[move] (X)	[conflict] (X)	$\perp$
$[\text{add}]^a(f)$	$[\text{delete}]^n(\text{src}(f))$	$[\text{conflict}](f, \text{src}(f))$	$\perp$
$[\text{add}]^a(f)$	$[\text{delete}]^n(\text{trg}(f))$	$[\text{conflict}](f, \text{trg}(f))$	$\perp$
[rename] (X)	[rename] (X)	[conflict] (X)	$\perp$

is, the predicate $[\text{common}]^n$ comes from the model $V_{i,iU}$. Moreover, $\perp$ means that the synchronised local model V_{jU} will not be created.

4 Related Work

The literature related to model evolution and in particular version control is becoming abundant. Firstly, we have the works that describe how to compute the difference of models: EMF Compare [2] and DSMDiff [11] are two model differencing tools which are based on a similar technique. The difference calculation is divided in two phases. The first is the detection of model mappings, where all the elements of the two input models are compared using metrics like signature matching and structural similarity. The second phase is the determination of model differences, where all the additions, deletions and changes are detected. This approach has the great benefit of being general, but at the price of being resource greedy.

Secondly, there are works which analyse how to represent differences among models conforming to an arbitrary metamodel. There are different approaches for the representation of model differences:

1. As models which conform to a difference metamodel. The difference metamodel can be generic [12], or obtained by an automated transformation [3]. Those models are in general minimalistic (i.e. only the necessary information to represent the difference is presented), transformative (i.e. each difference model induce a transformation), compositional (i.e. difference models can be composed sequentially or in parallel), and typically symmetric (i.e. given a difference representation we can compute the inverse of it).
2. As a model which is the union of the two compared models, with the modified elements highlighted by colours, tags, or symbols [13], which is similar to our visualisation. The adoption of this technique is typically beneficial for the designer, since the rationale of the modifications is easily readable. However, these quality factors are retained only if the base models are not large and not too many updates apply to the same elements, since the difference model consists of both base models to denote the differences.
3. As a sequence of atomic actions specifying how the initial model is procedurally modified [14]. While this technique has the great advantage of being very efficient, the difference representation is not readable and intuitive. In addition, edit scripts do not follow the “everything is a model vision”. They are suitable for internal representations but quite ineffective to be adopted for documenting changes in MDE environments.

Thirdly, there are works aimed at identifying the types of structural and semantic conflicts that can occur in distributed development. In [4] a predefined set of *a priori* conflicts is identified, stating that it is not possible to provide a generic technique for conflict detection with an arbitrary accuracy. However, in [15,16] the authors propose a Domain-Specific Modelling Language for the definition of weaving models which represent custom conflicting patterns. Moreover, it is possible to describe the resolution criteria through OCL expressions.

5 Conclusion and Future Work

In this paper, category-theoretical constructs are used to formalise concepts used in version control. Usual operations such as checkout, synchronise and commit that a developer perform in a distributed development are analysed. Moreover, the concepts of common and merge models are introduced and defined as pullback and pushout, respectively. In addition we defined a language Δ – specified as the signature Σ_{Δ} in DPF – which we have used to formalise model differences. The predicates of Σ_{Δ} enable the reasoning about and presentation of operations such as add, delete, move, and rename. That is, model elements which has been added, deleted, moved or renamed are tagged by predicates from Σ_{Δ} . Finally, we described how these predicates can be used for the identification of possible conflicting modifications. DPF has shown to have the expressiveness and flexibility which are required to define the language Δ .

The proposed approach to version control in MDE differs from the approaches in the related work mentioned above since it is based on common models instead of difference models. The difference between two models is identified by means of category-theoretical constructs and represented through the language Δ .

In this work, we focused only on the detection of a predefined set of syntactic conflicts which are derived from experience. In a future work, we analyse and formalise semantic conflicts, i.e. modifications which are violating metamodel constraints or predicate dependencies. Moreover, a prototype implementation of these techniques will be necessary to show the efficiency of the proposed techniques. This is a challenging task, considering the lack of mature standards and the issues related to the identification of model elements [17].

References

1. Baresi, L., Heckel, R.: Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 431–433. Springer, Heidelberg (2004)
2. Brun, C., Musset, J., Toulmé, A.: EMF Compare Project, <http://www.eclipse.org/emft/projects/compare/>
3. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology* 6(9), 165–185 (2007) (Special Issue on TOOLS Europe 2007)
4. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science* 127(3), 113–128 (2005)
5. Niu, N., Easterbrook, S., Sabetzadeh, M.: A Category-theoretic Approach to Syntactic Software Merging. In: *ICSM 2005: 21st IEEE International Conference on Software Maintenance*, pp. 197–206. IEEE Computer Society, Los Alamitos (2005)
6. Rutle, A., Wolter, U., Lamo, Y.: Diagrammatic Software Specifications. In: *NWPT 2006: 18th Nordic Workshop on Programming Theory* (October 2006)
7. Rutle, A., Wolter, U., Lamo, Y.: A Diagrammatic Approach to Model Transformations. In: *EATIS 2008: Euro American Conference on Telematics and Information Systems* (2008)
8. Rutle, A., Wolter, U., Lamo, Y.: A Formal Approach to Modeling and Model Transformations in Software Engineering. Technical Report 48, Turku Centre for Computer Science, Finland (2008)

9. Wolter, U., Diskin, Z.: Generalized Sketches: Towards a Universal Logic for Diagrammatic Modeling in Software Engineering. In: ACCAT Workshop 2007, satellite event of ETAPS 2007: European Joint Conferences on Theory and Practice of Software (to appear)
10. Diskin, Z.: Model Transformation via Pull-backs: Algebra vs. Heuristics. Technical Report 521, School of Computing, Queen's University, Kingston, Canada (September 2006)
11. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems* 16(4), 349–361 (2007) (Special Issue on Model-Driven Systems Development)
12. Rivera, J.E., Vallecillo, A.: Representing and Operating with Model Differences. In: 46th International Conference on TOOLS Europe 2008: Objects, Components, Models and Patterns. LNBP, vol. 11, pp. 141–160. Springer, Heidelberg (2008)
13. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. In: ESEC/FSE 2003: 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003, pp. 227–236. ACM, New York (2003)
14. Alanen, M., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
15. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
16. Cicchetti, A., Rossini, A.: Weaving Models in Conflict Detection Specifications. In: SAC 2007: 22nd ACM Symposium on Applied Computing, pp. 1035–1036. ACM, New York (2007)
17. Rutle, A., Rossini, A.: A Tentative Analysis of the Factors Affecting the Industrial Adoption of MDE. In: ChaMDE 2008: 1st International Workshop on Challenges in Model-Driven Software Engineering, pp. 57–61 (2008), http://sse1.vub.ac.be/ChaMDE08/_media/chamde2008_proceedings.pdf

Controller Synthesis from LSC Requirements^{*}

Hillel Kugler¹, Cory Plock¹, and Amir Pnueli²

¹ Computational Biology Group, Microsoft Research, Cambridge, UK
{hkugler,v-coploc}@microsoft.com

² Computer Science Department, New York University, New York, NY, USA
amir@cs.nyu.edu

Abstract. Live Sequence Charts (LSCs) is a visual requirements language for specifying reactive system behavior. When modeling and designing open reactive systems, it is often essential to have a guarantee that the requirements can be satisfied under all possible circumstances. We apply results in the area of controller synthesis to a subset of the LSC language to decide the realizability of LSC requirements. If realizable, we show how to generate system responses that are guaranteed to satisfy the requirements. We discuss one particular implementation of this result which is formulated as an extension of smart play-out, a method for direct execution of scenario-based requirements.

1 Introduction

Going directly from requirements to a correct implementation has long been a “holy grail” for system and software development. According to this vision, instead of implementing a system and then working hard to apply testing and verification methods to prove system correctness, a system is rather built correctly by construction. Synthesis is particularly challenging for reactive systems, in which the synthesized system must satisfy the requirements for any possible behavior of an external environment [2,25].

One formal specification language for reactive systems is Live Sequence Charts (LSCs) [4]. LSCs is a visual language, extending the classical message sequence charts with the ability to specify both safety and liveness properties. A methodology called the play-in/play-out approach was described in [11] as part of a tool called the Play-Engine. Play-in provides an intuitive means of capturing requirements by interacting with a graphical representation of the system, while play-out executes the scenarios in a way that gives a feeling of running an implementation of the system.

An improvement to play-out called *smart play-out* is introduced in [9]. This approach uses verification methods—in particular, model-checking—to run LSC specifications and avoid certain violations that may occur in the original version of play-out. Unfortunately, smart play-out cannot avoid all possible violations [7]. This paper addresses an improvement to smart play-out which guarantees non-violation over all computations, provided that the requirements are realizable. To accomplish this, we reformulate the previous model checking problem instead as a synthesis problem.

^{*} This research was supported in part by NIH grant R24-GM066969 and a donation by Robert B. K. Dewar and Edmond Schonberg.

We view the problem as a two-player open game between the *system* and the *environment*. The system refers to the components of an executable program we wish to construct; the environment represents external entities which produce system inputs. The system attempts to *win* the game by satisfying the LSC requirements, whereas the environment's goal is to foil the system by steering the game into a violating state. The game is carried out using a special transition system called a *game structure* that encodes the logic of user-supplied LSC requirements.

Given a game structure, the work of [24,23] provides a means of deciding realizability, which amounts to determining if a reactive system is capable of avoiding violation over all inputs and across all runs. If so, a transition system called a *controller* is extracted. The controller encodes the so-called *winning strategy* as the original input transition system with non-winning transitions removed to avoid violating the safety properties, and guards (possibly) added to certain edges to ensure satisfaction of the liveness properties. By following the transitions of the resulting controller, satisfaction of the complete requirements is guaranteed.

In this paper, we describe how to construct a game structure that expresses the behavior of a subset of the LSC language. We apply the results of [24], with certain modifications that allow us to deal with some advanced LSC constructs in a natural way, to determine realizability and extract the controller, provided it exists. If so, we use the winning strategy to choose correct system responses to every environment input. Responses are guaranteed to exist, provided the requirements are realizable.

The paper is organized as follows. Related work is discussed in section 2. We discuss the contributions and shortcomings of smart play-out in greater detail in section 3 and motivate the need for this work in section 4. We provide definitions in section 5 and a description of our synthesis methodology in section 6. Our main result is discussed in section 7 and conclusion in section 8.

2 Related Work

In recent years there have been considerable research efforts on synthesizing executable systems from scenario-based requirements [15,16,20,17,31,30,28,21,12]. In many of these papers, the requirements are given using a variant of classical message sequence charts and the synthesized system is state-based. Although there are many common aspects to our work and these papers, the main distinguishing feature of LSCs is that they are more expressive than most of the classical MSC variants.

Synthesis from LSCs was first studied in [8], and is tackled there by defining consistency, showing that LSC requirements are consistent iff they are satisfiable by a state-based object system. A satisfying system can then be synthesized. This line of work was continued in [10], which includes an implementation of a sound but not complete algorithm for Statechart synthesis. A game theoretic approach to synthesis from LSCs involving a reduction to parity games is described in [3]. Synthesis from LSCs using a reduction to CSP appears in [27]. All the above papers were either theoretical and did not include an implementation, or the synthesis approach was sound but not complete, or the synthesis time complexity was not encouraging. An alternative strategy for

synthesis from LSCs is to use a translation from LSCs to temporal logic [18,5] or automata [14] and then apply existing synthesis algorithms, e.g., [25,29].

In [24] a controller synthesis implementation for generalized Büchi winning conditions in the language of TLV-BASIC [26] is presented. The work is later extended in [23] to include Reactive(1) designs, or generalized Streett winning conditions. Neither of the results are specific to LSC requirements, but we utilize a modified version of the former implementation for our present work. In recent work [19], a compositional synthesis approach for a core subset of LSCs containing only messages is presented. The main contribution of [19] is the compositional approach, whereas this paper focuses on the basic synthesis algorithm for a wider LSC subset.

3 Smart Play-Out

Smart play-out [9] is a method for direct execution of scenario-based requirements, which allows a user to interact with an executing reactive system whose behavior was specified using Live Sequence Charts [4]. The user first creates the LSC requirements using play-in by manipulating the user interface of the target application (e.g., by pressing buttons, rotating knobs, etc.)

Once the requirements have been specified, smart play-out allows the user to play the role of the environment by injecting input events and then observe system responses that follow according to the requirements. More specifically, smart play-out accepts input events only when the Play-Engine is in a so-called *stable state*. Whenever an input (environment) event is injected within a stable state, smart play-out formulates a response—a sequence of outputs events called a *superstep*—which leads the computation to another stable state, provided a superstep exists. The main contribution of smart play-out is the means through which supersteps are identified and executed.

Smart play-out finds supersteps by first encoding the logic of LSCs into a transition system and then formulating a model checking problem for the specified environment input. Roughly speaking, smart play-out tries to verify the property “no superstep leading to a stable state exists” with the hope that the property is false. If it is indeed false, the model checker produces a counterexample as a witness to the existence of the superstep. Smart play-out then feeds the counterexample to the Play-Engine so that the user may witness the superstep being carried out graphically.

One limitation of smart play-out is that the model checking procedure explores the state space only to the extent necessary to identify a superstep leading to some successor stable state. The procedure disregards whether any supersteps exist from the successor stable state, or any stable state thereafter. Therefore, smart play-out may blindly lead the system into a state from which no superstep exists—a violation of the requirements, since reactive systems must supply a (correct) response to every environment input.

When the user sees the violation, they may arrive at an inaccurate conclusion that something is wrong with the requirements, when in fact the violation was due to the Play-Engine’s poor choice of supersteps. Better selection of supersteps could have yielded non-violation instead. The main problem is that the supersteps (i.e., counterexamples) seem to be chosen arbitrarily by the model checker. By choosing supersteps more wisely, it is possible to identify a priori whether supersteps exist for *all* possible

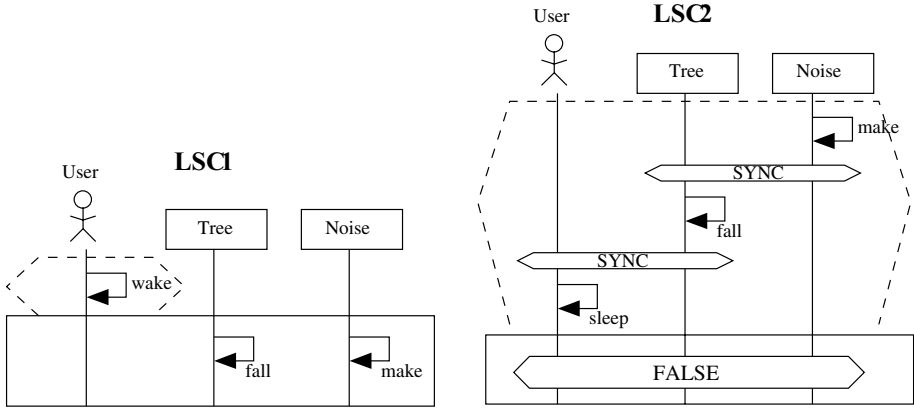


Fig. 1. LSC Requirements

sequences of inputs. To achieve this, we use synthesis techniques to perform a complete analysis of the state space. This allows for forward-looking decisions and complete avoidance of violations, provided the requirements are realizable.

4 Example

To solidify the above discussion with an example, consider LSC requirements consisting of the two LSCs shown in Fig. 1. Both scenarios include USER as an environment instance, and both TREE, NOISE as system instances. Accordingly, the behaviors of TREE and NOISE are within the control of the system we intend to construct, whereas the behaviors of the USER are assumed to be external.

According to LSC1, whenever USER sends the *wake* message, the controller must respond with a non-deterministic ordering of message *fall* and *make* in order to satisfy the main chart. Therefore, the traces *wake, fall, make* or *wake, make, fall* are both acceptable for satisfying LSC1. On the other hand, LSC2 is an *anti-scenario* that specifies the sequence *make, fall, sleep* cannot ever occur. The synchronizing conditions remove the otherwise non-deterministic ordering of *make, fall, sleep* to ensure that only traces with this precise ordering will satisfy the prechart.

We now consider how smart play-out might respond to an input message *wake* executed by the USER. After formulating a model-checking problem that checks for the non-existence of a satisfying trace, the resulting counter-example yields one of the two possible event sequences above. Supposing that smart play-out executes the sequence: *make, fall*, LSC1 would be satisfied, but the prechart of LSC2 would advance such that the next enabled message is *sleep*. The USER could then execute *sleep* and violate the requirements. This illustrates the inability of smart play-out to look ahead into the future by more than one superstep.

Using synthesis, it is possible for the controller to decisively choose an alternative sequence that would not allow the environment to violate the requirements. In response

to the message *wake*, the synthesis algorithm would have removed the transitions that permit the sequence *make*, *fall* to occur, leaving *fall*, *make* as the only existing path.

5 Game Structures

A *game structure* (GS) is defined by $G : \langle V, X, Y, \Theta, \rho_e, \rho_s, \varphi \rangle$ consisting of:

- V , a finite set of typed *state variables*. We define s to be an interpretation of V , assigning to each variable $v \in V$ a value $s[v] \in D_v$ within its respective domain. We denote by Σ the set of all states. We extend the evaluation function $s[\cdot]$ to expressions over V in the usual way. An *assertion* is a Boolean formula over V . A state s *satisfies* an assertion φ , denoted $s \models \varphi$, if $s[\varphi] = \text{T}$. We say that s is a φ -state if $s \models \varphi$.
- $X \subseteq V$ is a set of *input variables* controlled by the environment. Let $\bar{X}$ denote the set of all input variable valuations.
- $Y = V \setminus X$ is a set of *output variables* controlled by the system. Let $\bar{Y}$ denote the set of all output variable valuations.
- Θ is the initial condition characterizing all initial states of G .
- $\rho_e(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$ is the transition relation of the environment. This is an assertion relating state $s \in \Sigma$ to a possible input value $\bar{x}' \in \bar{X}$ by referring to unprimed and primed copies of $\bar{X}$ and $\bar{Y}$. The transition relation ρ_e identifies valuation $\bar{x}'$ as a possible *input* in state s , if for some output $\bar{y}'$, $(s, \bar{x}', \bar{y}') \models \rho_e(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$, where $(s, \bar{x}', \bar{y}')$ denotes a transition from state s to state $(\bar{x}', \bar{y}')$.
- $\rho_s(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$ is the transition relation of the system. This is an assertion relating state $s \in \Sigma$ to a possible output value $\bar{y}' \in \bar{Y}$ by referring to unprimed and primed copies of $\bar{X}$ and $\bar{Y}$. The transition relation ρ_s identifies valuation $\bar{y}'$ as a possible *output* in state s , if for some input $\bar{x}'$, $(s, \bar{x}', \bar{y}') \models \rho_s(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$.
- φ is the winning condition, given by an LTL (linear temporal logic) formula.

As can be seen, changes in state are characterized by changes in the variable valuations. We partition *variables* into those controlled by the environment (input variables) and those by the system (output variables). Each player may then observe and modify the valuations of its own variables, but can only observe the valuations of the opponent's.

5.1 Dependent vs. Independent Moves

We say that a player *moves* from a state whenever it modifies the variable valuations according to its transition relation. In most game settings, including [24], players strictly alternate between moving: a predesignated player moves first according to the current valuation of the input and output variables. The second player then observes the same valuations as the first and also the first player's move and then moves herself.

A different approach is presented in [6], whereby both players move simultaneously and independently. That is, they move at the same instant and both moves are a function of the current variable valuations only—a player can't observe the opponent's move. According to our approach, the players move simultaneously as before, but both players

are permitted to move dependently or independently. A player's move is *dependent* if it is a function of the current variable valuations *and* the opponent's move. If it is a function of only the current variable valuations, then it is an *independent* move, as above.

We adopt this approach because LSCs inherently require the system and environment to synchronize during certain points during the execution. This happens, for example, when system and environment instances arrive on an LSC condition. Our definition lends itself to modeling this type of behavior quite naturally. Although it is possible to simulate this type of synchronous behavior using the alternating-turn approach, extra memory and logic seems to be required.

To illustrate the above concepts, consider the following SMV [22] code:

```

1  next(env) := case
2    sys=0 & env=0 & next(sys)= 1 : 2;
3    sys=0 & env=0 : 3;
4    1 : env;
5  esac;
6
7  next(sys) := case
8    sys=0 & env=0 : {1,2};
9    1 : sys;
10  esac;
```

Listing 1.1. Example SMV Code

The example depicts the transition relation for environment input variable *env* between lines 1-5, and system output variable *sys* on lines 7-10. Elsewhere, variable *sys* is defined to range over $0, \dots, 2$ and *env* over $0, \dots, 3$.

The transition relation for each variable is expressed by a case statement. Each line of the case statement takes the form *expr* : *val* where *expr* is an expression over the variables and *val* is a legal next value (or set of values) when *expr* is true. Each line is evaluated in the order appearing in the input. If *expr* does not hold, then the next line is evaluated and so on, until one of the expressions holds. Expression “1” is a catch-all expression referring to all cases not covered by the expressions appearing above it.

For example, according to line 8, if *sys* and *env* are both 0 in the current state, then *sys* can nondeterministically choose between 1 or 2 in the next state. Line 9 states that the value of *sys* remains unchanged for all other cases. Lines 8 and 9 are examples of independent moves, since neither relies on the environment's move (the value of *env* in the next state.)

As for the transition relation of *env*, line 2 states that if *sys* and *env* are 0 and, furthermore, *sys* is 1 in the next state, then *env* is 2 in the next state. Line 2 is an example of a dependent move, since the case only holds with the cooperation of the system. Line 3 is also dependent since it implies that *next*(*sys*) is not equal to 1. However, line 4 is independent because it does not depend on any particular value of *next*(*env*).

5.2 Deadlock

Conceptually, each round of play proceeds as follows: from a given state, both players each choose among any available move which is legal according to their own transition relation. If both moves are independent then neither player risks interference from their opponent. If a player chooses a dependent move, then the set of allowable moves becomes restricted according to the opponent's move. Both players may also choose dependent moves. However, when at least one of the moves is dependent, there exists a possibility that moves which were legal according to each player's own transition relation may no longer be legal once combined. Such moves are said to be *deadlocked*.

To illustrate deadlock in this context, consider the following contrived example:

```

1  next(sys) := case
2    sys=0 & env=0 & next(env) = 1 : 1;
3    1 : sys;
4  esac;
5
6  next(env) := case
7    sys=0 & env=0 & next(sys)= 1 : 0;
8    1 : env;
9  esac;
```

Listing 1.2. Deadlock Example

First note that lines 2 and 7 both refer to dependent transitions, since the transition relation for each player's variable depends on the opponent's move. Now consider the state where $\text{sys}=\text{env}=0$ holds. According to line 2, if env is 1 in the next state, then sys must be 1 in the next state. However, according to line 7, if sys is 1 in the next state, then env must be 0 in the next state. The system's move on line 2 and the environment's move on line 7 are deadlocked because there will never be a way to proceed using this combination. Note that there could exist other moves that do work. For instance, both players may move from $\text{sys}=\text{env}=0$ to state $\text{sys}=\text{env}=0$.

Although not shown in this example, there could generally exist states from which all moves are deadlocked, leaving no possible next move. We refer to these states as *fully deadlocked*.

The presence of deadlocks, or even fully deadlocked states, in a transition system is not necessarily forbidden. For example, one may intentionally introduce deadlocks into a transition system to model some kind of real life dead-end situation, with the idea of having synthesis generate the strategy to avoid the deadlocks. In contrast with previous synthesis work based on the turn-based approach, such as [24], additional consideration is required for handling (fully) deadlocked states in the case of games with simultaneous transitions.

6 Synthesis

Let $\mathcal{G}$ be a game structure and s and s' be states of $\mathcal{G}$. We say s' is a *successor* of s if $(s, s') \models \rho_e \wedge \rho_s$. We freely switch between $(s, \vec{x}', \vec{y}') \models \rho_e$ and $\rho_e(s, \vec{x}', \vec{y}') = 1$ and similarly for ρ_s .

A play σ of $\mathcal{G}$ is a maximal sequence of states $\sigma : s_0, s_1, \dots$ satisfying *initiality* ($s_0 \models \Theta$) and *consecution* (for each $i \geq 0$, s_{i+1} is a successor of s_i). Let σ be a play of $\mathcal{G}$. From state s , the environment chooses an input $\bar{x}' \in X$ and system chooses an output $\bar{y}'$ such that $\rho_e(s, \bar{x}', \bar{y}') = 1$ and $\rho_s(s, \bar{x}', \bar{y}') = 1$.

We say that play σ is *winning for the system* if it is infinite and satisfies the winning condition φ . Otherwise, σ is *winning for the environment*.

Let $\sigma = s_0, \dots, s_n$. A *strategy* for the system is a function $f : \Sigma^+ \times \bar{X} \mapsto \bar{Y}$ where for every $\bar{x}' \in \bar{X}$ such that $\rho_e(s_n, \bar{x}', f(\sigma, \bar{x}')) = 1$, we have $\rho_s(s_n, \bar{x}', f(\sigma, \bar{x}')) = 1$. A play $s_0, s_1, \dots$ is said to be *compliant* with strategy f if for all $i \geq 0$ we have $f(s_0, \dots, s_i, s_{i+1}[\bar{X}]) = s_{i+1}[\bar{Y}]$, where $s_{i+1}[\bar{X}]$ and $s_{i+1}[\bar{Y}]$ are the restrictions of s_{i+1} to variable sets X and Y , respectively.

Strategy f is *winning* for the system from state $s \in \Sigma$ if all s -plays (plays departing from s) which are compliant with f are winning for the system. We denote by W_e the set of states from which there is a winning strategy for the system. $\mathcal{G}$ is said to be *winning* for the system if all initial states of $\mathcal{G}$ are winning for the system. In this case, we say $\mathcal{G}$ is *realizable* and we *synthesize* a winning strategy which is a working implementation for the system. Otherwise $\mathcal{G}$ is *unrealizable*.

6.1 Controllable Predecessors

States from which the system can force the game into p are referred to as *controllable predecessors* of p , denoted $\odot p$, where p is an assertion over the state space $(\bar{X}, \bar{Y})$. The main idea is that the system, from a controllable predecessor of p , can choose a move for which all remaining environment moves lead to p —or—for each possible environment move, can choose at least one move leading to p . That is, the system can take either an independent or dependent move. Our controllable predecessor formula is a disjunction of two parts, Φ_1 and Φ_2 . We have:

$$\Phi_1 = \exists \bar{y}' [\exists \bar{x}' \rho \wedge [\forall \bar{x}' \rho_e \rightarrow [\rho_s \wedge (\bar{x}', \bar{y}') \in \|p\|]]]$$

where $\|p\|$ denotes the set of states characterized by assertion p and $\rho = \rho_e \wedge \rho_s$ is the set of joint moves. Formula Φ_1 states that for some system move $\bar{y}'$, any legal environment move $\bar{x}'$ must lead to p . The left side of the conjunction assures the absence of fully deadlocked predecessors. Next we have:

$$\Phi_2 = [\exists \bar{x}' \exists \bar{y}' \rho] \wedge \forall \bar{x}' [[\forall \bar{y}' \neg \rho_e] \vee [\exists \bar{y}' \rho \wedge (\bar{x}', \bar{y}') \in \|p\|]]$$

The right side requires that for every environment input $\bar{x}'$, either there are no environment moves available or there must exist some system move $\bar{y}'$ leading to p . The left side of the conjunction assures the absence of fully deadlocked predecessors.

Putting it all together, we compute the set of controllable predecessors of p with:

$$\|\odot p\| = \{s \mid \Phi_1 \vee \Phi_2\}$$

6.2 Realizability and Winning Strategy

Once the notion of controllable predecessor is in place, the decision procedure for realizability and the extraction of the winning strategy proceeds according to [24], which

focuses on winning conditions which are recurrence properties, i.e., LTL formulas of the form $\Box \Diamond q$ for an assertion q . We restrict our attention to formulas of this form for the purposes of this paper.

A state satisfies $\bigodot p$ (for some assertion p) if the system can force the environment to reach a p -state in a single step. Based on this pre-image operator, a set of *winning states* is computed according to the following fix-point equation:

$$W_c = \nu Z \mu Y. \bigodot Y \vee q \wedge \bigodot Z \quad (1)$$

Given a game structure $\mathcal{G}$, we can check realizability of G by testing $\overline{W_c} \cap \Theta = \emptyset$. If $\mathcal{G}$ is winning for the system, a winning strategy is extracted by removing controllable transitions which lead to states outside of W_c .

7 Main Result

We now present a method for constructing a game structure from LSC requirements. Some of the LSC logic necessary for this result is already incorporated into smart play-out: whenever the user injects an input event, smart play-out constructs an LSC transition system. We avoid redundancy here by focusing most of our attention on the extensions necessary for synthesis. The interested reader can consult [9] for the specifics of the smart play-out construction.

On a high level, smart play-out defines one SMV module for every object in the requirements and composes the modules asynchronously for program executions and model-checking. In contrast, the synthesis algorithm of [24] requires precisely two transition systems—one for the system and one for the environment. One of our goals is therefore to express the collection of asynchronous transition systems as a game structure. Secondly, we add additional logic over and above that supplied by smart play-out which is necessary for synthesis. We begin by first introducing the variables used in our construction and then describe the transition relation for each.

7.1 Variables

Let $\mathcal{O}$ be an object system and let $\mathcal{LR} = L_1, \dots, L_n$ over $\mathcal{O}$ be a set of LSC requirements. We construct a game structure $\mathcal{G}$ with a set of *input variables* belonging to the environment and *output variables* belonging to the system. We now specify the set of variables V by defining the input variables X and output variables Y . The input variables are as follows:

1. act_{L_i} is 1 when the main chart of LSC L_i is active, and 0 otherwise.
2. $msg_{O_j \rightarrow O_k}^s$ denotes the sending of a message from object O_j to object O_k in which $O_j.own = env$ (O_j belongs to the environment.) The value is set to 1 at the occurrence of the send and is changed to 0 at the next state.
3. $msg_{O_j \rightarrow O_k}^r$ denotes the receipt of a message by object O_k from object O_j in which $O_k.own = env$. As in the case of sending, the value is 1 at the instant the message is received and changes to 0 in the next state.

4. l_{L_i, O_j} is the location of object O_j in the main chart of LSC L_i where $O_j.own = env$. The location number ranges over $0, \dots, l^{max}$ where l^{max} is the last location of O_j in the main chart of LSC L_i . This variable is meaningful only when act_{L_i} is 1.
5. $l_{pch(L_i), O_j}$ is the location of object O_j in the prechart of LSC L_i where $O_j.own = env$. Its value ranges over $0, \dots, l^{max}$ where l^{max} is the last location of O_j in the prechart of LSC L_i . This variable is meaningful only when act_{L_i} is 0.
6. $gbuchi$ is an auxiliary variable used to reduce a Generalized Büchi winning condition to a Büchi winning condition.
7. $envreq$ is a variable that determines which of the environment's objects has control in the next step.

The output variables belonging to the system are given by:

1. $msg_{O_j \rightarrow O_k}^s$ denoting the sending of a message from object O_j to object O_k in which $O_j.own = sys$ (O_j belongs to the system.)
2. $msg_{O_j \rightarrow O_k}^r$ denoting the receipt of a message by object O_k from object O_j in which $O_k.own = sys$.
3. l_{L_i, O_j} is the location of object O_j in the main chart of LSC L_i such that $O_j.own = sys$.
4. $l_{pch(L_i), O_j}$ is the location of object O_j in the prechart of LSC L_i such that $O_j.own = sys$.
5. $currobj$ is a number ranging over $1, \dots, |\mathcal{O}|$, referring to the object $O_{currobj}$ that currently has control of the execution.

The active flags, (act_{L_i} , for all i) and the auxiliary variable $gbuchi$ are not properties of the environment specifically, although they are environment variables. These are examples of *bookkeeping variables*, whose values are a function of the variables of both players. The choice of ownership could therefore be arbitrary. However, we assign ownership of these variables to the environment in order to be conservatively safe.

For example, if there exists a subtle error in the transition relation of any of these variables, the environment would find a way to utilize the error to its advantage in order to win the game and deem the requirements unrealizable. This is positive because we are forced to deal with the error in such a case. We could have alternatively chosen the system as the owner instead, in which case an error in the definitions could lead to false realizability—a more dangerous situation, particularly in the case of safety critical systems.

The purpose of the remaining variables, $envreq$ and $currobj$ are explained below.

7.2 Transitions

Smart play-out constructs a transition system comprised of an asynchronous composition of SMV modules. Generally speaking, each module defines the behaviors of one object in the LSC requirements, consisting of a set of variables and a transition relation. When generating traces, the TLV-BASIC [26] model-checking routine arbitrarily selects modules for execution one at a time. The corresponding variables are then updated according to the transition relation of the selected module. Intuitively, each module's (i.e., object's) transition relation permits the object to carry out the next behavior on the object's instance line, with respect to the object's present LSC location.

On the other hand, the current synthesis implementation requires a game structure in which all objects (and associated transition relations) belonging to the system are grouped into a single system module, and likewise for the environment. This raises the question of how to deal with the multiple definitions for each variable. We now describe the solution.

Let φ_i be any variable belonging to object O_i in the smart play-out construction. The transition relation, according to [9], for φ_i takes on the form:

$$\varphi'_i = \begin{cases} c_1^i & \text{if } \Omega_1^i \\ \vdots & \vdots \\ c_n^i & \text{if } \Omega_n^i \end{cases}$$

where c_j^i is a constant, Ω_j^i is a conditional expression over the variables of all objects in $\mathcal{O}$, and n is the number of SMV transition relation cases produced by smart play-out for φ_i . In our synthesis construction, we have:

$$\varphi' = \begin{cases} \varphi'_1 & \text{if } currobj = 1 \\ \vdots & \vdots \\ \varphi'_k & \text{if } currobj = k \end{cases}$$

where k is the number of objects. Therefore, we may simulate the asynchronous behavior of the smart play-out transition system by manipulating the variable $currobj$. This variable is responsible for determining which object, among the system and environment objects, move in the next step.

Variable $currobj$ must necessarily be owned by either the system or the environment. It would seem that permitting just one player to determine the current objects for both itself and its opponent could result in an unfair advantage. To level the playing field according to our result, the system and environment choose among their respective objects, but the decision of when each player gets their turn to decide goes to the system. To prevent the system from starving the environment of any opportunity to move, we will require the system to yield control to the environment infinitely often.

Formally, let $O_1, \dots, O_j$ be the set of objects belonging to the environment and $O_{j+1}, \dots, O_k$ be those of the system. We let:

$$envreq' \in \{1, \dots, j\}$$

The environment uses $envreq$ to non-deterministically choose which of its objects will be the next to move once given a turn. With this in place, the system selects the current object in the following way:

$$currobj' \in \{envreq', j + 1, \dots, k\}$$

Note that this permits the system to execute arbitrarily long supersteps, since it can just keep selecting values between $j + 1, \dots, k$. However, the winning condition discussed in the next section will require that $currobj' \leq j$ infinitely often, causing all supersteps to be finite.

7.3 Initial and Winning Conditions

The initial condition, Θ , of our game structure is the set of states in which $gbuchi = 0$, $act_{L_i} = 0$ for all i , all message variables are set to 0, and all location variables are set to 0. The initial value of $envreq$ is not specified in Θ , so the choice is therefore non-deterministic. The winning condition φ is the generalized Büchi LTL formula:

$$\Box \Diamond \bigwedge_{i=1}^n act_{L_i} = 0 \wedge \Box \Diamond currobj \leq j$$

which is equivalent to:

$$\Box \Diamond gbuchi = 0$$

where:

$$gbuchi' = \begin{cases} 1 & \text{if } gbuchi = 0 \\ 2 & \text{if } gbuchi = 1 \wedge \bigwedge_{i=1}^n act_{L_i} = 0 \\ 0 & \text{if } gbuchi = 2 \wedge currobj \leq j \end{cases}$$

The above winning condition ensures that a stable state—where all main charts are simultaneously inactive—is visited infinitely often and that all supersteps are finite. It assumes that no environment messages appear in a main chart. For this, a more expressive winning condition beyond the scope of this paper is necessary.

7.4 Synthesis in the Play-Engine

When a Play-Engine user creates LSC requirements and wishes to perform synthesis, the following steps occur: first, the LSC requirements are translated into a game structure according to the techniques of this section. Next, the synthesis algorithm described in subsection 6.2 is executed. If the algorithm yields an unrealizable outcome, the process terminates at this point and the user is notified. Otherwise, a single, synchronous, transition system is constructed. We refer the interested reader to [24] for more details on this construction, which we do not describe in this paper.

At this point, the Play-Engine user may act in the role of the environment by injecting environment inputs and observing system responses, in a manner nearly identical to smart play-out. Upon each input event, a model checking routine is executed on the above output transition system. Since the winning condition guarantees that all LSCs will infinitely often be simultaneously inactive for any realizable LSC requirements, it is therefore also guaranteed that a valid super-step will exist for every reachable stable state in the output transition system.

Note that while the model-checking procedure is executed each time the user injects an input, the synthesis need only run once. Moreover, LSC requirements and the synthesis algorithm need not exist on the same computer or platform as the application to be deployed, since the only deliverable is the output transition system.

8 Conclusion

In this paper, we introduced a method for overcoming the limitations of smart play-out by performing a complete analysis of the state space. We first described a modification to the previous turn-based approaches for synthesis which permits players to

transition simultaneously in a dependent or independent fashion. We then showed how to construct a game structure that expresses the behaviors of LSC requirements as a two-player game between the reactive system and its environment. After invoking the synthesis routine, the end result is a controller—a transition system—which consists only of transitions that collectively satisfy the LSC requirements, provided a satisfying system exists. The controller, which encodes the winning strategy, can be used for executing supersteps that satisfy the requirements.

We describe an implementation of the foregoing synthesis procedure as an extension to the Play-Engine's smart play-out feature. With this implementation, the user first plays in behavioral requirements, as before. Then the synthesis procedure may be invoked from the Play-Engine's user interface, which constructs the game structure, checks realizability, and extracts a controller if the requirements are realizable. The synthesis algorithm executes once, yielding a controller, from which supersteps may be extracted using a superstep extraction process similar to that already present in smart play-out.

We are currently implementing a new Scenario-Based Tool [1] with a special focus on scenario-based modeling of biological systems [13]. Consistency checking and synthesis are important capabilities required for biological modeling, thus we are implementing extensions and variants of the work described here. An experimental implementation of a new compositional synthesis algorithm was already implemented using this new tool [19]. Independently of any specific tool or application domain, however, we wish to place our current focus on a broader solution of synthesizing executable programs from scenario-based requirements, whereby the controller generated by the synthesis routine can be used to directly execute a general reactive system.

References

1. Microsoft Research Cambridge, Scenario-Based Tool for Biological Modeling (2009), <http://research.microsoft.com/SBT/>
2. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable concurrent program specifications. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
3. Bontemps, Y., Heymans, P., Schobbens, P.Y.: From live sequence charts to state machines and back: A guided tour. *IEEE Trans. Software Eng.* 31(12), 999–1014 (2005)
4. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. *Formal Methods in System Design* 19(1), 45–80 (2001); preliminary version appeared in: Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS 1999)
5. Damm, W., Toben, T., Westphal, B.: On the Expressive Power of Live Sequence Charts. In: Reps, T., Sagiv, M., Bauer, J. (eds.) *Wilhelm Festschrift*. LNCS, vol. 4444, pp. 225–246. Springer, Heidelberg (2007)
6. de Alfaro, L., Henzinger, T., Majumdar, R.: From verification to control: dynamic programs for omega-regular objectives. In: Proc. 16th IEEE Symp. Logic in Comp. Sci., pp. 279–290. IEEE Computer Society Press, Los Alamitos (2001)
7. Harel, D., Kantor, A., Maoz, S.: On the Power of Play-Out for Scenario-Based Programs. Technical report, Weizmann Institute (2009)
8. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. *Int. J. of Foundations of Computer Science (IJFCS)* 13(1), 5–51 (2002); also in: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, pp. 1–51. Springer, Heidelberg (2001)

9. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: Aagaard, M.D., O'Leary, J.W. (eds.) *FMCAD 2002*. LNCS, vol. 2517, pp. 378–398. Springer, Heidelberg (2002); also available as Tech. Report MCS02-08, The Weizmann Institute of Science
10. Harel, D., Kugler, H., Pnueli, A.: Synthesis Revisited: Generating Statechart Models from Scenarios-Based Requirements. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) *Formal Methods in Software and Systems Modeling*. LNCS, vol. 3393, pp. 309–324. Springer, Heidelberg (2005)
11. Harel, D., Marelly, R.: *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, Heidelberg (2003)
12. Hennicker, R., Knapp, A.: Activity-Driven Synthesis of State Machines. In: Dwyer, M.B., Lopes, A. (eds.) *FASE 2007*. LNCS, vol. 4422, pp. 87–101. Springer, Heidelberg (2007)
13. Kam, N., Kugler, H., Marelly, R., Appleby, L., Fisher, J., Pnueli, A., Harel, D., Stern, M., Hubbard, E.: A scenario-based approach to modeling development: A prototype model of *C. elegans* vulval fate specification. *Developmental Biology* 323(1), 1–5 (2008)
14. Klose, J., Wittke, H.: An automata based interpretation of live sequence chart. In: Margaria, T., Yi, W. (eds.) *TACAS 2001*. LNCS, vol. 2031, p. 512. Springer, Heidelberg (2001)
15. Koskimies, K., Makinen, E.: Automatic synthesis of state machines from trace diagrams. *Software — Practice and Experience* 24(7), 643–658 (1994)
16. Koskimies, K., Mannisto, T., Systa, T., Tuomi, J.: *SCED: A Tool for Dynamic Modeling of Object Systems*. Tech. Report A-1996-4, University of Tampere (July 1996)
17. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In: *Proc. Int. Workshop on Distributed and Parallel Embedded Systems (DIPES 1998)*, pp. 61–71. Kluwer Academic Publishers, Dordrecht (1999)
18. Kugler, H., 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)
19. Kugler, H., Segall, I.: Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. In: *Proc. 15th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*. LNCS. Springer, Heidelberg (2009)
20. Leue, S., Mehrmann, L., Rezaï, M.: Synthesizing ROOM models from message sequence chart specifications. Tech. Report 98-06, University of Waterloo (April 1998)
21. Liang, H., Dingel, J., Diskin, Z.: A comparative survey of scenario-based to state-based model synthesis approaches. In: *Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM 2006)*, pp. 5–12 (2006)
22. McMillan, K.: *Symbolic Model Checking*. Kluwer Academic Publishers, Boston (1993)
23. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) *VMCAI 2006*. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)
24. Pnueli, A.: *Extracting controllers for timed automata*. Technical report, New York University (2005)
25. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pp. 179–190 (1989)
26. Pnueli, A., Shahar, E.: A platform for combining deductive with algorithmic verification. In: Alur, R., Henzinger, T.A. (eds.) *CAV 1996*. LNCS, vol. 1102, pp. 184–195. Springer, Heidelberg (1996)
27. Sun, J., Dong, J.S.: Synthesis of distributed processes from scenario-based specifications. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) *FM 2005*. LNCS, vol. 3582, pp. 415–431. Springer, Heidelberg (2005)

28. Uchitel, S., Kramer, J., Magee, J.: Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Software Engin. Methods* 13(1), 37–85 (2004)
29. Vardi, M.: An automata-theoretic approach to fair realizability and synthesis. In: Wolper, P. (ed.) *CAV 1995. LNCS*, vol. 939, pp. 267–278. Springer, Heidelberg (1995)
30. Whittle, J., Saboo, J., Kwan, R.: From scenarios to code: an air traffic control case study. In: *25th International Conference on Software Engineering (ICSE 2003)*, pp. 490–495. IEEE Computer Society, Los Alamitos (2003)
31. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: *22nd International Conference on Software Engineering (ICSE 2000)*, pp. 314–323. ACM Press, New York (2000)

Interface Generation and Compositional Verification in JavaPathfinder

Dimitra Giannakopoulou and Corina S. Păsăreanu

Carnegie Mellon University/NASA Ames Research Center,
Moffett Field, CA 94035, USA,

{dimitra.giannakopoulou,corina.s.pasareanu}@nasa.gov

Abstract. We present a novel algorithm for interface generation of software components. Given a component, our algorithm uses learning techniques to compute a permissive interface representing legal usage of the component. Unlike our previous work, this algorithm does not require knowledge about the component's environment. Furthermore, in contrast to other related approaches, our algorithm computes permissive interfaces even in the presence of non-determinism in the component. Our algorithm is implemented in the JavaPathfinder model checking framework for UML statechart components. We have also added support for automated assume-guarantee style compositional verification in JavaPathfinder, using component interfaces. We report on the application of the approach to interface generation for flight-software components.

1 Introduction

Component interfaces are a central concept in component-based software engineering. Although in current practice, interfaces typically describe the services that a component provides and requires at a purely syntactic level, the need has been identified for interfaces that document richer aspects of component behavior. Such extended interfaces are usually not provided, which makes their automatic generation an area of active research [1,10,5].

This paper addresses the automatic generation of interfaces that describe legal sequences of component calls. Such interfaces can serve as a documentation aid to application programmers, but can also be used by verification tools in checking that components are invoked correctly within a system. In fact, component interfaces are key for modular program analysis. They reduce the task of verifying a system consisting of a component and a client, to the more tractable task of verifying that the client satisfies the component's interface.

In previous work [6,16], we presented a framework based on learning, to perform automated assume-guarantee model checking of safety properties. To check that a system consisting of components M_1 and M_2 satisfies a safety property P , the framework automatically builds and refines *assumptions* A for one of the components, for example M_1 , to satisfy P , which it then tries to discharge on the other component, M_2 . Although assumptions A essentially constitute interfaces for component M_1 , their generation relies on knowledge of component M_2 .

Moreover, the focus of the framework was to compute assumptions that would allow to prove or disprove the property in the system, rather than assumptions that precisely document the behavior of a component.

The algorithm presented here for interface generation is also based on learning. However, in contrast to our work discussed above, it concentrates on the creation of precise component interfaces, *irrespective* of the component clients. By precise, we mean *safe* and *permissive*, as defined in [10]. An interface is safe if it accepts no illegal sequence of calls to the component. An interface is permissive if it includes all the legal sequences of calls to the component. Moreover, in [8], we presented an algorithm for generating what we call *weakest* assumptions in the context of Labeled Transition Systems. Weakest assumptions essentially constitute precise component interfaces. The difference of the current algorithm is that it is iterative, meaning that it can return partial results. Moreover, the approach in [8] required an expensive determinization step that we avoid here by dealing with the non-determinism in the component *dynamically*, during component analysis, as guided by counter-examples. Furthermore, our past experience, as well as other independent work [5], has indicated that the learning-based approach is more efficient for components that have relatively small interfaces.

Henzinger et al. also target the generation of safe and permissive interfaces in [10]. Unlike our framework, their work based on abstraction techniques and it is only applicable to components that are *visibly deterministic*. The latter requires that the behavior of the component be deterministic with respect to the methods / actions in its communication interface (we will henceforth call the communication interface of a component its *alphabet* in order to avoid confusion with interface in this context). In the applications that we have been dealing with, this requirement proved too restrictive. For example, we often need to generate interfaces that focus on specific aspects of the component behavior, and that therefore include only a subset of the component's alphabet. Components that are visibly deterministic with respect to their full alphabet, typically lose this property when a subset of that alphabet is considered. Finally, Alur et al. [1] also use learning to synthesize interface specifications for abstracted Java components. However, their approach is heuristic-based, i.e., they do not always obtain precise interfaces.

We have implemented our algorithms in the JavaPathfinder (JPF) model checking framework for UML statechart components [11]. We have also added support for automated assume-guarantee style compositional verification in JPF, using component interfaces. JPF is an open source model checker for Java programs which, until now, provided no support for compositional verification.

The contributions of this work can be summarized as follows:

1. A novel algorithm for automated generation of precise component interfaces, also applicable to components that are not visibly deterministic
2. Implementation of our algorithm in the JPF open source model checker. In addition to interface generation, we have provided support for verification of safety properties expressed as finite state automata as well as assume-guarantee reasoning in JPF, where assumptions and guarantees are both

expressed as finite-state automata. The implementation is freely available as JPF's compositional verification (cv) extension.

3. Case studies in the context of NASA applications that demonstrate the use of our algorithm in practice.

Related Work. The work closest to ours was discussed above. Several other approaches to automatic generation of component interfaces have been proposed in the literature. For example, Whaley et al. [19] use a combination of static and dynamic analyses to generate interfaces for Java components. Tkachuk et. al [18] use static analysis to obtain component abstractions, used as environments during modular analysis. Some approaches are based on extracting interfaces from sample execution traces [3]. All these techniques generate approximate interfaces, as opposed to our work that aims at producing precise interfaces that provide correctness guarantees. Interface generation is related to compositional verification. In particular, assume-guarantee reasoning is a compositional approach that uses assumptions when reasoning about components in isolation [12,17,2,7] and component interfaces can be used as assumptions in this context. Finally, learning has also been used to generate models for formal verification [14,9]; those approaches do not address interface generation or compositional reasoning.

2 Background

We model software components using labeled finite state transition systems (LTSSs), where transitions are labeled with component actions.

Let $\mathcal{Act}$ be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment. Let π denote a special *error state*, which models safety violations in the associated transition system.

LTSs. An LTS M is a four-tuple $\langle Q, \alpha M, \delta, q_0 \rangle$ where: Q is a finite non-empty set of states; $\alpha M \subseteq \mathcal{Act}$ is a set of observable actions called the *alphabet* of M ; $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is a transition relation; and $q_0 \in Q$ is the initial state.

Let $M = \langle Q, \alpha M, \delta, q_0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q'_0 \rangle$. M *transits* into M' with action a , denoted $M \xrightarrow{a} M'$, if $(q_0, a, q'_0) \in \delta$, $Q = Q'$, $\alpha M = \alpha M'$, and $\delta = \delta'$.

An LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is *non-deterministic* if it contains τ -transitions or if there exists $(q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*.

Traces. A *trace* t of an LTS M is a finite sequence of observable actions that label the transitions that M can perform starting at its initial state, ignoring the τ -transitions. For $\Sigma \subseteq \mathcal{Act}$, we use $t \upharpoonright \Sigma$ to denote the trace obtained by removing from t all occurrences of actions $a \notin \Sigma$. For a set of traces T , $T \upharpoonright \Sigma = \{t \mid \exists t' \in T. t' \upharpoonright \Sigma = t\}$.

Parallel Composition. Parallel composition “ $\parallel$ ” is a commutative and associative operator: given LTSs $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$,

$M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$, where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows: (1) $M_1 \parallel M_2 \xrightarrow{a} M'_1 \parallel M_2$ if $M_1 \xrightarrow{a} M'_1$ and $a \notin \alpha M_2$, (2) $M_1 \parallel M_2 \xrightarrow{a} M_1 \parallel M'_2$ if $M_2 \xrightarrow{a} M'_2$ and $a \notin \alpha M_1$, (3) $M_1 \parallel M_2 \xrightarrow{a} M'_1 \parallel M'_2$ if $M_1 \xrightarrow{a} M'_1$, $M_2 \xrightarrow{a} M'_2$, and $a \neq \tau$.

3 Interface Generation

In this section we define safe and permissive interfaces for software components and we describe our approach to synthesizing such interfaces automatically.

3.1 Safe and Permissive Interfaces

Let M be a software component. For simplicity of presentation, we will first assume that M includes an error state that expresses the undesired behavior of M (for example, some assertion violations). Later in this section we will discuss the more general case where the component property is given as a separate (safety) automaton.

Let $\Sigma \subseteq \alpha M$ denote the communication alphabet of component M , i.e., the set of actions through which M communicates with its environment. Our goal is to compute M 's precise interface as a finite state automaton A over Σ . As mentioned, we need to make sure that A is both *safe* and *permissive*, as defined formally below.

Let us first define the legal and illegal languages of component M . A word $t \in \alpha M^*$ is *illegal* if it corresponds to *some* trace of M that leads to error state π ; otherwise, the word is *legal*. Then $\mathcal{L}_{\text{legal}}(M)$ denotes the set of legal words of M and $\mathcal{L}_{\text{illegal}}(M)$ denotes the set of illegal words of M . Note that $\mathcal{L}_{\text{legal}}(M)$ and $\mathcal{L}_{\text{illegal}}(M)$ are complementary. Furthermore, note that, while illegal words correspond to actual traces in the component, legal words may also represent behavior that is never executed by the component (and hence could never lead to violations).

Definition 1. A is a *safe interface* iff $\mathcal{L}_{\text{legal}}(A) \cap \mathcal{L}_{\text{illegal}}(M) \upharpoonright \Sigma = \emptyset$.

In other words, an interface is safe if it accepts no illegal words of M .

Definition 2. A is a *permissive interface* iff $\mathcal{L}_{\text{legal}}(M) \upharpoonright \Sigma \subseteq \mathcal{L}_{\text{legal}}(A)$.

In other words, an interface is permissive if it accepts all legal words of M .

3.2 Learning Interface Specifications with L^*

Our approach for learning interface specifications is illustrated in Figure 1. We use an off-the-shelf learning algorithm, L^* [4], to iteratively compute interface specification A for M that is both *safe* and *permissive*. L^* learns an unknown language (over a given alphabet) and produces a *minimal* deterministic finite state automaton that accepts it; the learning process is iterative and it uses a *teacher* that provides answers to queries and counterexamples to conjectures (for details on L^* see [4]). In our framework, the problem of answering queries and counterexamples is reduced to reachability problems, solved by a model checker.

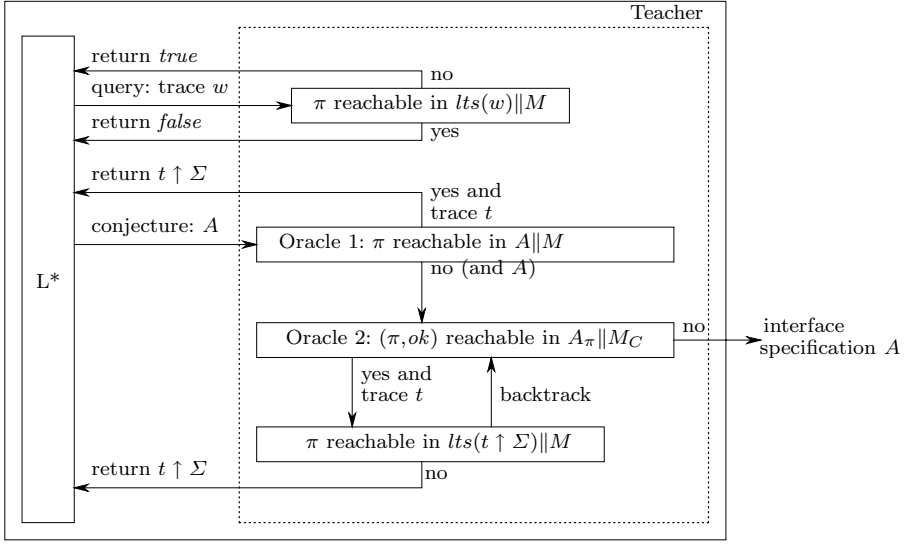


Fig. 1. Learning interface specifications with L^*

Queries L^* is first used to repeatedly *query* M to check whether or not, in the context of strings w , M violates the property. This is equivalent with checking if an error state π is reachable in $lts(w) \parallel M$. Here $lts(w)$ denotes an LTS over Σ that accepts string w (and its prefixes). The results of the queries are used by L^* to first make a “conjecture”, i.e. it builds an automaton A that accepts all the strings for the positive queries (the case error unreachable), and does not accept the strings for the negative queries (the case error reachable).

The conjectured automaton A is then checked to make sure it is both safe and permissive. This is done with the help of a *teacher* that implements two oracles as described below.

Oracle 1 checks if A is *safe* by checking whether π is reachable in $A \parallel M$. If it is, then it means that A is un-safe. The resulting counterexample t , projected on the interface alphabet Σ , is returned to L^* to refine its conjecture. If the error state is un-reachable, then it means A is safe and we proceed to Oracle 2.

Oracle 2 checks if safe interface A is also permissive, i.e. we want to check that $\mathcal{L}_{legal}(M) \uparrow \Sigma \subseteq \mathcal{L}_{legal}(A)$. This amounts to making sure that there are no words $w \in \Sigma^*$ such that $w \in \mathcal{L}_{legal}(M) \uparrow \Sigma \cap \mathcal{L}_{illegal}(A)$. This is equivalent to $w \in \mathcal{L}_{illegal}(A)$ and $\forall t \in \alpha M$ such that $w = t \uparrow \Sigma$, $t \in \mathcal{L}_{legal}(M)$.

We search for such words using a special reachability procedure performed on $A_\pi \parallel M_C$ (see pseudo-code in Figure 2). Here A_π denotes the *completion* of A with an error state, i.e. we complete each state with outgoing transitions to π , such that each state has outgoing transitions labeled with every action in Σ . Similarly, M_C denotes the *completion* of M with a special sink state. We

Oracle 2

input: safe interface A ;
begin
(1) Model-check $A_\pi \parallel M_C$:
(2) **if** (π, ok) is reachable by trace t **then**
(3) **if** π is not reachable in $lts(t \uparrow \Sigma) \parallel M$ **then**
(4) **return** $t \uparrow \Sigma$ to L^* ;
(5) **else**
(6) **backtrack**;
(7) **output:** safe and permissive interface A ;
end.

Fig. 2. Oracle 2

need these constructions to reason about traces in $\mathcal{L}_{illegal}(A)$ and $\mathcal{L}_{legal}(M)$, respectively. Note that $\mathcal{L}_{illegal}(A) = \mathcal{L}_{illegal}(A_\pi)$ and $\mathcal{L}_{legal}(M) = \mathcal{L}_{legal}(M_C)$. Note also that for Oracle 2, since both A_π and M_C contain error states, we need to distinguish between the two in $A_\pi \parallel M_C$ (this was not necessary for queries and Oracle 1).

Given the above constructions, checking permissiveness reduces to checking reachability of states of the form: (π, ok) , where π is an error state coming from A_π and ok denotes a non-error state in M_C . If such a combined state is found, then the trace t leading to it *may* indicate that A is not permissive, since $w = t \uparrow \Sigma$ leads to an error state in A_π but it is legal in M_C (and hence in M). However, due to non-determinism in M (and hence in M_C), it may be the case that on another path, t *does* lead to the error state. Even if this is not the case, there may exist other traces t' such that $w = t' \uparrow \Sigma$ and t' leads to an error in M_C on a different path (see Figure 3).

We check both these cases by performing a *query* on $t \uparrow \Sigma$. Note that *we do not stop the state space exploration*, but rather, we take trace t that is returned, and we check if, in the context of $t \uparrow \Sigma$, M violates its properties.

If the query returns true, then it means the interface is not permissive, and therefore $t \uparrow \Sigma$ is returned to L^* for refinement, and the learning process continues with more queries and eventually with a new conjecture.

If the query returns false, then t does not correspond to a real counterexample. Model checking therefore ignores this state; it backtracks, and then continues its state space exploration. If no traces that satisfy the condition above exist, then indeed the conjectured automaton is also the most permissive interface, and therefore it is output to the user.

We note that every query is stored in the L^* memoized table, so the result of the query on the same trace $t \uparrow \Sigma$ later (when A is the same) will be obtained directly (and faster) from the table.

Properties as safety automata. Assume now that M does not have error states, and we want to generate an interface specification for ensuring a property P , given as a (deterministic) safety automaton, encoding all the desired behaviors of the component. Conversely, P_π encodes all the un-desired behaviors of the

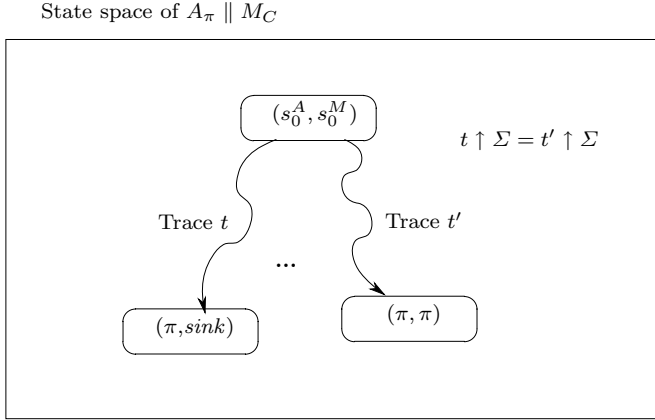


Fig. 3. Example for Oracle 2: dealing with non-determinism

component. The procedure described above will be exactly applicable to this case as well, if we treat $M \parallel P_\pi$ as M above.

3.3 Correctness and Termination

We argue here the correctness and termination of our approach. To argue correctness, we first show that Oracle 1 (and similarly the queries) guarantee a safe interface while Oracle 2 guarantees a permissive interface; therefore, the teacher implemented by our approach is correct.

Proposition 1. *Oracle 1 returns A iff $\mathcal{L}_{legal}(A) \cap \mathcal{L}_{illegal}(M) \uparrow \Sigma = \emptyset$.*

Proposition 2. *Oracle 2 returns A iff $\mathcal{L}_{legal}(M) \uparrow \Sigma \subseteq \mathcal{L}_{legal}(A)$.*

Due to lack of space we omit the proofs here; they proceed by contradiction and follow the arguments given informally in the previous section.

Theorem 1. *Given component finite state M (that may include error states), the algorithm implemented by our approach terminates and it returns a safe and permissive interface A .*

Proof. Correctness follows from the two propositions above. Termination follows from the correctness of L^* , which is guaranteed that, if it keeps receiving counterexamples, it will eventually terminate.

Discussion. As mentioned, in previous work we defined an algorithm for building safe and permissive interfaces for finite state components [8]. That algorithm involves the determinization of M (using the sub-set construction) that results in an exponential cost in computation time, regardless of the size of the interface specification. However, for components with small interfaces, the interface automaton is expected to be much smaller than the component itself. We address

this problem by using L^* , which builds incrementally automata with increasing size, finishing with the *minimal* deterministic automaton representing a safe and permissive interface.

We also note here that the approach of Henzinger et al. [10] can only handle components that are visibly deterministic, and therefore could not handle the case illustrated in Figure 3. On the other hand, the approach of Alur et al. [1] handles non-deterministic components, but it does not guarantee that the interface is permissive, since it only uses heuristics to implement what it amounts to Oracle 2 (called “superset query” in [1]). That work argues that the superset query can not be implemented efficiently, since it involves the determinization of component M . In our work we avoid an explicit determinization step of M . Instead, our approach deals with the non-determinism in the component dynamically (during model checking of the component) and only *selectively* (as guided by counterexamples).

4 Compositional Verification in JPF

4.1 Java PathFinder

Java PathFinder (JPF) [11] is an open-source verification framework developed by the RSE group at NASA Ames. It has been started as an explicit state model checker for Java byte-code. The focus of JPF is on finding bugs, such as concurrency related bugs (deadlocks, races, missed signals etc.), runtime related bugs (e.g. unhandled exceptions), etc. JPF can also check for violations of user-specified assertions that encode application specific requirements. JPF uses a variety of scalability enhancing mechanisms, such as user extensible state abstraction and matching, on-the-fly partial order reduction, configurable search strategies, and user definable heuristics (searches, choice generators).

4.2 JPF’s UML Statechart Extension

JPF has recently been extended with a statechart modeling and analysis capability that allows Java modeling of UML state machines [15]. Many UML development systems can produce code from diagrams, but this code is usually aimed at production systems, and is not suitable for software model checkers. The approach taken in JPF (Figure 4(left)) is based on a specific translation scheme from UML state charts into Java code that (a) is highly readable, (b) shows close correspondence between diagram and program, (c) provides a 1:1 mapping between model and program states, and (d) imposes few restrictions about aspects and actions that can be modeled.

The JPF Statechart extension is specialized to handle the obtained Java models more efficiently than random Java code. These Java models can be run in isolation, which corresponds to running them in the context of an external environment that may provide any input event at any stage (we will call this the universal environment). Alternatively, a guidance script may be provided by the

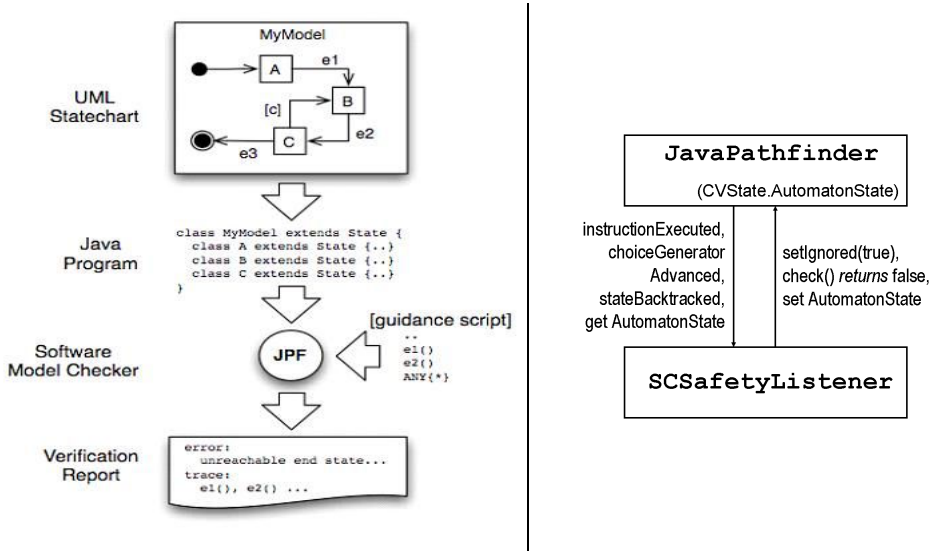


Fig. 4. Example illustrating JPF's UML extension (left) and JPF's listener (right)

user, which represents the input event sequences that can be provided by the external environment.

We have used the JPF statechart extension to implement our interface synthesis algorithms for components expressed in the JPF statechart framework. Note that we do not attempt to perform compositional reasoning for UML statecharts, which is a theoretical challenge that is beyond the scope of this work. Rather, we use UML statecharts, as supported by JPF, to represent finite state components with Labeled Transition System semantics. Therefore composition of components comes down to LTS composition, as described in Section 2. The interfaces that we generate are expressed as LTSs in the FSP notation [13].

4.3 Assume-Guarantee Reasoning in JPF

We have implemented assume-guarantee reasoning in JPF. As mentioned, components are given as UML statecharts (instances of class `CVState`). Both properties and assumptions are represented as finite state automata (instances of class `gov.nasa.jpf.cv.SCSafetAutomaton`).

Model checking using assumptions and properties has been implemented using JPF listeners (Figure 4(right)). A listener is essentially configured client code that is notified when certain events occur while JPF performs its search. The notified listener code can interact with JPF, e.g. a JPF “property” listener informs JPF if the property holds via the return value of its `check()` method.

Checking for both assumptions and properties is implemented with the `gov.nasa.jpf.cv.SCSafetyListener` class. On creation, a `SCSafetyListener` is associated with a finite state automaton P , which expresses the property or assumption to be used during model checking. Note that the state of a listener

is not included in the state that JPF explores / stores during model checking. However, the state of the automaton P needs to be part of the state space for correct state-space exploration and backtracking. We perform this by adding a static integer field of class `CVState` for the `cv` extension, which is set from within the listener.

An `SCSSafetyListener` listens for and reacts to the following events:

- **instructionExecuted**: Signals to the listener that an instruction was executed by JPF. The listener reacts by invoking method `advance(...)` on the automaton P . Advancing the automaton corresponds to making a state transition, if the instruction that was executed corresponds to an action in the alphabet of the automaton. If a transition on an alphabet action is undefined from the current state, this is an illegal transition (corresponds to a transition to the *error* state π). For properties, this means that an error has occurred, so the result returned by the listener's `check()` method is false.
- **choiceGeneratorAdvanced**: Signals that the next statechart action is selected for execution. The reaction of the listener is to check whether this action would make P transition to the error state if it were to be executed (this does not change the state of P since the transition is not really executed yet). Reaching an error state in an assumptions means that the current path explored is not a valid path under this assumption and must therefore be ignored. The listener forces JPF to backtrack (by executing `vm.getSystemState().setIgnored(true)`).
- **stateBacktracked**: When the model checker backtracks, then the automaton must backtrack accordingly.

For example, in order to check some property described as an automaton provided in some file `Foo`, we need to include the following arguments when running JPF's main class `gov.nasa.jpf.JPF`:

```
+jpf.listener=cv.SCSSafetyListener
+safetyListener1.property= Foo
```

The first argument informs JPF that an `SCSSafetyListener` will need to be notified of specific events, and the second one provides details for the listener, i.e., its unique id is "1", it is of type `property` (as opposed to `assumption`), and the automaton associated with it is provided in file `Foo` (this may also include the full path to `Foo`).

4.4 Interface Generation and Discharge

The interface generation in JPF is implemented in the main class `gov.nasa.jpf.tools.cv.ScRunCV`. The user can customize the generation via the following arguments:

```
+assumption.alphabet=<actions> defines the interface alphabet;
+assumption.outputFile=<file name> defines a file in which the generated
interface is output.
```

```

public boolean query(Vector sequence) throws SETException {

    Boolean recalled = memoized_.getResult(sequence);
    if (recalled != null) {
        return (!recalled.booleanValue());
    } else {
        // play the query as an assumption
        System.out.println("\n New query: " + sequence);
        SCSafetyListener assumption = new SCSafetyListener(
            new SCSafetyAutomaton
                (true, sequence, alphabet_, "Query", module1_));

        JPF jpf = createJPFInstance(assumption, property, module1_);
        jpf.run();
        boolean violating = jpf.foundErrors();
        memoized_.setResult(sequence, violating);
        return (!violating);
    }
}

```

Fig. 5. Answering queries in `SCModularTeacher`

This allows for a generated interface to be used for subsequent reasoning, either as an assumption, or as a property. The format currently used for expressing the interface is the FSP language.

The main method of `gov.nasa.jpf.tools.cv.ScRunCV` creates an instance of class `gov.nasa.jpf.tools.cv.SETLearner` to carry out the learning of the interface; an associated instance of `gov.nasa.jpf.tools.cv.SCModularTeacher` serves as the teacher. Our learning algorithm implementation uses JPF to perform the model checking steps described in Section 3. JPF model checks individual components in the context of the universal environment. Listeners are added as necessary to reflect the work of the Teacher, which consists of answering Queries, and implementing Oracle 1 and Oracle 2 in order to answer conjectures, as described in more detail below.

Queries and Oracle 1. Queries and Oracle 1 are performed in a similar fashion because they are concerned with checking whether error states are reachable in the component, in the context of a particular sequence (for queries) or finite state automaton (for Oracle1). As illustrated in Figure 5, to respond to a query, a listener instance `assumption` is created with an associated automaton that reflects the particular sequence that is being queried. JPF is then invoked, together with the `assumption` listener. If JPF returns errors, the answer to the query is `false`, otherwise the answer is `true`. Oracle 1 works in a similar fashion, with the difference that it also returns a counterexample.

Oracle 2. Oracle 2 checks for permissiveness of a computed interface. It needs to work on the completed component, as described in Section 3. This is a manual step that we intend to automate in the future. It similarly invokes JPF,

but performs the search in the context of a specialized type of listener, the `gov.nasa.jpf.cv.SCConformanceListener`. Its aim is to detect the reachability of a (π, ok) combination of states in the interface and component where the interface is in an error state, while the component is in a non-error state.

The `gov.nasa.jpf.cv.SCConformanceListener` listens for and reacts to the following events:

- **executeInstruction**: When the instruction about to be executed by JPF is an assertion violation, then it means that the component has entered an error state. Since such states are not targeted by the listener, it performs `ti.skipInstruction();`
`vm.getState().setIgnored(true);`
The first command ensures that the exception is not processed by JPF, for efficiency. The second asks JPF to backtrack since this path cannot lead to the targeted combination of states.
- **instructionExecuted**: Similar to `gov.nasa.jpf.cv.SCSafetyListener`. When the automaton associated with the listener moves to an error state, the result returned by the `check()` method of the listener is set to false, since the component is in a legal state (illegal states are never reached since the listener advises JPF to backtrack when it reacts to `executeInstruction` events), while the interface is in an error state.
- **stateBacktracked**: Similar to `gov.nasa.jpf.cv.SCSafetyListener`.

As described in Section 3, when an (π, ok) state is detected by the model checker, the counterexample leading to this state is queried, and if it is not a real counterexample, the model checker will backtrack. Since a query involved calling the model checker, this would involve nested model checker calls. To avoid such nesting, our implementation exploits a memoized table that is used by the learner to store results of previous queries. Oracle 2 checks for the reachability of (π, ok) states in a *loop*. Whenever a counterexample is obtained by the model checker, then Oracle2 invokes a query on it. Each query stores its result in the memoized table.

Whenever a real counterexample is obtained, Oracle 2 exits the loop and reports the result to the learner. When a counterexample is spurious, then another iteration of the loop is entered. In this iteration, we wish to ensure that the model checker will not report the same spurious counterexample. We achieve this as follows. When a `gov.nasa.jpf.cv.SCSafetyAutomaton` is asked to advance in the context of a `gov.nasa.jpf.cv.SCConformanceListener`, if the automaton reaches an error state, it will get the path to this state from JPF. It will then check the memoized table to see if there is a result for the corresponding sequence stored there. If there is, and the result is true, then it means that this is a spurious counterexample, and it notifies JPF to backtrack. Therefore, we have implemented the nested model checking calls by consecutive calls to the model checker, where the information of spurious counterexamples is shared through the memoized table.

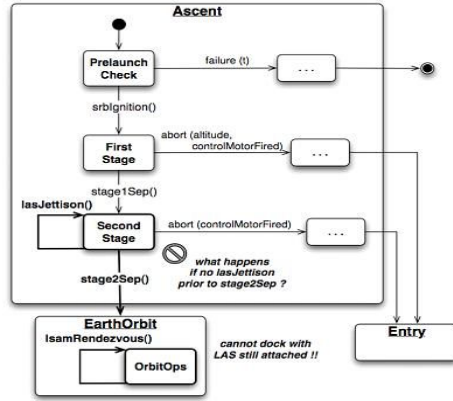


Fig. 6. Model of the Ascent and Earth Orbit flight phases of a spacecraft

Interface discharge. For compositional reasoning, one needs to also discharge the generated interface on the component environment. This can be performed by model checking the environment component in the presence of a `gov.nasa.jpf.cv.SCSafetyListener` using the interface as a property.

5 Experience

In order to evaluate our implementation, we used a statechart model of the *Ascent* and *EarthOrbit* flight phases of a space-craft (see Figure 6). The JAVA model is available with the JPF distribution under `examples/jpfESAS`. The UML statechart diagrams for the model are included in `examples/jpfESAS.doc`.

The model was created and used to demonstrate the features of the JPF UML statechart extension to our NASA mission customers. Several properties were analyzed on the model, and JPF returned violations for some of these properties. When the counterexamples obtained were analyzed, it was clear that some of the violations were spurious. The violations were related to the following properties:

- An event *lsamRendezvous*, which represents a docking maneuver with another spacecraft, fails if the LAS (launch abort system) is still attached to the spacecraft.
- Event *tliBurn* (trans-lunar interface burn takes spacecraft out of the earth orbit and gets it into transition to the moon) can only be invoked if EDS (Earth Departure Stage) rocket is available.

These violations were due to the fact that the universal environment was too general. The models had been created under the assumption that the use of the model respects some implicit flight rules. We decided to use our interface generation techniques to formalize the flight rules. More specifically, for each property,

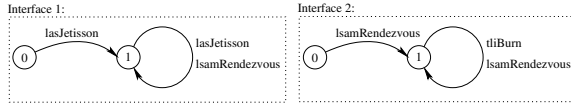


Fig. 7. Generated interface specifications

we generated a safe and permissive interface to eliminate its corresponding violations. To do this, we added a listener that eliminated all assertion violations that were not related to the targeted property, through the following arguments:

```
+jpf.listener=.tools.ChoiceTracker:.cv.AssertionFilteringListener
+assertionFilter.include=<method_name>
```

These arguments specify that all assertion violations that occur outside the particular `<method_name>` will be ignored.

The generated interface specifications are illustrated in Figure 7. The first one expresses the fact that the *lsamRendezvous* maneuvers cannot start before the *las* module of the spacecraft has jettisoned. According to the second one, it does not make sense to perform *tliBurn* prior to performing *lsamRendezvous*. These interfaces were inspected by the developer of the model that confirmed that they encode actual flight rules. Interface generation can therefore be used by developers to help them in the expression of the assumptions that their models encode. We note that other examples, including the input-output example from [6], are available with the JPF distribution.

6 Conclusions

We have proposed an algorithm for automatically synthesizing behavioral interface specifications for finite state software components. Our algorithm is the first iterative approach that is guaranteed to compute interfaces that are both safe and permissive, even in the presence of non-determinism in the visible behavior of a component. We have implemented our approach in the JavaPathfinder model checking framework for UML statechart components, and have obtained promising results from its application to several systems. The source code for the implementation and the examples is available through JPF's distribution.

In the future, we plan to investigate interface generation for methods with parameters. We have made some initial experiments using JPF's symbolic execution extension to generate values for parameters with infinite domains, and used these values to define finite interface alphabets related to their corresponding methods. We wish to pursue this direction further, and also plan to extend our results to generic Java components. For components that may be infinite-state, we will combine our approach with techniques such as predicate abstraction (similar to [1]). Finally, we plan to perform extensive evaluations of our approach.

Acknowledgements

We thank Peter Mehrlitz and Suzette Person for helping with the implementation.

References

1. Alur, R., Cerny, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: *Proceedings of POPL 2005*, pp. 98–109 (2005)
2. Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., Tasiran, S.: MOCHA: Modularity in Model Checking. In: Vardi, M.Y. (ed.) *CAV 1998*. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
3. Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. In: *Proceedings of ACM POPL 2002*, pp. 4–16 (2002)
4. Angluin, D.: Learning regular sets from queries and counterexamples. *Information and Computation* 75(2), 87–106 (1987)
5. Beyer, D., Henzinger, T.A., Singh, V.: Algorithms for Interface Synthesis. In: Damm, W., Hermanns, H. (eds.) *CAV 2007*. LNCS, vol. 4590, pp. 4–19. Springer, Heidelberg (2007)
6. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning Assumptions for Compositional Verification. In: Garavel, H., Hatcliff, J. (eds.) *TACAS 2003*. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
7. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-Modular Verification for Shared-Memory Programs. In: Le Métayer, D. (ed.) *ESOP 2002*. LNCS, vol. 2305, pp. 262–277. Springer, Heidelberg (2002)
8. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption Generation for Software Component Verification. In: *Proceedings of ASE 2002*, pp. 3–12. IEEE Computer Society, Los Alamitos (2002)
9. Groce, A., Peled, D., Yannakakis, M.: Adaptive Model Checking. In: Katoen, J.-P., Stevens, P. (eds.) *TACAS 2002*. LNCS, vol. 2280, p. 357. Springer, Heidelberg (2002)
10. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive Interfaces. In: *Proceedings of ESEC/SIGSOFT FSE 2005*, pp. 31–40 (2005)
11. Java PathFinder, <http://javapathfinder.sourceforge.net>
12. Jones, C.B.: Specification and Design of (Parallel) Programs. In: *Information Processing 1983: Proceedings of the IFIP 9th World Congress*, IFIP, pp. 321–332. North Holland, Amsterdam (1983)
13. Magee, J., Kramer, J.: *Concurrency: State Models & Java Programs*. John Wiley & Sons, Chichester (1999)
14. Margaria, T., Raffelt, H., Steffen, B., Leucker, M.: The LearnLib in FMICS-jETI. In: *Proceedings of ICECCS 2007* (2007)
15. Mehltitz, P.: Trust Your Model - Verifying Aerospace System Models with Java Pathfinder. In: *IEEE/Aero* (2008)
16. Pasareanu, C.S., Giannakopoulou, D., Gheorghiu Bobaru, M., Cobleigh, J.M., Barringer, H.: Learning to Divide-and-Conquer: Applying the L* Algorithm to Automate Assume-Guarantee Reasoning. In: *FMSD* (January 2008)
17. Pnueli, A.: In Transition from Global to Modular Temporal Reasoning about Programs. In: *Logic and Models of Concurrent Systems*, vol. 13, pp. 123–144 (1984)
18. Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking. In: Johansson, T. (ed.) *FSE 2003*. LNCS, vol. 2887, pp. 188–197. Springer, Heidelberg (2003)
19. Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. In: *Proceedings of ISSTA 2002*, pp. 218–228 (2002)

A Formal Way from Text to Code Templates

Guido Wachsmuth

Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
guwac@gk-metrik.de

Abstract. We present an approach to define template languages for generating syntactically correct code. In the first part of the paper, we define the syntax and semantics of a template language for text generation. We use Natural Semantics for expressing both the static and the dynamic semantics of the language. In the second part, we deal with template languages for code generation in a particular target language. We provide construction steps for the syntax and semantics of such languages. The approach is generic and can be applied to any target language.

1 Introduction

Code generation forms a central part of Model-driven Engineering (MDE). Template languages provide means to specify code generation. They are used in mature modelling frameworks, for example *Java Emitter Templates* [1] in the Eclipse Modeling Framework [2] or openArchitectureWare's *XPand* [3] in the Eclipse Graphical Modeling Framework [4]. With its *MOF Model to Text Transformation Language* (MOF M2T) [5], the Object Management Group proposes a standardised template language for model to text transformations. Beyond MDE, template languages are generally used in generative programming. For example, *StringTemplate* [6] is used for parser generation with ANTLR [7] as well as for generating web pages [8].

Template languages allow to specify code in concrete syntax. To generate code in different target languages, most template languages treat code simply as text. As a consequence, template languages do not provide any static judgement on syntactical correctness of templates with respect to a particular target language. In this paper, we investigate a formal method to develop a template language TL_λ for generating syntactically correct code in a given target language λ . We enhance the grammar of λ to derive a grammar for TL_λ . For this enhancement, we rely on well-defined grammar adaptation steps [9]. We use Natural Semantics [10] for expressing both the static and the dynamic semantics of TL_λ . The approach is generic and can be applied to any target language.

The remainder of the paper is structured as follows: In Sec. 2, we describe the formal foundations of our approach, i.e. grammar adaptation and Natural Semantics. In Sec. 3, we define syntax and semantics of the core concepts of template languages. In Sec. 4, we provide construction steps for template languages concerned with true code generation. The paper is concluded in Sec. 5.

```

adaptation = step*
  step = introduce rrule | include rule | fold rule
        | foreach vn : yielder do step* endfor
rrule = rule | nt*
rule = nt = elem*
elem = nt | vn | kw | [vn] | elem *
yielder = N | M | L
  nt    nonterminals
  vn    variable names
  kw    keywords

```

Fig. 1. An operator suite for grammar adaptation

2 Preliminaries

Grammar adaptation. In this paper, we employ formal grammar transformations for the stepwise adaptation of grammars [9]. This approach offers several benefits: First, the derivation of the template language is formally defined by the application of well-defined adaptation operators. Second, we prevent accidental modifications of the target language since preservation properties of adaptation operators are well understood [9]. Third, the adaptation is generic and can be applied to any target language. Finally, existing tool support automates the derivation process [11].

We rely on a very restricted operator suite for this paper. Figure 1 shows its concrete syntax: First, we can **introduce** a new rule for a fresh nonterminal. We use the same operator to introduce nonterminals without a defining rule (e.g. for morphem classes). Second, we can **include** an additional rule for a nonterminal. If the nonterminal is yet undefined, **include** operates as **introduce**. Third, we can **fold** a phrase. This introduces a fresh nonterminal defined by the phrase. All occurrences of the phrase are replaced by the nonterminal. All these operators are purely constructive, i.e. they extend the language defined by the grammar [9]. Finally, we can execute operators **foreach** nonterminal in a set. There are three yielders of such sets: **N** yields all nonterminals except morphem classes, **M** yields all morphem classes, and **L** yields all nonterminals to which a Kleene closure is applied. Inside a **foreach** block, a variable provides access to the current nonterminal. `[.]` yields a nonterminal’s name as a keyword.

Natural Semantics. In this paper, we use Natural Semantics [10] for the static and dynamic semantics of template languages. Typically, template languages are functional languages [6]. Describing functional languages by Natural Semantics is well understood. For example, Natural Semantics was used for both the static and dynamic semantics in the formal specification of *Standard ML* [12]. The general idea of a semantics definition in Natural Semantics is to provide axioms and inference rules for *judgements* over syntactical and semantical *domains*. A static semantics is typically described in terms of well-typedness judgements for

```

 $tcoll = tmpl^*$ 
 $tmpl = \ll \text{define } tn \text{ targ}^* \gg \text{tstmt}^* \ll \text{enddef} \gg$ 
 $targ = ttype \text{ } tvn$ 
 $tstmt = text \mid \ll \text{expr} \gg \mid \ll \text{expand } tn \text{ } \text{expr}^* \gg$ 
            $\mid \ll \text{if } \text{expr} \gg \text{tstmt}^* \ll \text{else} \gg \text{tstmt}^* \ll \text{endif} \gg$ 
            $\mid \ll \text{for } tvn \text{ in } \text{expr} \gg \text{tstmt}^* \ll \text{endfor} \gg$ 
 $tn$       template names
 $text$     text phrases
 $\text{expr}$    expressions
 $ttype$    variable types
 $tvn$     variable names

```

Fig. 2. Syntactical domains of $TL_{\perp}$

the various syntactical domains of a language. A dynamic semantics is described in an operational style by judgements for the execution of language instances.

We follow some notational conventions in this paper: For semantical domains, we use standard domain constructors, namely products (“ $\times$ ”), list types (“ * ”), and function domains (“ $\rightarrow_{fin}$ ”). We restrict ourselves to pointwise definition of functions, i.e. these functions are only defined for a finite subset of the domain X in $f : X \rightarrow_{fin} Y$. For most $x \in X$ we have that $f(x) = \perp$. The entirely undefined function is denoted by “ $\perp$ ”. In inference rules, we reuse domain names as stems of meta-variables. Tuples and sequences are constructed via $\langle \cdot, \dots, \cdot \rangle$. In order to concatenate several lists to a new one, we use the notation $\langle \cdot : \dots : \cdot \rangle$. Update of a function f for a point x to return y is denoted by $f[x \mapsto y]$. Function application is highlighted using the notation $f @ x$.

3 A Core Text Template Language

In this section, we provide formal semantics of the core concepts in template languages for text generation. For this purpose, we introduce $TL_{\perp}$, a functional language similar to StringTemplate, XPand, and even more to MOF M2T.

Syntax. Figure 2 shows a grammar of the template language $TL_{\perp}$. We briefly read its rules: A template collection $tcoll$ consists of a list $tmpl^*$ of templates. A template declares a list $targ^*$ of arguments and a list $tstmt^*$ of template statements. Template statements include simple text, expression evaluation, template expansion, conditional statement, and iteration.

Template languages typically comprise an expression language for navigating the source model. This sublanguage highly depends on the technological space of the source model. In the grammarware space, we need to navigate syntax trees. In the modelware space, we need to navigate object-oriented models. For this reason, we do not specify an expression sublanguage in detail. Instead, we specify several requirements for its syntax and semantics. Syntactically, we assume domains for expressions (expr), types ($ttype$), and variables (tvn).

<i>Domains</i>	
$\tau = ttype$	(Expression types)
$T = tvn \rightarrow_{fin} \tau$	(Variable type table)
$\Theta = tn \rightarrow_{fin} \tau^*$	(Template type table)
<i>Principal judgements</i>	
$\vdash tcoll$	(Well-typedness of template collections)
$\Theta \vdash tmpl : \Theta$	(Extraction of type context)
$\Theta \vdash tmpl$	(Well-typedness of templates)
$\Theta, T \vdash tstmt$	(Well-typedness of template statements)
$T \vdash texpr : \tau$	(Well-typedness of template expressions)
$\vdash_L \tau : \tau$	(Decomposition of list types)
$\vdash_B \tau$	(Booleanness of types)

Fig. 3. Model of $TL_{\perp}$ static semantics

Static semantics. The domains involved in the static semantics and the principal judgements are shown in Fig. 3. For the expression sublanguage, we require a domain of types (τ), a domain for variable environments (T) to store variable types, and a judgement of well-typedness of expressions under a given environment. Furthermore, we require types for list-like data structures. We assume the judgment $\vdash_L \tau : \tau'$ to hold for a list type τ iff the list elements are of type τ' . The judgement $\vdash_B \tau$ is expected to hold iff τ is boolean-like, that is, elements of this type can be mapped to boolean values.

As for the core concepts of $TL_{\perp}$, we define a domain for type tables (Θ) to store the argument types of templates. Judgements concern the extraction of a type table for a given template collection, the overall well-typedness of a template collection, the well-typedness of a template definition for a given type table, and the well-typedness of a statement for a given type table and variable environment.

Figure 4 lists the inference rules for the judgements. Since most of them are straightforward, we do not discuss all these rules in detail. Let us focus on the well-typedness judgement for statements. $\Theta, T \vdash tstmt$ is meant to hold if the statement $tstmt$ is well-typed in the context of Θ and T . The parameter Θ covers the template types whereas T corresponds to an environment mapping variables for template arguments or iterations to types.

The axiom [text] encodes the well-typedness of any text. The rule [exp] defines well-typedness for expression evaluation: An expression evaluation is well-typed if its expression has a type in the context of the current variable environment.

The premises of rule [expand] say that a template expansion is well-typed if (1) a table look-up for tn delivers the types of the template arguments and (2) the actual parameter types are subsumed by the formal ones.

The rule [for] defines well-typedness of iteration as follows: (1) The type τ of the iteration expression is determined under the current variable environment T . This type needs to be a list type with a corresponding type τ' for the list

<i>Well-typedness of template collections</i>	$\vdash \text{tcoll}$
$\frac{\perp \vdash \text{tmpl}_1 : \Theta_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta_{n-1} \vdash \text{tmpl}_n : \Theta_n \quad \wedge \quad \Theta_n \vdash \text{tmpl}_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta_n \vdash \text{tmpl}_n}{\vdash \langle \text{tmpl}_1, \dots, \text{tmpl}_n \rangle}$	[collection]
<i>Extraction of type context</i>	$\Theta \vdash \text{tmpl} : \Theta$
$\frac{\text{targ}_1 = \text{ttype}_1 \text{ tvn}_1 \quad \wedge \quad \dots \quad \wedge \quad \text{targ}_n = \text{ttype}_n \text{ tvn}_n \quad \wedge \quad \Theta @ \text{tn} = \perp \quad \wedge \quad \Theta' = \Theta[\text{tn} \mapsto \langle \text{ttype}_1, \dots, \text{ttype}_n \rangle]}{\Theta \vdash \ll \text{define tn } \langle \text{targ}_1, \dots, \text{targ} \rangle \gg \dots \ll \text{enddef} \gg : \Theta'}$	[extract]
<i>Well-typedness of templates</i>	$\Theta \vdash \text{tmpl}$
$\frac{\text{targ}_1 = \text{ttype}_1 \text{ tvn}_1 \quad \wedge \quad \dots \quad \wedge \quad \text{targ}_n = \text{ttype}_n \text{ tvn}_n \quad \wedge \quad T = \perp[\text{tvn}_1 \mapsto \text{ttype}_1, \dots, \text{tvn}_n \mapsto \text{ttype}_n] \quad \wedge \quad \Theta, T \vdash \text{tstmt}_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta, T \vdash \text{tstmt}_m}{\Theta \vdash \ll \text{define tn } \langle \text{targ}_1, \dots, \text{targ}_n \rangle \gg \langle \text{tstmt}_1, \dots, \text{tstmt}_m \rangle \ll \text{enddef} \gg}$	[template]
<i>Well-typedness of statements</i>	$\Theta, T \vdash \text{tstmt}$
$\Theta, T \vdash \text{text}$	[text]
$\frac{T \vdash \text{texpr} : \tau}{\Theta, T \vdash \ll \text{texpr} \gg}$	[expr]
$\frac{(1) \quad \Theta @ \text{tn} = \langle \tau_1, \dots, \tau_n \rangle \quad (2) \wedge T \vdash \text{texpr}_1 : \tau_1 \quad \wedge \quad \dots \quad \wedge \quad T \vdash \text{texpr}_n : \tau_n}{\Theta, T \vdash \ll \text{expand tn } \langle \text{texpr}_1, \dots, \text{texpr}_n \rangle \gg}$	[expand]
$\frac{(1) \quad T \vdash \text{texpr} : \tau \quad \wedge \quad \vdash_L \tau : \tau' \quad (2) \wedge T' = T[\text{tvn} \mapsto \tau'] \quad (3) \wedge \Theta, T' \vdash \text{tstmt}_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta, T' \vdash \text{tstmt}_n}{\Theta, T \vdash \ll \text{for tvn in texpr} \gg \langle \text{tstmt}_1, \dots, \text{tstmt}_n \rangle \ll \text{endfor} \gg}$	[for]
$\frac{(1) \quad T \vdash \text{texpr} : \tau \quad \wedge \quad \vdash_B \tau \quad (2) \wedge \Theta, T \vdash \text{tstmt}_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta, T \vdash \text{tstmt}_m}{\Theta, T \vdash \ll \text{if texpr} \gg \langle \text{tstmt}_1, \dots, \text{tstmt}_n \rangle \ll \text{else} \gg \langle \text{tstmt}_{n+1}, \dots, \text{tstmt}_m \rangle \ll \text{endif} \gg}$	[if]

Fig. 4. Rules of $TL_{\perp}$ static semantics

<i>Domains</i>	
ν	(Expression values)
$\beta = \mathbf{true} \mid \mathbf{false}$	(Boolean values)
$\varphi = \text{text}$	(Text phrases)
$\psi = \varphi^*$	
$T = \text{tvn} \rightarrow_{\text{fin}} \nu$	(Variable environment)
$\Theta = \text{tn} \rightarrow_{\text{fin}} (\text{tvn}^* \times \text{tstmt}^*)$	(Template code table)
<i>Principal judgements</i>	
$\vdash \text{tcoll} \Rightarrow \Theta$	(Extraction of code table)
$\Theta \vdash \text{tmpl} \Rightarrow \psi$	(Execution of template statements)
$T, \Theta \vdash \text{tstmt} \Rightarrow \psi$	(Evaluation of template expressions)
$T \vdash \text{texpr} \Rightarrow \nu$	(Text conversion of values)
$\vdash \nu \Rightarrow \varphi$	(Decomposition of lists)
$\vdash_L \nu \Rightarrow \nu^*$	(Decomposition of lists)
$\vdash_B \nu \Rightarrow \beta$	(Boolean conversion of values)

Fig. 5. Model of $TL_{\perp}$ dynamic semantics

elements. (2) A new variable environment T' is retrieved by updating the type of the iteration variable tvn to τ' . (3) All statements in the iteration need to be well-typed under the new environment.

The rule [if] for conditional statements reads similarly: (1) The type τ of the condition expression is determined. This type needs to be boolean-like. (2) All statements in a conditional statement need to be well-typed in the current context.

Dynamic semantics. While the static semantics is concerned with well-typedness judgements, the dynamic semantics provides judgements about text generation. Figure 5 shows the model of the dynamic semantics. We use a style that emphasises the similarities of the models of static and dynamic semantics. While Θ covers the *template types* in the static case, it models the *template code* in the dynamic case. Similar variations apply to the principal judgements. That is, code table extraction corresponds to type table extraction, text generation out of template statements to well-typedness of template statements, and text generation out of template expressions to type assignment for template expressions.

For the expression sublanguage, we require a domain of values (ν), a domain for variable environments (T) to store variable values, and a judgement for evaluating expressions under a given environment. The structure of this judgement implies side-condition free evaluation. This ensures model view separation [13]. Furthermore, we assume the judgment $\vdash_L \nu \Rightarrow \langle \nu_1, \dots, \nu_n \rangle$ to hold for a list-like data structure ν iff $\nu_1, \dots, \nu_n$ are the elements of this structure. The judgement $\vdash_B \nu \Rightarrow \beta$ is expected to hold iff ν can be converted into the boolean value β . Finally, we require a judgement $\vdash \nu \Rightarrow \varphi$ for the conversion of values to text.

Figure 6 lists the inference rules for the judgements. Again, we do not discuss all these rules in detail. Let us focus on the text generation judgement for

<i>Well-typedness of template collections</i>	$\vdash tcoll$
$\frac{\perp \vdash tmpl_1 : \Theta_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta_{n-1} \vdash tmpl_n : \Theta_n \quad \wedge \quad \Theta_n \vdash tmpl_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta_n \vdash tmpl_n}{\vdash \langle tmpl_1, \dots, tmpl_n \rangle}$	[collection]
<i>Extraction of type context</i>	$\Theta \vdash tmpl : \Theta$
$\frac{targ_1 = ttype_1 \quad tvn_1 \quad \wedge \quad \dots \quad \wedge \quad targ_n = ttype_n \quad tvn_n \quad \wedge \quad \Theta @ tn = \perp \quad \wedge \quad \Theta' = \Theta[tn \mapsto \langle ttype_1, \dots, ttype_n \rangle]}{\Theta \vdash \ll \text{define } tn \langle targ_1, \dots, targ \rangle \gg \dots \ll \text{enddef} \gg : \Theta'}$	[extract]
<i>Well-typedness of templates</i>	$\Theta \vdash tmpl$
$\frac{targ_1 = ttype_1 \quad tvn_1 \quad \wedge \quad \dots \quad \wedge \quad targ_n = ttype_n \quad tvn_n \quad \wedge \quad T = \perp[tvn_1 \mapsto ttype_1, \dots, tvn_n \mapsto ttype_n] \quad \wedge \quad \Theta, T \vdash tstmt_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta, T \vdash tstmt_m}{\Theta \vdash \ll \text{define } tn \langle targ_1, \dots, targ_n \rangle \gg \langle tstmt_1, \dots, tstmt_m \rangle \ll \text{enddef} \gg}$	[template]
<i>Well-typedness of statements</i>	$\Theta, T \vdash tstmt$
$\Theta, T \vdash text$	[text]
$\frac{T \vdash texpr : \tau}{\Theta, T \vdash \ll texpr \gg}$	[expr]
$\frac{(1) \quad \Theta @ tn = \langle \tau_1, \dots, \tau_n \rangle \quad (2) \wedge T \vdash texpr_1 : \tau_1 \quad \wedge \quad \dots \quad \wedge \quad T \vdash texpr_n : \tau_n}{\Theta, T \vdash \ll \text{expand } tn \langle texpr_1, \dots, texpr_n \rangle \gg}$	[expand]
$\frac{(1) \quad T \vdash texpr : \tau \quad \wedge \quad \vdash_L \tau : \tau' \quad (2) \wedge T' = T[tvn \mapsto \tau'] \quad (3) \wedge \Theta, T' \vdash tstmt_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta, T' \vdash tstmt_n}{\Theta, T \vdash \ll \text{for } tvn \text{ in } texpr \gg \langle tstmt_1, \dots, tstmt_n \rangle \ll \text{endfor} \gg}$	[for]
$\frac{(1) \quad T \vdash texpr : \tau \quad \wedge \quad \vdash_B \tau \quad (2) \wedge \Theta, T \vdash tstmt_1 \quad \wedge \quad \dots \quad \wedge \quad \Theta, T \vdash tstmt_m}{\Theta, T \vdash \ll \text{if } texpr \gg \langle tstmt_1, \dots, tstmt_n \rangle \ll \text{else} \gg \langle tstmt_{n+1}, \dots, tstmt_m \rangle \ll \text{endif} \gg}$	[if]

Fig. 6. Rules of $TL_{\perp}$ dynamic semantics

statements. $T, \Theta \vdash \text{tstmt} \Rightarrow \psi$ is meant to hold if the statement *tstmt* generates a list of text phrases ψ in the context of Θ and T .

The axiom [text] states that text generates itself. The rule [expr] defines text generation for template expression. The expression is evaluated in the context of the current variable environment to a value which in turn is converted to text.

The premises of rule [expand] specifies text generation by template expansion as follows: (1) A table look-up for *tn* delivers the parameter names and the statements of the template. (2) Parameter expressions are evaluated under the current variable environment T and (3) the results are assigned to the parameter names in a fresh variable environment T' . (4) The statements are evaluated under the new variable environment and the resulting text phrases are concatenated.

The rule [for] defines iterative text generation as follows: (1) The value ν of the iteration expression is determined under the current variable environment T . This value needs to be a list of elements $\nu_1, \dots, \nu_m$. (2) New variable environments $T_1, \dots, T_m$ are retrieved by updating the value of the iteration variable *tvn* to the corresponding value. (3) All statements in the iteration are executed under each new environment and (4) the generated text phrases are concatenated.

The rules [then] and [else] for conditional statements read similarly: (1) The value ν of the condition expression is evaluated under the current variable environment T and converted into a boolean value. (2) For **true**, the rule [then] executes the first branch. For **false**, the rule [else] executes the second branch. All statements in the branch are executed and (3) the generated text phrases are concatenated.

4 Generating Code Template Languages

In this section, we provide a generic approach to derive a template language from a target language. The resulting template language ensures the generation of syntactically correct code with respect to the target language. The derivation is split into three phases: First, we rely on grammar adaptation to syntactically enhance the target language with constructs for template definitions. This steps results in a grammar for the template language. Second, we rely on Natural Semantics to define the static semantics of the resulting template language. This includes a static judgement about the syntactical correctness of templates with respect to the target language. Finally, we rely again on Natural Semantics to define the dynamic semantics of the resulting template language.

Syntactical enhancement. We use grammar adaptation to enhance the grammar of a target language. Figure 7 gives the corresponding adaptation script which can be applied generically to any target language. First, we introduce expressions, types, variable names, template names, and template arguments. While names are typically morphem classes, expressions and types need to refer to the grammar of an expression language. Second, we enhance the definition of each nonterminal except morphem classes. We include rules for template expansion and for a conditional statement. Additionally, we include a rule for templates. Third, we enhance the definition of each morphem class. We fold each

```

introduce texpr ttype tvn tn
introduce targ = ttype tvn
foreach nt : N
  include nt =  $\ll \text{expand } tn \text{ } texpr^* \gg$ 
  include nt =  $\ll \text{if } texpr^* \gg \text{ } nt \ll \text{else} \gg nt \ll \text{endif} \gg$ 
  include tmpl =  $\ll \text{define } [nt] \text{ } tn \text{ } targ^* \gg nt \ll \text{enddef} \gg$ 
  include tstmt = nt
  include dn =  $[nt]$ 
endfor
foreach nt : M
  fold ntM = nt
  include ntM =  $\ll texpr \gg$ 
  include tstmt = ntM
  include dn =  $[nt]$ 
endfor
foreach nt : L
  fold ntL = nt
  include ntL =  $\ll \text{for } tvn \text{ in } texpr \gg nt \ll \text{endfor} \gg$ 
  include tstmt = ntL
endfor
introduce tcoll = tmpl*

```

Fig. 7. Generic adaptation script for the syntactical enhancement of a target language

morphem class to a fresh nonterminal and include a rule for expression evaluation. Fourth, we enhance the definition of nonterminals occurring in a Kleene closure. We fold each of these domains to a fresh nonterminal and include a rule for iteration. Finally, we introduce template collections.

The result of the adaptation is a grammar for a template language particularly concerned with the target language. In this template language, templates are associated with a particular syntactical domain of the target language. Figure 8 gives an example. The upper part of the figure shows the grammar of a simple programming language *PL*: A program *prog* consists of a list of statements *pstmt*^{*}. A statement is either a variable declaration, an assignment, a while loop, or a conditional statement. An expression *pepr* is either an integer number, a character string, a variable, a sum, a difference, or a string concatenation. The lower part of Fig. 8 shows the resulting grammar for the template language *TL_{PL}*.

Static semantics. During the syntactical enhancement, two helper domains *tstmt* and *dn* are constructed. This allows us to use a generic model of the static semantics. The model differs only slightly from the model for *TL_⊥*. Figure 9 highlights the modifications. In Θ , we keep the syntactical domain of the template in addition to the parameter types. In the well-typedness judgement for template statements, we assign the addressed syntactical domain. Furthermore, we add a judgement yielding the syntactical domain of an expression when converted into text.

```

pprog = begin pstmt* end
pstmt = pvn : ptype | pvn := pexpr | while pexpr do pstmt* od
        | if pexpr then pstmt* else pstmt* fi
pexpr = pvn | in | cs | pexpr + pexpr | pexpr - pexpr | pexpr || pexpr
ptype = int | string
pvn    variable names
in     integer numbers
cs     character strings

```

```

tcoll = tmpl*
tmpl = << define pprog tn targ* >>pprog << enddef >>
      | << define pstmt tn targ* >>pstmt << enddef >>
      | << define pexpr tn targ* >>pexpr << enddef >>
      | << define ptype tn targ* >>ptype << enddef >>
targ = ttype tvn
pprog = begin pstmt*_L end
      | << expand tn texpr* >>
      | << if texpr* >> pprog << else >> pprog << endif >>
pstmt = pvn_M : ptype | pvn_M := pexpr | while pexpr do pstmt*_L od
      | if pexpr then pstmt*_L else pstmt*_L fi
      | << expand tn texpr* >>
      | << if texpr* >> pstmt << else >> pstmt << endif >>
pstmt_L = pstmt | << for tvn in texpr >> pstmt << endfor >>
pvn_M = pvn | << texpr >>
...
tstmt = pprog | pstmt | pexpr | ptype | pvn_M | in_M | cs_M | pstmt_L
dn = pstmt | pexpr | ptype | pvn | in | cs

```

Fig. 8. Syntactical domains of a simple programming language PL and its corresponding template language TL_{PL}

Domains

$$\Theta = tn \rightarrow_{fin} (\tau^* \times \boxed{dn})$$

(Template types and domains)

Principal judgements

$$\Theta, T \vdash tstmt : \boxed{dn}$$

(Well-typedness of template statements)

$$\boxed{\vdash \tau : dn}$$

(Syntactical domains of expression types)

Fig. 9. Generic model of TL_{PL} static semantics (excerpt)

There are two kinds of inference rules. First, we provide generic rules for language constructs introduced during the syntactical enhancement. Second, we need to generate inference rules covering original constructs of the target language. The generic rules deal with template statements introduced by the syntactical enhancement. Only minor modifications are needed to the inference rules of the static semantics of $TL_{\perp}$. These modifications deal with domain

assignment for templates, statements, and expressions. The upper part of Fig. 10 presents the affected rules.

The lower part of the figure shows some of the inference rules generated for TL_{PL} . For each morphem class m , we generate an axiom of the form

$$\Theta, T \vdash m : [m]$$

where $[m]$ yields the name of m . For each grammar rule $nt = \langle e_1, \dots, e_n \rangle$, we generate an inference rule of the form

$$\frac{premise_1 \wedge \dots \wedge premise_m}{\Theta, T \vdash \langle p_1, \dots, p_n \rangle : [nt]}$$

Each right-hand side element e_i is mapped to a corresponding pattern p_i in the inference rule and premises might be added.

1. A nonterminal nt is mapped to a fresh variable v of the corresponding domain. The premise $\Theta, T \vdash v : [nt]$ is added.
2. A Kleene closure nt^* is mapped to a list pattern $\langle v_1, \dots, v_k \rangle$ with fresh variables. The premises $\Theta, T \vdash v_1 : [nt] \wedge \dots \wedge \Theta, T \vdash v_k : [nt]$ are added.
3. A morphem class m is mapped to a fresh variable v of the corresponding domain for m_M which resulted from folding m during the syntactical enhancement. The premise $\Theta, T \vdash v : [m]$ is added.
4. A keyword is mapped to itself. No premise is added.

Dynamic semantics. As for the static semantics, we can reuse the model of $TL_{\perp}$ dynamic semantics. The only modification affects the code table: We only need to store single template statements instead of statement lists. Again, there are two kinds of inference rules, generated and generic ones. The upper part of Fig. 11 shows the generic rules. Modifications to inference rules of $TL_{\perp}$ dynamic semantics are highlighted.

The lower part of the figure shows generated rules for TL_{PL} . Generation is quite similar to the static case. For each morphem class m , we generate an axiom of the form

$$\Theta, T \vdash m \Rightarrow m$$

stating that a morphem generates itself. For each grammar rule $nt = \langle e_1, \dots, e_n \rangle$, we generate an inference rule of the form

$$\frac{premise_1 \wedge \dots \wedge premise_m}{\Theta, T \vdash \langle p_1, \dots, p_n \rangle \Rightarrow \langle \psi_1 : \dots : \psi_n \rangle}$$

The mapping of right-hand side element e_i to a corresponding pattern p_i is the same as in the static case. The following premises are added:

1. For a nonterminal, the premise $\Theta, T \vdash v \Rightarrow \psi_i$ is added.
2. For a Kleene closure, premises $\Theta, T \vdash v_1 \Rightarrow \psi_{i,1} \wedge \dots \wedge \Theta, T \vdash v_1 \Rightarrow \psi_{i,m} \wedge \psi_i = \langle \psi_{i,1} : \dots : \psi_{i,k} \rangle$ are added.
3. For a morphem class, the premise $\Theta, T \vdash v \Rightarrow \psi_i$ is added.
4. For a keyword kw , the premise $\psi_i = kw$ is added.

<i>Extraction of type context</i>	$\boxed{\Theta \vdash \text{tmpl} : \Theta}$
$\frac{\begin{array}{l} \text{targ}_1 = \text{ttype}_1 \text{ tvn}_1 \wedge \dots \wedge \text{targ}_n = \text{ttype}_n \text{ tvn}_n \\ \wedge \Theta' = \Theta[\text{tn} \mapsto \langle \text{ttype}_1, \dots, \text{ttype}_n \rangle, \boxed{\text{dn}} \rangle] \end{array}}{\Theta \vdash \ll \text{define } \boxed{\text{dn}} \text{ tn } \langle \text{targ}_1, \dots, \text{targ}_n \rangle \gg \dots \ll \text{enddef} \gg : \Theta'}$	[extract]
<i>Well-typedness of templates</i>	$\boxed{\Theta \vdash \text{tmpl} : \text{dn}}$
$\frac{\begin{array}{l} \text{targ}_1 = \text{ttype}_1 \text{ tvn}_1 \wedge \dots \wedge \text{targ}_n = \text{ttype}_n \text{ tvn}_n \\ \wedge T = \perp[\text{tvn}_1 \mapsto \text{ttype}_1, \dots, \text{tvn}_n \mapsto \text{ttype}_n] \\ \wedge \Theta, T \vdash \text{tstmt} : \boxed{\text{dn}} \end{array}}{\Theta \vdash \ll \text{define } \boxed{\text{dn}} \text{ tn } \langle \text{targ}_1, \dots, \text{targ}_n \rangle \gg \boxed{\text{tstmt}} \ll \text{enddef} \gg : \boxed{\text{dn}}}$	[template]
<i>Well-typedness of statements</i>	$\boxed{\Theta, T \vdash \text{tstmt} : \text{dn}}$
$\frac{T \vdash \text{texpr} : \tau \wedge \boxed{\vdash \tau : \text{dn}}}{\Theta, T \vdash \ll \text{texpr} \gg : \boxed{\text{dn}}}$	[expr]
$\frac{\begin{array}{l} \Theta @ \text{tn} = \langle \tau_1, \dots, \tau_n \rangle, \boxed{\text{dn}} \rangle \\ \wedge T \vdash \text{texpr}_1 : \tau_1 \wedge \dots \wedge T \vdash \text{texpr}_n : \tau_n \end{array}}{\Theta, T \vdash \ll \text{expand tn } \langle \text{texpr}_1, \dots, \text{texpr}_n \rangle \gg : \boxed{\text{dn}}}$	[expand]
$\frac{\begin{array}{l} T \vdash \text{texpr} : \tau \wedge \vdash_L \tau : \tau' \\ \wedge T' = T[\text{tvn} \mapsto \tau'] \\ \wedge \Theta, T' \vdash \text{tstmt} : \boxed{\text{dn}} \end{array}}{\Theta, T \vdash \ll \text{for tvn in texpr} \gg \boxed{\text{tstmt}} \ll \text{endfor} \gg : \boxed{\text{dn}}}$	[for]
$\frac{\begin{array}{l} T \vdash \text{texpr} : \tau \wedge \vdash_B \tau \\ \wedge \Theta, T \vdash \text{tstmt}_1 : \boxed{\text{dn}} \wedge \Theta, T \vdash \text{tstmt}_2 : \boxed{\text{dn}} \end{array}}{\Theta, T \vdash \ll \text{if texpr} \gg \boxed{\text{tstmt}_1} \ll \text{else} \gg \boxed{\text{tstmt}_2} \ll \text{endif} \gg : \boxed{\text{dn}}}$	[if]
$\frac{(2) \ \Theta, T \vdash \text{pstmt}_1 : \text{pstmt} \wedge \dots \wedge \Theta, T \vdash \text{pstmt}_n : \text{pstmt}}{\Theta, T \vdash \text{begin } \langle \text{pstmt}_1, \dots, \text{pstmt}_n \rangle \text{ end} : \text{pprog}}$	[pprog]
$\frac{\begin{array}{l} (3) \ \Theta, T \vdash \text{pvn}_M : \text{pvn} \\ (1) \wedge \Theta, T \vdash \text{ptype} : \text{ptype} \end{array}}{\Theta, T \vdash \text{pvn}_M : \text{ptype} : \text{pstmt}}$	[pstmt ₁]
$\vdots$	
$\Theta, T \vdash \text{cs} : \text{cs}$	[cs]

Fig. 10. Rules of TL_{PL} static semantics (excerpt)

<i>Code table extraction</i>	$\Theta \vdash \text{tmpl} \Rightarrow \Theta$
$\frac{\begin{array}{c} \text{targ}_1 = \text{ttype}_1 \text{ tvn}_1 \quad \wedge \quad \dots \quad \wedge \quad \text{targ}_n = \text{ttype}_n \text{ tvn}_n \\ \wedge \quad \Theta' = \Theta[\text{tn} \mapsto \langle \langle \text{tvn}_1, \dots, \text{tvn}_n \rangle, \boxed{\text{tstmt}} \rangle] \end{array}}{\Theta \vdash \ll \text{define } \boxed{\text{dn}} \text{ tn } \langle \text{targ}_1, \dots, \text{targ}_n \rangle \gg \boxed{\text{tstmt}} \ll \text{enddef} \gg \Rightarrow \Theta'} \quad [\text{extract}]$	
<i>Statement evaluation</i>	$T, \Theta \vdash \text{tstmt} \Rightarrow \psi$
$\begin{array}{l} \Theta @ \text{tn} = \langle \langle \text{tvn}_1, \dots, \text{tvn}_n \rangle, \boxed{\text{tstmt}} \rangle \\ \wedge T \vdash \text{texpr}_1 \Rightarrow \nu_1 \quad \wedge \quad \dots \quad \wedge \quad T \vdash \text{texpr}_n \Rightarrow \nu_n \\ \wedge T' = \perp[\text{tvn}_1 \mapsto \nu_1, \dots, \text{tvn}_n \mapsto \nu_n] \\ \wedge T', \Theta \vdash \boxed{\text{tstmt}} \Rightarrow \boxed{\psi} \end{array}$ $T, \Theta \vdash \ll \text{expand tn } \langle \text{texpr}_1, \dots, \text{texpr}_n \rangle \gg \Rightarrow \boxed{\psi} \quad [\text{expand}]$	
$\begin{array}{l} T \vdash \text{texpr} \Rightarrow \nu \quad \wedge \quad \vdash_L \nu \Rightarrow \langle \nu_1, \dots, \nu_m \rangle \\ \wedge T_1 = T[\text{tvn} \mapsto \nu_1] \quad \wedge \quad \dots \quad \wedge \quad T_m = T[\text{tvn} \mapsto \nu_m] \\ \wedge T_1, \Theta \vdash \boxed{\text{tstmt}} \Rightarrow \boxed{\psi_1} \\ \wedge \dots \\ \wedge T_m, \Theta \vdash \boxed{\text{tstmt}} \Rightarrow \boxed{\psi_m} \\ \wedge \psi = \langle \boxed{\psi_1} : \dots : \boxed{\psi_m} \rangle \end{array}$ $T, \Theta \vdash \ll \text{for tvn in texpr} \gg \boxed{\text{tstmt}} \ll \text{endfor} \gg \Rightarrow \psi \quad [\text{for}]$	
$\begin{array}{l} T \vdash \text{texpr} \Rightarrow \nu \quad \wedge \quad \vdash_B \nu \Rightarrow \text{true} \\ \wedge T, \Theta \vdash \boxed{\text{tstmt}} \Rightarrow \boxed{\psi} \end{array}$ $T, \Theta \vdash \ll \text{if texpr} \gg \boxed{\text{tstmt}} \ll \text{else} \gg \dots \ll \text{endif} \gg \Rightarrow \psi \quad [\text{then}]$	
$\begin{array}{l} T \vdash \text{texpr} \Rightarrow \nu \quad \wedge \quad \vdash_B \nu \Rightarrow \text{false} \\ \wedge T, \Theta \vdash \boxed{\text{tstmt}} \Rightarrow \boxed{\psi} \end{array}$ $T, \Theta \vdash \ll \text{if texpr} \gg \dots \ll \text{else} \gg \boxed{\text{tstmt}} \ll \text{endif} \gg \Rightarrow \psi \quad [\text{else}]$	
$\begin{array}{l} (4) \quad \psi_1 = \text{begin} \\ (2) \wedge \Theta, T \vdash \text{pstmt}_1 \Rightarrow \psi_{2,1} \quad \wedge \quad \dots \quad \wedge \quad \Theta, T \vdash \text{pstmt}_n \Rightarrow \psi_{2,n} \\ \quad \wedge \psi_2 = \langle \psi_{2,1} : \dots : \psi_{2,n} \rangle \\ (4) \wedge \psi_3 = \text{end} \end{array}$ $\Theta, T \vdash \text{begin } \langle \text{pstmt}_1, \dots, \text{pstmt}_n \rangle \text{ end} \Rightarrow \langle \psi_1 : \psi_2 : \psi_3 \rangle \quad [\text{pprog}]$	
$\begin{array}{l} (3) \quad \Theta, T \vdash \text{pvn}_M \Rightarrow \psi_1 \\ (4) \wedge \psi_2 = : \\ (1) \wedge \Theta, T \vdash \text{ptype} \Rightarrow \psi_3 \end{array}$ $\Theta, T \vdash \text{pvn}_M : \text{ptype} \Rightarrow \langle \psi_1 : \psi_2 : \psi_3 \rangle \quad [\text{pstmt}_1]$ ⋮	

Fig. 11. Rules of TL_{PL} dynamic semantics (excerpt)

5 Conclusion

Contribution. We give formal semantics to a core template language for text generation. This way, we provide a starting point for semantics definitions of template languages like MOF M2T. Furthermore, we make a transition from text to true code generation. We show how a template language concerned with a particular target language can be derived from the target language itself. The resulting template language has clear semantics and ensures the syntactical correctness of generated code. This contributes to the E in MDE. The approach is generic and can be applied to any target language. In general, this paper contributes to software language engineering.

Related Work. In the technological space of grammarware, some program transformation languages like ASF [14] and Stratego/XT [15] allow to specify program transformations in the concrete syntax of the object language. This enables code generation based on concrete syntax. [16], provides a case study for code generation with Stratego/XT. Furthermore, the benefits of using concrete syntax in transformations and of a judgement about the syntactical correctness of transformations are discussed. In [17], a generic method to integrate target language grammars into arbitrary program transformation languages is presented. The method is based on modular syntax definitions in the syntax definition formalism SDF. In contrast to this approach, we prevent manual integration by using formal grammar adaptation steps. Furthermore, we address the semantics of the transformation language. In general, our approach is related to the embedding of languages, e.g. SQL, into host languages [18,19], e.g. Java [20].

Future Work. In this paper, we concern syntactical correctness of generated code. In a next step, the static semantics of the target language should be taken into account. This includes a restricted form of well-typedness checks (in terms of the target language) for templates. Another important point is tool support. When it comes to the target language, current template editors miss many useful features like error highlighting and code completion. This inhibits productive template engineering. Our approach is a starting point to overcome these shortcomings: Grammars and Natural Semantics allow for generic prototypical tool support. Thus, we can directly develop language tools on base of the formal syntax and semantics of a derived template language.

Acknowledgement. This work is supported by grants from the DFG (German Research Foundation, Graduiertenkolleg METRIK). The author is indebted to Ralf Lämmel for providing him with layout templates for the Natural Semantics descriptions.

References

1. The Eclipse Foundation: Java Emitter Templates (JET) (2008), <http://www.eclipse.org/modeling/emf/>
2. The Eclipse Foundation: Eclipse Modeling Framework (EMF) (2007), <http://www.eclipse.org/modeling/emf/>
3. OpenArchitectureWare: XPand (2008), <http://www.openarchitectureware.org>
4. The Eclipse Foundation: Eclipse Graphical Modeling Framework (GMF) (2008), <http://www.eclipse.org/gmf/>
5. Object Management Group: MOF Model to Text Transformation Language, version 1.0 (January 2008)
6. Parr, T.J.: A functional language for generating structured text. Draft (2006)
7. Parr, T.J., Quong, R.W.: Antlr: a predicated-ll(k) parser generator. *Softw. Pract. Exper.* 25(7), 789–810 (1995)
8. Parr, T.J.: Intelligent web site page generation (2007)
9. Lämmel, R.: Grammar adaptation. In: Oliveira, J.N., Zave, P. (eds.) *FME 2001*. LNCS, vol. 2021, pp. 550–570. Springer, Heidelberg (2001)
10. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) *STACS 1987*. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
11. Lämmel, R., Wachsmuth, G.: Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. *ENTCS* 44(2) (2001)
12. Milner, R., Tofte, M., Harper, R., MacQueen, D.: *The Definition of Standard ML (Revised)*. MIT Press, Cambridge (1997)
13. Parr, T.J.: Enforcing strict model-view separation in template engines. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) *WWW 2004*, pp. 224–233. ACM, New York (2004)
14. van den Brand, M., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In: Wilhelm, R. (ed.) *CC 2001*. LNCS, vol. 2027, p. 365. Springer, Heidelberg (2001)
15. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/xt 0.16: components for transformation systems. In: Hatcliff, J., Tip, F. (eds.) *PEPM 2006*, pp. 95–99. ACM, New York (2006)
16. Hemel, Z., Kats, L.C.L., Visser, E.: Code generation by model transformation. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) *ICMT 2008*. LNCS, vol. 5063, pp. 183–198. Springer, Heidelberg (2008)
17. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (eds.) *GPCE 2002*. LNCS, vol. 2487, pp. 299–315. Springer, Heidelberg (2002)
18. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. In: Consel, C., Lawall, J.L. (eds.) *GPCE 2007*, pp. 3–12. ACM, New York (2007)
19. Gao, J., Heimdahl, M., Van Wyk, E.: Flexible and extensible notations for modeling languages. In: Dwyer, M.B., Lopes, A. (eds.) *FASE 2007*. LNCS, vol. 4422, pp. 102–116. Springer, Heidelberg (2007)
20. Van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute grammar-based language extensions for java. In: Ernst, E. (ed.) *ECOOP 2007*. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)

Context-Aware Adaptive Services: The PLASTIC Approach*

Marco Autili, Paolo Di Benedetto, and Paola Inverardi

Dipartimento di Informatica - Università degli Studi di L'Aquila, Italy
{marco.autili,paolo.dibenedetto,inverard}@di.univaq.it

Abstract. The near future envisions a pervasive heterogeneous computing infrastructure that makes it possible for mobile users to run software services on a variety of devices, from networks of devices to stand-alone wireless resource-constrained ones. To ensure that users meet their non-functional requirements by experiencing the best Quality of Service according to their needs and specific contexts of use, services need to be context-aware and adaptable. The development and the execution of such services is a big challenge and it is far to be solved. In this paper we present our experience in this direction by describing our approach to context-aware adaptive services within the IST PLASTIC project. The approach makes use of CHAMELEON, a formal framework for adaptive Java applications.

1 Introduction

Pervasive computing is an emerging paradigm that is rapidly changing the ways we use technologies to perform everyday tasks. The wide spread of small computing devices and the introduction of new communication infrastructures make it possible for mobile users to run software services on a variety of devices from networks of devices to stand-alone wireless resource-constrained ones. B3G networks [30] have gained importance as an effective way to realize pervasive computing by offering broad connectivity through various network technologies pursuing the convergence of wireless telecommunication and IP networks (e.g., UMTS, WiFi and Bluetooth).

Ubiquitous networking empowered by B3G networks makes it possible for mobile users to access networked software services across heterogeneous infrastructures through resource-constrained devices characterized by their *heterogeneity* and *limitedness*. Software applications running over this kind of infrastructure must cope with resource scarcity and with the inherent faulty and heterogeneous nature of this environment [9]. Indeed to ensure that users meet their non-functional requirements by experiencing the best Quality of Service (QoS) according to their needs and specific contexts of use, services need to be context-aware and adaptable. The development and the execution of such services is a big challenge for the

* This work is part of the IST PLASTIC project and has been funded by the European Commission, FP6 contract number 026955, <http://www.ist-plastic.org/>.

research community and it far to be solved. This paper describes our experience in this direction and, by extending our preliminary work in [17], presents our approach to context-aware adaptive services within the IST PLASTIC project [21]. The approach makes use of CHAMELEON, a formal framework for adaptive Java applications [7]. The goal of the PLASTIC project is the rapid and easy development, deployment and execution of adaptable services for B3G networks [30]. PLASTIC builds on Web Services (WS) and component-based technologies, and introduces the notion of *requested Service Level Specification* (SLS) and *offered SLS* to deal with the non-functional dimensions - i.e., QoS - that will be used to establish the *Service Level Agreement* (SLA) between the service consumer and the service provider. Services are implemented as adaptable components and are deployed on heterogeneous resource-constrained mobile devices. The new contribution of this paper is to describe the two types of adaptation supported by a PLASTIC service, namely adaptation driven by the (requested) SLS and adaptation with respect to the characteristic of the execution context. We name these two types of adaptation *SLS-based adaptation* and *context-aware adaptation*, respectively. Service adaptability is achieved via a development paradigm based on SLS and resource-aware programming supported by CHAMELEON that takes into account the characteristics of the hosting environment like resource availability, network conditions, and the SLSs.

The paper is organized as follows: Section 2 defines the two dimensions of the PLASTIC adaptation and Section 3 briefly describes the PLASTIC development environment. Section 4 introduces the CHAMELEON framework and describes how it has been used for implementing the PLASTIC Service-oriented Interaction Pattern. Section 5 describes the actual implementation of CHAMELEON by showing how the approach has been applied to the PLASTIC e-Health Remote Diagnosis case study. Related work is discussed in Section 6. Concluding remarks and future directions are given in Section 7.

2 PLASTIC Adaptation(s)

In this section we define the two dimensions of adaptation supported by a PLASTIC service.

The first kind of adaptation we consider is *SLS-based adaptation*. For this adaptation the context of interest is represented by the preferences expressed by the user in the *requested SLS* and by the provider in the *offered SLS*. The SLS represents the non-functional characteristics of the service. It is coupled with the service interface and it is used to establish the *SLA* between a user requesting the service and a provider of the service. The SLA defines the conditions on the QoS accepted by both the service consumer and the service provider. Adaptation in PLASTIC is used by the service provider to exhibit a service with different SLS. The exposed service is actually a generic one that at “matching time” is adapted with respect to the requested SLS and the available execution context.

The second type of adaptation is *context-aware adaptation*. For this adaptation the context of interest is represented by the provider, network and consumer contexts which represent the environment in which a service is provisioned and

consumed. PLASTIC applications are deployed over heterogeneous, resource-constrained devices, thus the provider and consumer contexts are their respective device resource characteristics - e.g., screen resolution, CPU frequency, memory size, available radio interfaces, networks in reach. The B3G network context identifies the characteristics of the (multiple) network(s) between consumer and provider such as number and type of networks, number of active users, number of available services, security, transmission protocol, access policy, etc. The network context impacts on the network QoS in terms of bitrate, transfer delay, packet-loss, network coverage, price and energy consumption for using a given network. Adaptation in this case is practiced by the service programmer who can produce a generic service that can be customized with respect to the actual resource characteristics of the execution context so that its execution can be correctly supported.

In PLASTIC adaptation is restricted at discovery time, that is at the moment in which the service execution context and the user QoS preferences (requested SLS) are known, and a SLA can be put in place. The advantage of this approach to adaptation is that it is cost effective since it is a compromise between static and fully dynamic adaptation. The limit is that unpredictable changes of contexts might invalidate the SLA and thus make the service unusable. However in highly heterogeneous and autonomous infrastructures like B3G, QoS attributes cannot be (a priori) guaranteed. Thus SLA violations can occur and must be monitored and detected. Indeed, in [17] we present a first attempt to tackle this problem by monitoring service execution to detect possible SLA violations. Upon violation either (i) the service can be adapted to the new context so that it can continue respecting the agreed SLA or (ii) a re-negotiation of the SLA can happen which can in turn drive a new adaptation.

3 PLASTIC Development Environment

In this section we briefly describe the PLASTIC development environment from the perspective of a service developer. PLASTIC provides a set of tools¹ that are all based on the *PLASTIC Service Conceptual Model*² and support the service life cycle, from design to implementation to validation to execution. The conceptual model formalizes all the concepts needed for developing B3G service-oriented applications. The overall approach is model driven, starting from the conceptual model till the execution service model used to monitor the service.

With reference to Figure 1, the PLASTIC conceptual model has been concretely implemented as a *UML2 profile* and, by means of the PLASTIC development environment tools, the functional behavior of the service and its non-functional characteristics can be modeled. Then, non functional analysis and development activities are iteratively performed [10]. The analysis aims at computing QoS indices of the service at different levels of detail, from early design

¹ Available at <http://gforge.inria.fr/projects/plastic-dvp/>

² The *Formal description of the PLASTIC conceptual model and of its relationship with the PLASTIC platform toolset* is available at <http://www.ist-plastic.org/>

to implementation to publication, to support designers and programmers in the development of services that satisfy the specified QoSs, i.e., SLSs. In PLASTIC, among QoS measures, we only consider performance and reliability. The principal performance indices are *utilization* and *throughput* of (logical and physical) resources, as well as *response time* for a given task. The considered reliability measures are, instead, *probability of failure on demand* and *mean time to failure*. Discrete set of values - e.g., high, medium, low - are used to identify ranges.

The analysis and validation activities rely on artifacts produced from the PLASTIC service model through different model transformations. For instance, the service model editor [8] and the SLA editor, that are part of the PLASTIC platform toolset, are integrated through a model-to-code transformation. Once the service model has been specified, a model-to-code transformation can be performed in order to translate the parts of the service model that are needed for specifying the agreement (e.g., involved parties, other services, operations, etc.) into a HUTN file (i.e., a human-usable textual notation for SLA) which the SLA editor is capable to import. The SLS attached to the published service and the SLA are formally specified by using the language SLang [16].

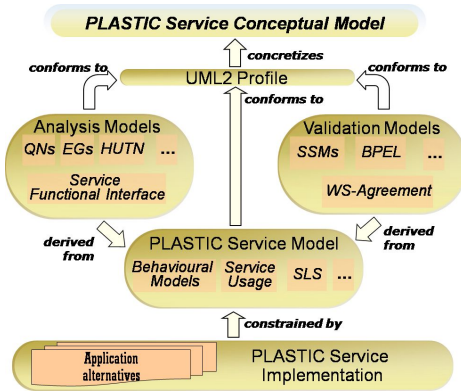


Fig. 1. Development Environment

monitoring [23].

For the purposes of this work, hereafter we will concentrate on how PLASTIC services are implemented and how the two types of adaptation, namely, *SLS-based adaptation* and *context-aware adaptation*, are supported.

4 PLASTIC Services Deployment and Access

PLASTIC services are implemented by using the *CHAMELEON Programming Model* (presented in Section 5) that, extending the Java language, permits developers to implement services in terms of *generic code*. Such a generic code, opportunely preprocessed, generates a set of different *application alternatives*, i.e., different standard Java components that represent different ways of

After the service has been implemented, the PLASTIC validation framework enables the off-line, prior to the service publication, and on-line, after the service publication, validation of the services with respect to functional - through test models such as Symbolic State Machines (SSMs) or based on BPEL processes - and non-functional properties [12]. This means that through validation it is possible to assess whether the service exhibits the given SLS. On-line validation concerns the “checking” activities that are performed after service deployment such as SLA

implementing a provider/consumer application. Therefore, an adaptable software service might be implemented as and consumed by different application alternatives (i.e., different adaptations). Each alternative is characterized by (i) the resources it demands to be correctly executed (i.e., *Resource Demand*) and (ii) the so called *Code-embedded SLSs*. The latter are QoS indices retrieved by the non-functional analysis. They are specified by the developers at generic code level through annotations attached to methods and are then automatically “injected” into the application alternatives by CHAMELEON. As it will be clear in Section 5, code-embedded SLSs contribute to determine the final SLSs offered by the different alternatives.

In the remainder of this section we describe how adaptive PLASTIC services are published, discovered and accessed (Section 4.1), and how both provider- and consumer-side application alternatives (stored in the *Applications Registry*) can be over-the-air delivered and deployed on devices (Section 4.2).

4.1 The PLASTIC Service-Oriented Interaction Pattern

The PLASTIC Service-oriented Interaction Pattern for provision and consumption of adaptive services (Figure 2) involves the following steps.

The service provider publishes into the *PLASTIC Registry* the service description in terms of both functional specifications and associated offered SLSs (1). Specifically, a provider can publish a service with different SLSs, each one associated to a different provider-side application alternative that represents a way of adapting the service. The service consumer queries the PLASTIC registry for a service functionality, additionally specifying the requested SLS (2). The PLASTIC

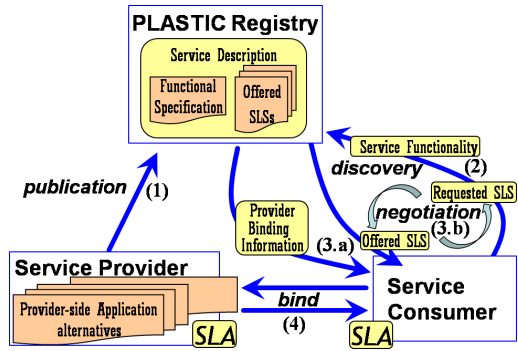


Fig. 2. PLASTIC Interaction Pattern

registry searches for service descriptions that satisfy the consumer request. If suitable service descriptions are present in the service registry, the service consumer can choose one of them on the base of their offered SLSs. After the service consumer accepts an offered SLS, the registry returns the actual reference to the provider-side application alternative that implements the (adapted) service with the accepted SLS (3.a). Thus, the SLA can be established and the service consumption can take place (4). If no suitable published service is able to directly and fully satisfy the requested SLS, negotiation is necessary (3.b). The negotiation phase starts by offering a set of alternative SLSs. The consumer can accept one of the proposed SLSs, or perform an “adjusted” request by reiterating the process till an SLA is possibly reached. In Figure 2 the box *SLA* labeling the provider and the consumer represents the agreement reached by both of them.

4.2 Over-the-Air Application Alternatives Delivery and Deployment

With reference to Figure 3, we call *PLASTIC-enabled devices* the devices deploying and running the CHAMELEON *Client* component and the *PLASTIC B3G Middleware* [13] that together are able to retrieve contextual information. A PLASTIC-enabled device provides a declarative description of the execution context in terms of the resources it supplies (i.e., *Resource Supply*) and a description of the impact that computational elements (i.e., code instructions) have on the resources (i.e., *Resource Consumption Profile*). PLASTIC-enabled devices can host both service consumers and service providers.

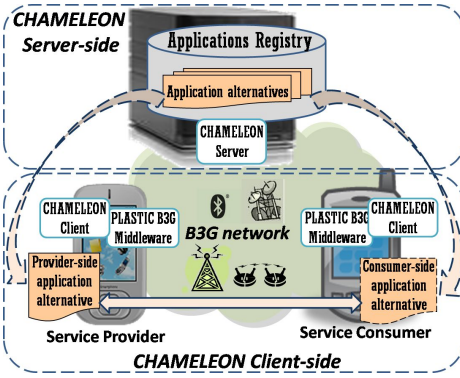


Fig. 3. CHAMELEON Client-Server

Indeed, the CHAMELEON *Client* component can interact with the CHAMELEON *Server* component in order to dynamically download (from the CHAMELEON *Applications Registry*), deploy and run (i) provider-side application alternatives, e.g., a .war file for a web application to be exposed as service and, if needed, (ii) ad-hoc consumer-side application alternatives, e.g., a .jar and a .jad files for a midlet to be used for consuming a service. Note that, the CHAMELEON client is a lightweight component that effortlessly runs on limited devices; the CHAMELEON server runs on a

back-end server which does not suffer resource limitations.

By referring to Figure 4, (i) a provider that wants to offer a service S can connect directly to the CHAMELEON application registry and search for an application among a set of already implemented application alternatives to be exposed as service S . The choice is based on the functional description of S and the code-embedded SLSs restricting to those alternatives that are compatible (i.e., will run safely) with respect to the execution context (i.e., resource supply and resource consumption profile). The chosen alternative will be automatically delivered and deployed via the Over-The-Air (OTA) provisioning

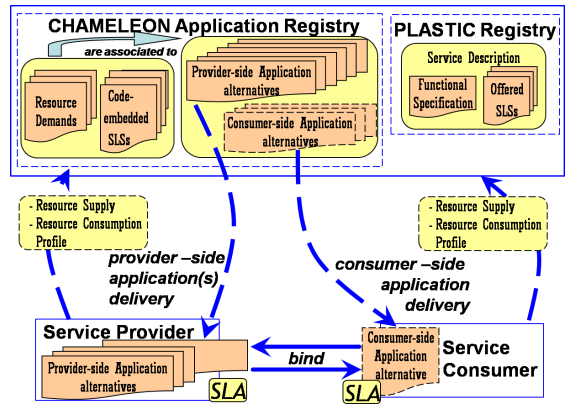


Fig. 4. Application Alternatives Delivery

technique [2] on the provider device. Note that, the process can be used for delivering and deploying more than one alternative $A_1, \dots, A_n$ for S . Then, the functional description of S will be published into the PLASTIC registry along with the final offered SLSs defined on the base of the code-embedded SLSs, associated to $A_1, \dots, A_n$, possibly refined by the provider through the SLA editor (see Section 3).

(ii) Differently, if to consume the service S an ad-hoc client application needs to be deployed on the consumer device, the final offered SLSs published by the provider will be defined on the base of the code-embedded SLSs associated to all the chosen provider application alternatives $A_1, \dots, A_n$ combined with the code-embedded SLSs associated to all the consumer application alternatives of S . In this case, upon the consumer request, the PLASTIC registry relies on the CHAMELEON application registry to search for a set of compatible consumer-side application alternatives $C_1, \dots, C_m$ that are able to safely run on the consumer device and properly interact with the service S (i.e., $A_1, \dots, A_n$). The delivered consumer application alternative will depend on the SLS chosen among the only offered SLSs related to $C_1, \dots, C_m$.

5 CHAMELEON-Based PLASTIC Services Implementation

In this section we present the CHAMELEON framework and show how it supports the implementation of PLASTIC adaptive services and their provision and consumption. For more detailed presentations of the formalisms and definitions underlying the framework please refer to [6,7] (and references therein). The CHAMELEON framework has been implemented [7] on the Java platform and it exploits XML-based technologies for data exchange. The framework will be presented by means of the PLASTIC e-Health Remote Diagnosis case study.

The e-Health service allows to establish a link between patients and assistants providing support for video conferences, medical agenda management, alarm generation and management, remote diagnosis (RD), etc. Both professionals and patients are considered to be nomadic and can move with their mobile devices e.g., move from outdoor to their office/home. Therefore, mobility becomes a key issue and services need to be adapted, both to the heterogeneous networks capabilities and to the terminals that could be used by professionals and patients.

For the purposes of this paper, we focus on the RD functionality. When an alarm is generated on the patient side due to some event like patient inactivity, dangerous vital parameters or help request, the e-Health system contacts one or more doctors to perform a diagnosis. On the doctor side the diagnosis process is supported by an RD consumer application that, connecting to an RD service provider installed on the patient side, allows the doctor to check the patient camera and monitor vital parameters, e.g., blood pressure, temperature, heart rate. The whole service is adapted according to both the patient (the provider) and doctor (the consumer) context.

► **Programming Model.** Referring to right-hand side of Figure 5, the *Development Environment* (DE) is based on a *Programming Model* that provides

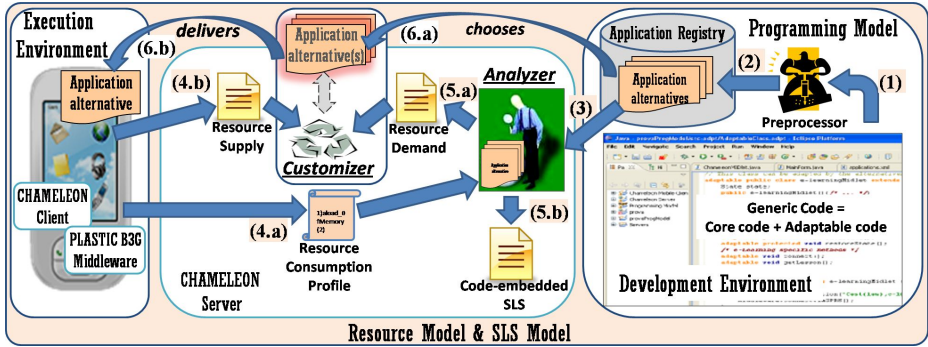


Fig. 5. CHAMELEON Framework

developers with a set of ad-hoc extensions to Java for easily specifying services code in a flexible and declarative way. As already mentioned, services code is a *generic* code that consists of two parts: the *core* and the *adaptable* code - see in Figure 5 the screen-shot of our DE implemented as an Eclipse plugin [7]. The core code is the frozen portion of the application and represents its invariant semantics. The adaptable one represents the degree of variability that makes the code capable to adapt. The generic code is preprocessed by the CHAMELEON *Preprocessor* (1), also part of the DE, and a set of different standard Java application alternatives is automatically derived and stored into the *Application Registry* (2).

Figure 6 represents an excerpt of a generic code, as part of the RD consumer MIDlet, written by the developer according to the CHAMELEON programming model. The core code is a standard code and, hence, can be specified through standard Java classes; the adaptable code is an “extended” code and is specified through *Adaptable Classes* that declare one or more *Adaptable Methods*. Methods are the smallest building blocks that constitute the entry-points for a behavior that can be adapted. *Alternative Classes* define how one or more adaptable methods can actually be adapted. For instance, the adaptable class *RemoteDiagnosis* declares three adaptable methods (see the keyword **adaptable**): *visualCheck*, *vitalParameters* and *connect*. The implementation of these adaptable methods is defined by two alternative classes (see the keywords **alternative** and **adapts**): *HighSupportRD* and *LowSupportRD*. Such a generic code will be preprocessed by the CHAMELEON *Preprocessor* and the two standard Java application alternatives described in Table 1 will be derived by suitably combining the adaptable methods implementations specified by the various alternatives.

The programming model also allows for specifying additional information by using *Annotations*. Annotations are specified at the generic code level and permit to specify resource demand (*Resource Annotation*), code-embedded SLS (*SLS Annotation*), upper bound on the number of loop iterations (*Loop Annotation*) and recursive method calls (*Call Annotation*).

```

adaptable public class RemoteDiagnosis extends MIDlet {
    adaptable void connect();
    adaptable void visualCheck();
    adaptable void vitalParameters();
    ...
}
/*****
alternative class HighSupportRD adapts RemoteDiagnosis {
    private void connect() { ...
        Annotation.SLSAnnotation("Throughput(high)");
        Annotation.resourceAnnotation("WiFi(true)");
        QoSInfo.setBitrate(HIGH);
        PlasticMiddleware.selectNetwork(QoSInfo);
    }
    private void visualCheck() {/*shows video streaming from patient's cameras*/}
    private void vitalParameters() {/*draws diagrams of vital parameters*/}
}
*****/
alternative class LowSupportRD adapts RemoteDiagnosis {
    private void connect() { ...
        Annotation.SLSAnnotation("Throughput(low)");
        QoSInfo.setBitrate(LOW);
        PlasticMiddleware.selectNetwork(QoSInfo);
    }
    private void visualCheck() {/* shows patient's camera images */}
    private void vitalParameters() {/*shows textual data of vital parameters*/}
}

```

Fig. 6. An Adaptable MIDlet

Table 1. Remote Diagnosis Consumer Application Alternatives

Alter-native	Features	Resource Demand	Code-embedded SLS
Cons_1	Shows patient's camera images and textual data of vital parameters	{Energy(200)}	{Throughput(low)}
Cons_2	Shows video streaming and images from patient's cameras and draws diagrams of vital parameters	{WiFi(true), Energy(400)}	{Throughput(high)}

Annotations can be specified by calls to “do nothing” static methods of the Annotation class in Figure 7. For instance, in

```

public class Annotation {
    public static void resourceAnnotation(String ann){};
    public static void SLSAnnotation(String ann){};
    public static void loopAnnotation(int n){};
    public static void callAnnotation(int n){};
}

```

Fig. 7. Annotation Class

Figure 6 the method calls `Annotation.resourceAnnotation("WiFi(true)")` and `Annotation.SLSAnnotation("Throughput(high)")` are used to specify that the `HighSupportRD` alternative class demands for a WiFi radio-interface on the consumer device and provides a high quality remote diagnosis support. Note that, a high throughput is related to the usage of the resource WiFi. These annotations will contribute to determine the resource demand and the code-embedded SLSs, respectively, of the derived alternatives. Indeed, the whole framework is based on the *Resource* and *SLS Models* (see Figure 5) that, in particular, allow for specifying conforming resource and SLS annotations, respectively.

► **Resource and SLS Models.** The resource model is a formal model that allows the characterization of the resources needed to consume/provide a service and it is at the base of context-aware adaptation. The SLS model is a model that

permits developers to attach non-functional information at generic code level through code-embedded SLSs and is used for SLS-based adaptation purposes (see Section 4).

A resource is modeled as a typed identifier that can be associated to natural, boolean or enumerated values. Natural values are used for consumable resources whose availability varies during execution (e.g., energy, heap space). Boolean values define non-consumable resources that can be present or not (e.g., function libraries, network radio interfaces) and enumerated values define non-consumable resources that provide a restricted set of admissible values (e.g. screen resolution, network type). Figure 8 shows an example of some resource and SLS definitions for the RD case study. Both the *Resource Demand* and the *Resource Supply* are specified in terms of *resource sets* that couple resources to their values in the form $\{res_1(val_1), \dots, res_n(val_n)\}$. Table 1 also reports the resource demand and the code-embedded SLS calculated by the analyzer (see below). For example, the resource demand of the Cons_2 application alternative in Table 1, specifies that, to run safely, the alternative will require a WiFi network radio-interface (**WiFi(true)**) and a battery state-of-charge of the target consumer device at least of 400 energy units (**Energy(400)**).

The *SLS Model* bases itself around the same formalisms as the resource model and it is used for specifying SLSs. For example, the code-embedded SLS of the Cons_2 alternative of Table 1, specifies that the alternative offers a high throughput (**Throughput(high)**).

► **Chameleon Server.** Still referring to Figure 5, the *Analyzer* (running on the CHAMELEON server) is an interpreter that, abstracting a standard JVM, is able to analyze the application alternatives (3) and derive their resource consumption (5.a) and the code-embedded SLSs (5.b). The analyzer is parametric with respect to the characteristics of the execution environment as described through the *resource consumptions profile* sent by the device (4.a). We remind that the profile provides a characterization of the target execution environment, in terms of the impact that Java bytecode instructions have on the resources. Note that this impact depends on the execution environment since the same bytecode instruction may require different resources in different execution environments.

More precisely, these profiles associate resources consumption to particular patterns of bytecode instructions specified as *regular expressions*. Since the bytecode is a verbose language³, this allows to define the resource consumption associated to both basic instructions (e.g., **ipush**, **iload**, etc.) and complex ones, e.g., method calls. Figure 9 represents an example of a resource consumption profile. For instance, the last row states that a call to the **getLocalDevice()**

Resource Definition

```
defineRES Energy as Natural
defineRES Bluetooth as Boolean
```

SLS Definition

```
defineSLS Throughput as {low, medium, high}
defineSLS Mobility
as {low, medium, high}
```

Fig. 8. Resource and SLS Definitions

³ This is particularly true for method invocations where the method is uniquely identified by a fully qualified *id* (base class identifier + name + formal parameters).

1) <code>aload_0</code> $\rightarrow$ {Memory(2)}	2) <code>.*</code> $\rightarrow$ {Memory(1), Energy(1)}
3) <code>invokestatic LocalDevice.getLocalDevice()</code> $\rightarrow$ {Bluetooth(true), Energy(20)}	

Fig. 9. A Resource Consumption Profile

static method of the `LocalDevice` class within the `javax.bluetooth` library requires the presence of Bluetooth on the device (`Bluetooth (true)`), and it causes a consumption of the resource Energy equal to 20 cost units.

The analyzer performs a worst-case analysis of the program by statically inspecting the Java bytecode of the different application alternatives. More specifically, the analyzer scans each possible execution path of the application alternative bytecode by reconstructing (through the DAVA decompiler⁴) and traversing the bytecode abstract syntax tree (BAST). Within a path, each encountered instruction is matched against the resource consumption profile and, by combining the demand of each instruction, the overall resource demand of the path is derived. The final resource demand (5.a) of the application alternative will be the one of the most demanding execution path. At the same time the encountered SLS annotations contribute to determine the code-embedded SLS (5.b). The analyzer is based on an operational semantic that has been formalized using a transition system. It is out of the scope of this paper to go into details of our analysis technique, and we refer to [6] for further details.

$$\frac{\begin{array}{l} \text{IsLeaf}(n) \quad \text{Label}(n) = \text{instr} \\ \text{instr !Like}(\text{"invoke*"}) \\ \text{!IsAnnotation}(n) \\ r = b(\text{instr}) \end{array}}{\langle e, b, M, n \rangle \rightarrow_{ARA} \{\langle r, \phi \rangle\}}$$

Fig. 10. Fall-Back Leaf Rule

As an example, in Figure 10 we show a very simple rule that is applied when the transition system (reaching a BAST leaf node n) encounters a basic bytecode instruction `instr` that is neither a method invocation (i.e., `!Like("invoke*")`) nor an annotation (i.e., `!IsAnnotation(n)`). The rule simply uses the function b that matches `instr` against the resource consumption profile in order to obtain its resource demand r . The result is a set of pairs $\langle r, s \rangle$ where s contains the encountered code-embedded SLSs (empty in this case since `instr` is not an annotation).

Still referring to Figure 5, the resource demands (5.a) of the application alternatives together with the resource supply sent by the device (4.b) are used by the *Customizer* that is able to choose (6.a) and propose a set of “best” suited application alternatives, and deliver (6.b) consumer- and/or provider-side standard Java applications that (through the delivery mechanism described in Section 4.2) can be automatically deployed in the target devices for execution. The customizer bases on the notion of *compatibility* that is used to decide if an application alternative can run safely on the requesting device, i.e., if for every resource demanded by the alternative a “sufficient amount” is supplied by the execution environment. For instance, a resource supply {`WiFi(true)`, `Energy (300)`} would be compatible only with the resource demand of adaptation `Cons_1` in Table 1.

⁴ Available at <http://www.sable.mcgill.ca/dava/>

Table 2. Remote Diagnosis Provider Application Alternatives

Alternative	Features	Resource Demand	Code-embedded SLS
Prov_1	Transmits images from patient camera, stores and makes available manually inserted vital parameters values	{GPRS(true), Memory(128), ...}	{Throughput(low), Mobility(high)}
Prov_2	Streams video from patient camera, collects and makes available data automatically retrieved by measurement instruments, and provides all functionalities of alternative Prov_1	{Memory(512), PressureMeter(true), HRM(true), WiFi(true)...}	{Throughput(high), Mobility(medium)}

Now, let us assume that the two application alternatives in Table 2 have been derived for the provider (i.e., the patient): Prov_1 provides a high patient mobility⁵ (**Mobility(high)**) but a low throughput (**Throughput(low)**), Prov_2 provides a limited patient mobility but a high throughput.

Since to consume the RD service an ad-hoc client application needs to be deployed on the consumer device (see Section 4.2), the code-embedded SLSs associated to provider alternatives that will be stored in the CHAMELEON application registry will be those in Table 3. They results from the combination of the code-embedded SLSs associated to the provider alternatives in Table 2 with the ones associated to the consumer alternatives in Table 1. Note that merging **Throughput(low)** with **Throughput(high)** produces **Throughput(low)**. Still referring to Section 4.2, let

us now assume that a patient (a provider), deploying both the alternatives Prov_1 and Prov_2 of Table 2, has published the RD Service in the PLASTIC registry along with the offered SLSs associated to them in Table 3. Upon a doctor (a consumer) request, the PLASTIC registry relies on the CHAMELEON server to obtain a set of applications compatible with the doctor device. If the doctor device is compatible only with the consumer-side RD alternatives Cons_1 of Table 1, the doctor will be allowed to choose among only the offered SLSs related to Cons_1 (i.e., the combined SLSs of Cons_1-Prov_1 and Cons_1-Prov_2 in Table 3).

Table 3. Combined Code-embedded SLSs

	Cons_1	Cons_2
Prov_1	{Throughput(low), Mobility(high)}	{Throughput(low), Mobility(high)}
Prov_2	{Throughput(low), Mobility(medium)}	{Throughput(high), Mobility(medium)}

6 Related Work

Our resource model and analyzer can be related to other approaches to resource-oriented analysis. The MRG Project [5] proposes a framework (based on proof-carrying code) for giving correct guarantees that programs are free from run-time violations of resource bounds. The MRG approach is based on a resource-counting semantics that takes into account, through a resource component, the number of executed instructions and the maximum size of the stack frame. The same line of research is continued in the Mobius project [11]. For example, in [3]

⁵ Mobility describes the size of network coverage considered by the PLASTIC B3G middleware.

the authors propose static analysis framework for the cost analysis of sequential Java bytecode that adds cost relations for defining the cost of a program as a function of its input data size.

In [25] the authors present a framework that, at both deployment- and run-time, is able to estimate the energy consumption of a distributed Java-based system. In particular, at the component level, they integrate an energy cost model and a communication cost model that allow for estimating the overall energy cost of each component. This information can then be used by software engineers in order to make decisions when adapting an application.

Tivoli [19] provides a resource modeler tool that enables the specification of resources and allows for the automatic monitoring of them by instrumenting the environment in which programs are executed.

The worst-case execution-time (WCET) problem has been deeply studied in the literature. In [28], the authors give an exhaustive survey of methods and tools for estimating (under precise assumptions) the WCET of hard real-time systems. Even though not strictly related to our work, this work provides an in-depth insight into the static and dynamic program analyses techniques used in this research area. In particular, the work in [4] also addresses the WCET problem and proposes a parametric timing analysis that instead of computing a single numeric value for WCET, as done by numeric timing analyses, derives symbolic formulas for representing the WCET. By accounting for parameters of the program, processor behaviour, and by deriving parametric loop bounds, the proposed analysis allows for deriving precise WCETs.

All these approaches use a resource model and aim at giving an absolute (over-)estimation of resources' consumption. Our approach instead, does not aim at establishing a punctual estimation rather aims at supporting a reasoning mechanism for selecting alternatives. Its correctness thus is restricted to consistently reflect the ordering among resource consumptions of a set of alternatives that will run in the same execution context.

By exploring all the possible computation paths and by mapping JVM bytecode to a transition system, the CHAMELEON analyzer uses the same exhaustive technique of other existent tools such as Java Pathfinder (JPF) [26]. JPF checks for property violations (deadlocks or unhandled exceptions) traversing all possible execution paths. Differently from JPF, our analyzer gets rid of variable values and abstracts the JVM with respect to resource consumption. In this abstraction we consider bytecode instructions behavior only by taking into account their effects on resources.

For the sake of space, we cannot address all the recent related works in the wide domain of software services. In the following, we provide only some major references. In [20] an interesting approach is presented that considers separation of concerns between application logic and adaptation logic. The approach makes use of Java annotations to express metadata needed to enable dynamic adaptation. Stemming from the same separation of concerns idea, in [24] in the scope of the MUSIC project [1], the authors propose the design of a middleware- and architectural-based approach to support the dynamic

adaptation and reconfiguration of the components and service composition structure. They use a planning-based middleware that, based on metadata included in the available plans, enables the selection of the right alternative architectural plan for the current context. Similarly to us this approach is based on requested and offered QoS, and supports SLA negotiation. Differently from us, they do not consider adaptability to the device execution context basing on a resource oriented analysis that is parametric with respect to resource consumption profiles.

Current (Web-)service development technologies, e.g., [14,15,22,27,29] (just to cite some), address only the functional design of complex services, that is they do not take into account the extra-functional aspects (e.g., QoS requirements) and the context-awareness. Our process borrows concepts from these well assessed technologies and builds on them in order to make QoS issues, context-awareness and adaptiveness emerge in service development.

7 Discussion and Future Work

In this paper we have described how the CHAMELEON framework is used to realize a form of service adaptation in the IST PLASTIC project. CHAMELEON allows for the development and deployment of adaptable applications (consuming and providing services) targeted to mobile resource-constrained devices in the heterogeneous B3G network. We have proposed a Service-oriented Interaction Pattern for adaptable service provision and consumption, and have described how it is based on CHAMELEON to support the two-fold PLASTIC adaptation from both service provider side and consumer side.

Right now, PLASTIC adaptation happens at discovery time, thus the deployed application is customized (i.e., it is tailored) with respect to the context at binding time but, at run time, it is frozen with respect to evolution. If context changes at run time the service needs to dynamically adapt to continue respecting the reached SLA and dependability. As first attempt to tackle this problem, in [17] we propose a mechanism that, exploiting ad-hoc methods for *saving* and *restoring* the (current) application state, enables services' evolution against monitored SLA violations. Evolution is achieved by dynamically un-deploying the no longer apt alternative and subsequently (re-)deploying a new alternative that is able to preserve the agreed SLA. If no alternative is able to continue respecting the agreed SLA, a re-negotiation of the SLA can happen which can in turn drive a new adaptation.

Moreover, assuming that upon service request the user knows (at least a stochastic distribution of) the mobility pattern [18] they will follow during service usage, this permits to identify the successive finite contexts they can traverse during service usage. Following this approach, an enhanced version of CHAMELEON would be able to generate code that is a compromise between self-contained (i.e., embedding adaptation logic) and tailored adaptable code. The code would merge the adaptation alternatives that, associated to the specified mobility pattern, are necessary to preserve the *offered SLS*. The code will also embed some dynamic *adaptation logic* which is able to recognize context changes, seamlessly

“switching” among the embedded adaptation alternatives. In this way, a seamless evolution will be performed among the embedded alternatives associated to the mobility pattern, while the un-/re-deployment evolution will be performed when moving out of the mobility pattern’s context.

We are currently performing an empirical analysis of the framework in order to evaluate its correctness and its applicability to more complex real scenarios. Moreover, we are investigating how ontology based specifications might be used to establish a common vocabulary and relationships among resource/SLS types. This would allow to relate resource/SLS types and to predicate about a common set of related types. For instance, a demand of **Energy** could be related to a supply of **Battery**.

References

1. MUSIC Project, <http://www.ist-music.eu/>
2. Over-The-Air (OTA), <http://developers.sun.com/mobility/midp/articles/ota/>
3. 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)
4. Altmeyer, S., Hümbert, C., Lisper, B., Wilhelm, R.: Parametric Timing Analysis for Complex Architectures. In: Proc. of the 14th IEEE RTCSA, pp. 367–376. IEEE Computer Society Press, Los Alamitos (2008)
5. Aspinall, D., MacKenzie, K.: Mobile resource guarantees and policies. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (2006)
6. Autili, M., Di Benedetto, P., Inverardi, P.: Resource oriented static analysis of Java programs. Technical Report univaq-1243 (2008), <http://www.di.univaq.it/chameleon/output/download.php?fileID=1243>
7. Autili, M., Di Benedetto, P., Inverardi, P., Mancinelli, F.: Chameleon project - SEA group, <http://di.univaq.it/chameleon/>
8. Autili, M., Berardinelli, L., Cortellessa, V., Di Marco, A., Di Ruscio, D., Inverardi, P., Tivoli, M.: A development process for self-adapting service oriented applications. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 442–448. Springer, Heidelberg (2007)
9. Autili, M., Caporuscio, M., Issarny, V.: A reference model for service oriented middleware. Technical Report inria-00326479, INRIA Paris-Rocquencourt (2008)
10. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE TSE 30(5), 295–310 (2004)
11. Barthe, G.: Mobius, securing the next generation of java-based global computers. In: ERCIM News (2005)
12. Bertolino, A., De Angelis, G., Di Marco, A., Inverardi, P., Sabetta, A., Tivoli, M.: A Framework for Analyzing and Testing the Performance of Software Services. In: Proc. of the 3rd ISoLA. CCIS, vol. 17, Springer, Heidelberg (2008)
13. Caporuscio, M., Raverdy, P.-G., Moun gla, H., Issarny, V.: ubiSOAP: A service oriented middleware for seamless networking. In: Proc. of 6th ICSOC (2008)
14. Eclipse.org. Eclipse Web Standard Tools, <http://www.eclipse.org/webtools>
15. IBM. BPEL4WS, Business Process Execution Language for Web Services (2003)
16. Skene, J., Lamanna, D., Emmerich, W.: Precise service level agreements. In: Proc. of the 26th ICSE, pp. 179–188, Edinburgh, UK (May 2004)

17. Autili, M., Di Benedetto, P., Inverardi, P., Tamburri, D.A.: Towards self-evolving context-aware services. In: Proc. of CAMPUS (DisCoTec), vol. 11 (2008)
18. Di Marco, A., Mascolo, C.: Performance analysis and prediction of physically mobile systems. In: Proc. of WOSP, NY, USA, pp. 129–132 (2007)
19. Moeller, M., Callahan, B., Gucer, V., Hollis, J., Weber, S.: Introducing Tivoli Distributed Monitoring Workbench 4.1. IBM Redbooks (2002)
20. Paspallis, N., Papadopoulos, G.A.: An approach for developing adaptive, mobile applications with separation of concerns. In: COMPSAC (2006)
21. PLASTIC project, <http://www.ist-plastic.org>
22. A-MUSE Project. Methodological Framework for Freeband Services Development (2004), <https://doc.telin.nl/dscgi/ds.py/Get/File-47390/>
23. Raimondi, F., Skene, J., Emmerich, W.: Efficient Online Monitoring of Web-Service SLAs. In: Proc. of the 16th ACM SIGSOFT/FSE (November 2008)
24. Rouvoy, R., Eliassen, F., Floch, J., Hallsteinsen, S.O., Stav, E.: Composing components and services using a planning-based adaptation middleware. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 52–67. Springer, Heidelberg (2008)
25. Seo, C., Malek, S., Medvidovic, N.: An energy consumption framework for distributed java-based systems. In: ASE (2007)
26. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. ASE journal 10(2) (2003)
27. W3C. Web Service Definition Language, <http://www.w3.org/tr/wsd1>
28. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. Trans. on Embedded Computing Sys. 7(3), 1–53 (2008)
29. Yun, H., Kim, Y., Kim, E., Park, J.: Web Services Development Process. In: Proc. of Parallel and Distributed Computing and Systems (PDCS) (2005)
30. Zahariadis, T., Doshi, B.: Applications and services for the B3G/4G era. Wireless Comm. 11(5) (2004)

Synchronous Modeling and Validation of Priority Inheritance Schedulers^{*}

Erwan Jahier, Nicolas Halbwachs, and Pascal Raymond

VERIMAG (CNRS, UJF, INPG)
Grenoble, France

{Erwan.Jahier,Nicolas.Halbwachs,Pascal.Raymond}@imag.fr
<http://www-verimag.imag.fr>

Abstract. Architecture Description Languages (ADLs) allow embedded systems to be described as assemblies of hardware and software components. It is attractive to use such a global modelling as a basis for early system analysis. However, in such descriptions, the applicative software is often abstracted away, and is supposed to be developed in some host programming language. This forbids to take the applicative software into account in such early validation. To overcome this limitation, a solution consists in translating the ADL description into an executable model, which can be simulated and validated together with the software. In a previous paper [1], we proposed such a translation of AADL (Architecture Analysis & Design Language) specifications into an executable synchronous model.

The present paper is a continuation of this work, and deals with expressing the behavior of complex scheduling policies managing shared resources. We provide a synchronous specification for two shared resource scheduling protocols: the well-known basic priority inheritance protocol (BIP), and the priority ceiling protocol (PCP). This results in an automated translation of AADL models into a purely Boolean synchronous (Lustre) scheduler, that can be directly model-checked, possibly with the actual software.

Keywords: Embedded systems, Simulation, Scheduling, Formal Verification, Architecture Description Languages, Synchronous Languages.

1 Introduction

The European project ASSERT is devoted to safe model-driven design of embedded systems, with aerospace systems as a main application domain. Such systems are deployed on specific architectures that need to be described and simulated in order to allow early validation of the integrated system.

The approach taken in the ASSERT project is to describe the execution architecture separately from the software components. The target architecture is

^{*} This work was partially supported by the European Commission under the Integrated Project ASSERT, IST 004033, which ended in 2008.

described in the AADL architecture description language [2,3]. AADL provides a collection of classical systems components, which can be instantiated and assembled to describe the actual execution platform. In a typical AADL description, a system is made of several *computers*, communicating through *buses*; a computer is made of *memory* and *processors*, and a processor runs a *scheduler* and several *tasks*; at last, tasks are running *applicative software*. Those software components can be developed using several programming languages, including ADA, C, or even Scade and Lustre via a C wrapping.

AADL components are decorated with information like rates and Worst Case Execution Time (WCET) for periodic tasks, scheduling policy, etc. Those informations are intended to be used in the validation of the platform, mainly by checking properties like the absence of deadlocks, or the respect of deadlines. The functional part is expressed by the software components, and thus generally completely ignored, although it may influence some non-functional aspects. For instance, a software component may produce some event that wakes up a task; the scheduling environment and the execution times are then modified.

Our main objective is to perform simulation and validation that take into account both the system architecture and the functional aspects. We consider the case where software components are implemented in the synchronous programming language Lustre/Scade¹. Our proposal in [1] is to build automatically a simulator of the architecture, expressed in a synchronous language like the software components. This approach presents several advantages: first, synchronous languages are well-known to be able to express non-synchronous behaviors, while the converse is more difficult; now, getting all aspects in the same model allows both functional and system aspects to be considered jointly. For instance, in AADL, sporadic tasks can be activated by the output of some other components. Therefore, in such cases, more realistic simulation and finer-grained formal verification can be performed.

The translation proposed in [1] takes into account various asynchronous aspects of AADL such as task execution time, periodic or sporadic activations, multitasking (using Rate Monotonic Scheduling [5]), and clock drifts. The result is an executable integrated synchronous model, combining architecture behavior with actual software components, which can be validated with tools available for synchronous programs.

In this paper, we propose to extend this work by taking into account shared resources using different protocols (no lock, blocking, basic inheritance, priority ceiling). We also show how various properties related to determinism, schedulability, or the absence of deadlock can be automatically checked on given architecture models.

The article is organized as follows. We first recall in Section 2 the principles of simulation of AADL in the synchronous paradigm. Then we describe in Section 3 how to deal with shared resources and various shared access protocols in a synchronous program. Finally, we show in Section 4 how one can use the

¹ Scade is the industrial version of Lustre[4], and is maintained and distributed by the Esterel-Technology company.

resulting executable model to check various kinds of properties (determinism, schedulability, absence of deadlock), and to perform monitored simulations.

2 From AADL to Synchronous Programs

This section recalls the main features of the Architecture Analysis & Design Language (AADL), as well as the synchronous paradigm. Then it briefly recalls how the behavior of an (asynchronous) AADL model can be modeled by a non-deterministic synchronous program. This subject is presented in detail in [1].

2.1 The AADL Description Language

An AADL model is made of an arborescent assembly of software and hardware components [2,3]. A component is defined by an interface (input and output *ports*), a set of sub-components, a set of *connections* linking up the subcomponents ports, and a set of typed attributes (called *properties*). The main kinds of AADL components are the following.

Systems are top-level components; they describe the mapping between software and hardware components. *Device* components model hardware responsible for interfacing the system with its environment. They are typically used to represent sensors or actuators. From a functional point of view, they correspond to the inputs and the outputs of the system. *Processor* components are abstractions of hardware and software responsible for scheduling and executing threads.

Memory components (hardware) are used to specify the amount and the kind of memory that is available to other components.

Data components (software) are used to represent data types in the source text. Other components might have a shared access to data components. The access policy is controlled by the `ConcurrencyControlProtocol` property (lock, priority ceiling protocol, cf. Section 3). *Bus* components (hardware) are used to exchange data between components on different processors.

Process components are abstractions of software responsible for defining a memory space that can be accessed by the *thread* sub-components it contains. *Thread* components are abstractions of software responsible for executing applicative programs. When several threads run under the same processor, the sharing of the processor is managed by a runtime scheduler. The `dispatch_protocol` property is used to specify that scheduling policy. For instance, the value `periodic` means that the thread must be activated according to the specified `period`; the value `aperiodic` means that the thread is activated via one of the other components' output ports (called *event ports*). *Sub-program* components are the leaves of this arborescent description. Their implementations need to be provided in some host language. In our approach, if one wants to be able to formally analyze aperiodic threads whose activation depends on the functional output of some program component, one needs to provide for it a synchronous program (or at least a wrapper), e.g., written in Scade or Lustre. The property `compute_exec_time` specifies a range for the worst case execution time (WCET) of the program. In the sequel, we use the term *task* to denote a thread running a program.

2.2 The Synchronous Paradigm

We present now the essentials of the synchronous paradigm, focusing on the aspects that will be used later on.

A synchronous program (also called a node) is a dynamic system evolving on a discrete time scale. It has an internal memory made of state variables, inputs and outputs, and its behavior is a (virtually infinite) sequence of atomic reactions. Each reaction consists in reading current inputs, computing outputs, and updating the internal memory (state). In other terms, synchronous programs are a straightforward generalization of synchronous circuits (i.e., sequential circuits or Mealy machines), where data can be of arbitrary types rather than just Boolean values.

A synchronous program is characterized by a vector of inputs $\mathbf{i}$, a vector of outputs $\mathbf{o}$, a vector of state variables $\mathbf{s}$. Its semantics is defined by its initial state s_0 (the initial value of $\mathbf{s}$), and the functions f_o and f_s , respectively returning the output and the next state from the current inputs and the current state. For each instant t :

$$o_t = f_o(i_t, s_t) \quad ; \quad s_{t+1} = f_s(i_t, s_t)$$

A program without state is called a *combinational node*; usual functions like arithmetic or logical operators are then naturally lifted to synchronous programs.

For any data-type τ with a well-defined default value d , a “delay” (or register) program can be defined as follows:

$$s_0 = d \quad ; \quad f_o(i, s) = s \quad ; \quad f_s(i, s) = i$$

In the sequel we mainly use Boolean (resp. integer) registers, with false (resp. 0), and represented by the symbol $\bullet$.

The main characteristic of synchronous programs is the way they are composed: when connecting several sub-programs, a reaction of the whole program consists of a *simultaneous* reaction of all the components. In other words, the synchronous paradigm provides an idealized representation of parallelism.

An important consequence is that a big synchronous program can be described as a parallel composition of smaller sub-programs. In this approach, a program is described as a data-flow network of synchronous programs connected by wires. Fig. 1 shows the data-flow network of a synchronous counter, made of a delay node and three combinational nodes (two “if-then-else” and an adder). For the sake of conciseness, we use sets of equations rather than drawings for representing such networks. For instance, the set of equations equivalent to the counter is the following:

$$c = \text{if } r \text{ then } 0 \text{ else } s \quad ; \quad s = \bullet c + \text{if } x \text{ then } 1 \text{ else } 0$$

Note that such a set of equations has a straightforward solution as long as it does not contain combinational loops. In other words, any feed-back loop in the network should pass through a delay operator. In the following, we will take care to define only such well-founded data-flow networks.

At last, all synchronous formalisms are providing a notion of under-sampling, also called activation condition, or clock-enable in the domain of synchronous

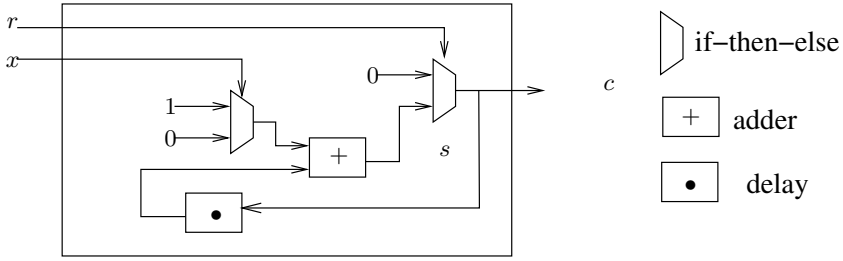


Fig. 1. The data-flow network of a synchronous counter

circuits. The activation condition is an higher-order operator that takes a synchronous node P , a Boolean input b , and produces a new node. The behavior of that node is defined as follows: whenever b is true, it behaves exactly like P , and whenever b is false, both the internal state and the outputs are “frozen” (i.e. they keep their previous value).

2.3 Modeling Asynchrony in the Synchronous Framework

The ability of the synchronous framework to model asynchrony is well-known [6], and has been often used [7,8,9,10,11]. In [1], we used a similar technique for translating a subset of AADL into synchronous data flow equations.

This goal is mainly achieved by using “oracles” (i.e., additional inputs) for modeling non-determinism, and activation conditions for modeling the asynchronous aspects: time-consuming tasks, multi-tasking, clock jitter.

However, in this previous work, multi-tasking was only considered in the case of simple fixed priorities rate monotonic scheduling. In this article, we consider more sophisticated policies that take into account shared resources with protected access, and all the problems they raise: priority inversion, and deadlock.

3 Handling Shared Resources

In AADL, **Data** component accesses can be shared between several components. In contrast to other kinds of components (thread, process, sub-program) which are translated into nodes, data components are translated into local variables of the surrounding component node. Depending on the kind of access that is associated with them (`read_only`, `write_only`, or `read_write`), the necessary wires are added to the interface of the node: a data component that has a write (resp., read) access to a resource has an additional output (resp., an additional input), and the data update is performed at its dispatch time using an activation condition.

In order to guarantee the data integrity, it is necessary to prevent the resource from being accessed by several components at the same time. For that purpose, several concurrency control protocols were defined [12], that modify the classical Rate Monotonic scheduling. In AADL, this is specified through the “`Concurrency_Control_Protocol`” property, attached to a data component. In this section, we explain how to implement four kinds of concurrency control protocol:

- **NoneSpecified**: components access the shared resource with no constraint at all (no lock mechanism).
- **Lock**: Before accessing a shared resource, a component should ask for it, and gets it only if no other component has locked it before; otherwise, it is suspended until the resource is unlocked. When a component obtains a resource, we say that the component enters a *critical section*.
Hence, a low priority thread tl can block a high priority one th if th wants to access a resource that is locked by tl . The problem with this protocol is that tl can block th , even when tl is not in critical section. This is referred to as *the priority inversion problem* [12].
- **BIP**: The Basic Inheritance Protocol, also known as Priority Inheritance Protocol, refines the previous one to prevent priority inversions.
- **PCP**: The Priority Ceiling Protocol is a refinement of BIP defined in order to prevent deadlock.

In the following, we describe those protocols more precisely, and explain how to implement them in terms of synchronous data-flow equations. Defining a scheduling protocol consists of defining a node, called hereafter a scheduler, that decides at each instant which thread the CPU is attributed to.

3.1 The *No Lock* Protocol

The simplest way of handling shared resources is to ignore them, and to always give the CPU to the highest priority thread. This (absence of) protocol is straightforward and generally useless for systems involving shared resources. But this simplest scheduler is refined in later sections for the other protocols. It is basically the scheduler used in [1].

Concretely, we need to generate a synchronous program that takes as inputs Boolean variables indicating which threads ask for the CPU ($Dispatched_1, \dots, Dispatched_n$), and that returns Boolean variables indicating which thread is elected ($cpu_1, \dots, cpu_n$). Of course, at most one among the cpu_i should be true at each instant. The program that computes the $Dispatched_i$ variables is derived from the period and the WCET of threads, which is specified in the AADL code.

The convention here is that t_i has priority over t_j if $i < j$. A possible way of implementing that node is as follows:

$$\forall k \in [1, n] : cpu_k = Dispatched_k \wedge \bigwedge_{0 < i < k} \overline{cpu_i} \quad (1)$$

Henceforth, the convention is that the program input variables begin with an uppercase letter (e.g., $Dispatched_k$); and $\overline{cpu_i}$ stands for the negation of cpu_i .

3.2 The *Blocking* Protocol

In order to take into account shared resources, we need additional inputs: the Boolean variable named $Asks\_cs_{r_\ell}^{t_i}$ indicates that the thread t_i wants to access the resource r_ℓ (their values come from the output of the predefined AADL sub-programs `Get_resource` and `Set_resource` [3]).

In order to ease the definition of cpu_k , we introduce the following auxiliary variables:

- the Boolean variable $has\_cs_{r_\ell}^{t_k}$ indicates that the thread t_k is in Critical Section on resource r_ℓ ;
- the Boolean variable $t_i\_blocks_{r_\ell}^{t_k}$ indicates that the thread t_k asks for a resource r_ℓ , which is locked by another thread t_i .

Computing which thread is in critical section. A thread t_k is in critical section for a resource r_ℓ if it asks for the resource, and if either

- it was in critical section before $(\bullet has\_cs_{r_\ell}^{t_k})$;²
- or it enters in critical section at the current instant. It enters a critical section when and only when it obtains the CPU.

Hence, the following definition of $has\_cs_{r_\ell}^{t_k}$:

$$\forall k \in [1, n], \forall \ell \in [1, m] : has\_cs_{r_\ell}^{t_k} = Asks\_cs_{r_\ell}^{t_k} \wedge (cpu_k \vee \bullet has\_cs_{r_\ell}^{t_k}) \quad (2)$$

Note that when we define such a relation, the quantification “ $\forall k \in [1, n], \forall \ell \in [1, m]$ ” suggests that we generate $n \times m$ equations for defining the scheduler. But in fact, it is generally much less, since in the AADL model, all threads may not have access connections to all resources. This remark actually holds for all the variables relating threads and resources in the following.

Computing the blocks relation. We say that a thread t_i blocks a thread t_k via a resource r_ℓ if both threads ask for the resource, and if the thread t_i was owning r_ℓ at the previous instant.

$$\forall k, i \in [1, n], i \neq k, \forall \ell \in [1, m] :$$

$$t_i\_blocks_{r_\ell}^{t_k} = Asks\_cs_{r_\ell}^{t_k} \wedge Asks\_cs_{r_\ell}^{t_i} \wedge \bullet has\_cs_{r_\ell}^{t_i} \quad (3)$$

Computing the elected thread. Once we have defined those two auxiliary relations, cpu_k can easily be defined similarly as in Section 3.1: the highest priority thread obtains the CPU, except if it is blocked by some other thread:

$$\forall k \in [1, n] : cpu_k = Dispatched_k \wedge \bigwedge_{0 < i < k} \overline{cpu_i} \wedge \bigwedge_{i \neq k, \ell \in [1, m]} \overline{t_i\_blocks_{r_\ell}^{t_k}} \quad (4)$$

Note that those three sets of equations defines a valid synchronous program, since they do not contain any combinational cycle (cf Section 2.2).

3.3 The Basic Inheritance Protocol

The Basic Inheritance Protocol was introduced [12] to avoid the *priority inversion problem*. Indeed, with the previous protocol, when a high-priority thread t_1 wants to access a resource shared by a lower priority thread t_3 , which have put a lock on it, then t_3 keeps the CPU. Moreover, t_3 can be interrupted by t_2 of lower priority than t_1 , even though t_2 does not try to access any shared resource.

The idea of the Basic Inheritance Protocol (BIP) is to modify the priority of t_3 in such a way that it inherits the priority of t_1 , when t_3 has the lock on a resource

² All Boolean delays $(\bullet)$ are implicitly initialized to false.

r_ℓ requested by t_1 . Indeed, this prevents t_2 to interrupt t_3 , and hence prevents the priority inversion.

The intuition of our BIP (synchronous data-flow) encoding is the following: first consider the dispatched thread with the highest priority. If it is not blocked, it must obtain the CPU. Otherwise, consider its blocking thread, and check if it is itself blocked, and so on until we find a thread that is not blocked. When we find the thread that is not blocked³, we give it the CPU. Hence, the first thing to do is to compute the transitive closure of the $t\_blocks_r^t$ relation.

Computing the $t_i\_blocks_{t_k}^*$ relation. Let an *inhibition path* from a thread t_i to a thread t_k be a set of $s + 1$ threads $\{t_i = t_{i0}, \dots, t_{is} = t_k\}$ such that there exist s resources $r_1, \dots, r_s$, that may be respectively accessed by t_{i0} and t_{i1} , t_{i1} and t_{i2} , $\dots$, t_{is-1} and t_{is} . Such a path is said to be *cycle-free* if all the threads in the path are distinct. Let $Path(i, k)$ be the set of cycle-free paths from t_i to t_k (this set can be computed from the AADL source code). A thread t_i *blocks*^{*} another thread t_k if t_i is not itself blocked, and if there exists an inhibition path in $Path(i, k)$ that is true.

$$\forall i, k \in [1, n], i \neq k : \quad (5)$$

$$t_i\_blocks_{t_k}^* = \overline{t_i\_is\_blocked} \wedge \bigvee_{p=\{i_0, \dots, i_s\} \in Path(i, k)} t_{i_0}\_blocks_{r_1}^{t_{i_1}} \wedge \dots \wedge t_{i_{s-1}}\_blocks_{r_s}^{t_{i_s}}$$

where:

$$\forall k \in [1, n] : t_k\_is\_blocked = \bigvee_{\ell \in [1, m], j \in [1, n], j \neq k} t_j\_blocks_{r_\ell}^{t_k}$$

The protocol. The BIP states that a thread in critical section on a resource *inherits* the priority of any other higher priority thread that asks for the same resource. The difficulty is to translate this “dynamic” condition⁴ into a Boolean condition. To do that we use an accumulator, (named *ii*, which stands for inhibiting index), that carries the value of the inhibitor of the thread that has the highest priority, if the dispatched thread with the highest priority is blocked (*ii* is set to 0 or -1 otherwise). For readability, we use a switch-like notation, where $c_1 \rightarrow x_1, c_2 \rightarrow x_2, \dots, c_n \rightarrow x_n$ stands for *if c_1 then x_1 else if c_2 then x_2 ... if c_n then x_n* .

$$ii_0 = 0$$

$$\forall k \in [1, n] : (cpu_k, ii_k) = \overline{dispatched_k} \rightarrow (False, ii_{k-1}) \quad (6)$$

$$(cpu_1 \vee \dots \vee cpu_{k-1}) \rightarrow (False, -1) \quad (7)$$

$$ii_{k-1} = k \rightarrow (True, -1) \quad (8)$$

$$ii_{k-1} > 0 \rightarrow (False, ii_{k-1}) \quad (9)$$

$$\{ t_j\_blocks_{t_k}^* \rightarrow (False, j) \}_{j \in [1, n], j \neq k} \quad (10)$$

$$True \rightarrow \overline{(t_k\_is\_blocked, 0)} \quad (11)$$

³ if such a thread does not exist, the model-checker will tell us (cf Section 4.1).

⁴ The priority of each thread depends on the history. But by chance, it only depends on a very short history, that is, the previous instant.

For each $k > 0$, cpu_k and ii_k depend on cpu_{k-1} and ii_{k-1} , which means that cpu_1 and ii_1 are computed first, and then cpu_2 and ii_2 , and so on, until cpu_n and ii_n . At the beginning, the inhibiting index is equal to 0 ($ii_0 = 0$). Then, the pairs (cpu_1, ii_1) , ..., (cpu_n, ii_n) are computed in turn. As long as cpu_{k-1} is set to *False* (i.e., when conditions of lines 8 and 11 do not hold):

- If t_k is blocked by a lower priority thread t_j (line 10), the inhibiting index takes the priority of the inhibitor j . Then, the inhibiting index keeps this value (lines 6 and 9), until the index of the inhibitor is reached (line 8). In that case, the corresponding *cpu* variable is set to *True*, the remaining values of *cpu* are set to *False* (line 7), and ii_k is unused for bigger k (-1).
- Otherwise (line 11), if t_k is not blocked at all, it gets the CPU, and all the remaining values of *cpu* are set to false (line 7). If it is blocked, the system deadlocks.

3.4 The *Priority Ceiling Protocol*

The problem with the BIP is that it does not prevent deadlocks. Indeed, consider the following scenario, where 2 threads t_1 and t_2 share 2 resources r_1 and r_2 :

1. t_2 asks for the CPU (*Dispatched*₁) and gets it.
2. t_2 locks r_1 .
3. t_1 asks for the CPU. It has a higher priority than t_2 , hence t_1 gets the CPU.
4. t_1 locks r_2 .
5. t_1 tries to lock r_1 . But t_2 has locked it. Therefore t_2 gets the CPU.
6. t_2 tries to lock r_2 . But t_1 has locked it. Nobody can get the CPU. The system is blocked (or deadlocks).

One solution is to (statically) forbid such intertwined use of locks. Another solution is to use the so-called Priority Ceiling Protocol (PCP). The PCP is a refinement of the BIP.

The *priority ceiling of a resource* r_ℓ is the maximal priority of all the threads that may use r_ℓ ; we note it $PC(\ell)$. The *priority ceiling of a thread* t_k is the maximum of the priority ceilings of the resources locked by other threads; we note it PC_k . Contrary to $PC(\ell)$, PC_k is a dynamic value. The PCP consists in adding the following constraint to the BIP: t_k can lock a resource r only if its priority is higher than its priority ceiling ($k < PC_k$).

The Priority Ceiling of resources locked by threads other than k . PC_k formal definition is just a direct translation of the definition given above.

$$\forall k \in [1, n] : PC_k = \quad (12)$$

$$\text{Min } \{n + 1\} \cup \left\{ PC(\ell) / \text{Asks\_cs}_{r_\ell}^{t_i} \wedge \bullet \text{has\_cs}_{r_\ell}^{t_i} \right\} \ell \in [1, m], i \in [1, n], i \neq k$$

The tk_ask relation. We first define yet another auxiliary relation that states whether a thread wants to enter a critical section at the current instant (i.e., a thread asks for a resource that it hasn't locked yet).

$$\forall k \in [1, n] : \text{asks\_cs}^{t_k} = \bigvee_{\ell \in [1, m]} (\text{Asks\_cs}_{r_\ell}^{t_k} \wedge \bullet \text{has\_cs}_{r_\ell}^{t_k})$$

The protocol. The PCP encoding is the same as the BIP one, except that we modify the definition of the *blocks* relation (previously defined in equation 3). Indeed, there is now a second reason for a thread t_k to be blocked by another thread t_i : if t_k wants to enter a critical section ($asks\_cs^{t_k}$) when its priority ceiling PC_k is not higher than its own priority ($PC_k \leq k$). Note that the priority ceiling of t_k (i.e., the value of PC_k) is a consequence of the lock that t_i has on the resource ℓ ($PC(\ell) = PC_k$).

$$\forall k, i \in [1, n], i \neq k, \forall \ell \in [1, m] : t_i \_blocks_{r_\ell}^{t_k} = \text{Asks\_cs}_{r_\ell}^{t_i} \wedge \bullet has\_cs_{r_\ell}^{t_i} \wedge (\text{Asks\_cs}_{r_\ell}^{t_k} \vee (\text{asks\_cs}^{t_k} \wedge PC(\ell) = PC_k \leq k)) \quad (13)$$

Here again, those set of equations defines a valid synchronous program as they do not contain any combinational loop.

4 Validation

We have encoded all the equations given in the previous Section into an OCAML (meta-)program that, given a set of tasks, a set of resources, and a set of task/resource pairs, generates a LUSTRE program⁵. The resulting LUSTRE program is a task scheduler, computing one Boolean variable (cpu_i) per thread (t_i), from Boolean inputs indicating which threads ask for the CPU ($Dispatched_{t_i}$), and which threads ask for which resource ($Asks\_cs_{r_i}^{t_i}$).

In the following, we illustrate the use of a state-explorer (i.e., a model-checker) to prove various properties of this generated program. This was very useful to debug the equations given in this paper, and also to debug the OCAML encoding of those equations. We'll also argue why we believe it might also be useful for AADL end-users.

4.1 Absence of Deadlock

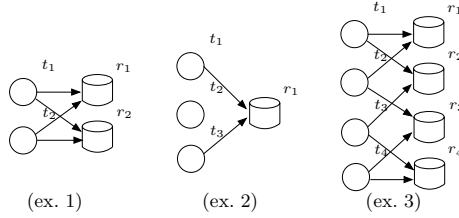
In order to prove the absence of deadlock, we used LESAR [13], a LUSTRE model-checker. This tool implements state-of-the-art state-reachability algorithms, based on Binary Decision Diagrams. We used both an enumerative algorithm which complexity is related to the number of states, and a symbolic algorithm, which complexity is related to the diameter of the state space.

We proved with LESAR that, whenever at least one thread asks for the CPU, at least one of the cpu_i is true. Actually, we even prove that exactly one cpu_i is true in that case, which simply proves that our scheduler is correct in the sense that it does not give the processor to more than one thread:

$$(\bigvee_{i \in [1, n]} Dispatched_i) \Rightarrow \bigvee_{i \in [1, n]} cpu_i$$

We performed this on the examples of Fig. 2. For instance, the first example (ex. 1) of Fig. 2 consists of a system with two threads $t1$ and $t2$, that can

⁵ We put a copy of this OCAML program as well as a copy of the resulting LUSTRE programs at the url <http://www-verimag.imag.fr/~jahier/aadl-schedul/>

**Fig. 2.** Examples of tasks accessing shared resources

access two resources r_1 and r_2 . This example is precisely the one given in [12] to illustrate the fact that the BIP does not prevent deadlock, and which motivates the definition of PCP.

LESAR was indeed able to generate a counter-example that exhibits a deadlock; the scenario it provides is almost the same as the one given [12] (and also in Section 3.4). LESAR proved the absence of deadlock for the PCP on the three examples. The results of those experiments are outlined in Fig. 3. When the property is false, we indicate the length of the counter-example. When the property is true, we indicate the diameter of the graph, and its number of states. All runs lasted less that a second.

An interesting point in those experiments is that it is not always worth using the PCP (that is deadlock-free by construction) since the BIP and the lock protocol can provably be deadlock-free in some configurations (e.g., in ex. 2). Note that in order to avoid false alarms, we need to tell the state-explorer that the inputs of the scheduler are not completely random. For instance, it was necessary to assert that a thread cannot change its requests for resources when it does not own the CPU.

	Lock	BIP	PCP
ex. 1	ko: 5	ko: 5	ok: 5/40
ex. 2	ok: 6/96	ok: 7/96	ok: 7/96
ex. 3	ko: 9	ko: 9	ok: 12/2316

Fig. 3. Deadlock property exp.

Lock	BIP	PCP
ok: 6/46	ok: 6/46	ok: 5/40
ko: 4	ok: 7/96	ok: 7/96
ko: 4	ok: 10/3708	ok: 12/3216

Fig. 4. Priority-inversion property exp

4.2 Priority Inversion

The priority inversion corresponds to situations when a thread is blocked by a lower priority thread. This occurs very naturally when two threads share the same resource, locked by the lower priority thread. Priority inversion is more problematic when it happens as in the example of Section 3.3 (which was the example given in [12] to motivate the introduction of the BIP). Indeed, threads are generally supposed to remain in critical section for a short time. Now, if a thread that does not lock any resource preempts a lower priority thread in critical section, the corresponding resource might be locked for a long time.

Therefore, we check the following property: if a thread t_k gets the CPU, when a higher priority thread asks to enter in critical section, then t_k should have

at least a lock on one of the resource. In other words, we want to be sure that a thread that does not lock any resource cannot block any higher priority thread:

$$\forall i \in [2, n], \forall j \in [1, i - 1] : (cpu_i \wedge asks\_cs^{t_j}) \Rightarrow \bigvee_{\ell \in [1, m]} has\_cs_{r_\ell}^{t_i}$$

We actually ask the model-checker to prove a slightly higher refined property, which is that for any system that does not deadlock, there is no priority inversion. Indeed, as soon as two tasks deadlock, any other thread can get the CPU even if it is not supposed to, according to the priorities defined by the protocol. This is the kind of subtlety that can be discovered using a model-checker.

As summarized in Fig. 4, LESAR found counter-examples that falsify the property for the last two examples of Fig. 2 using the lock protocol. And it proved the property with the BIP and the PCP. The second example is the one given [12] (and in Section 3.3) for motivating the introduction of the BIP.

4.3 Schedulability

The thread scheduler we generate in Section 3 is just a part of the AADL2LUSTRE translator [1]. The program that computes the values of the *Dispatched_i* variables is derived from the AADL code (from the threads period and WCET).

In order to check the schedulability of an AADL system, we look at the sequences of values taken by the *Dispatched_i* and *cpu_i* variables. The set of valid sequences is defined by the automaton of Fig. 5. In this automaton, *d* stands for “dispatch”, and is defined as the *Dispatched* rising edge; *a* stands for “activate”, and is defined as the *cpu* rising edge; and *r* stands for “release”, and is defined as the *Dispatched* falling edge. All omitted transitions in this automaton target the “scheduling-error” state. A system is well-scheduled if this error state is never reached. Indeed, nothing prevents the generated scheduler to issue a “dispatch” event between an “activate” and a “release” event. This is what occurs when the system is not schedulable, i.e., when some deadline is missed. Once encoded into a LUSTRE formula, this automaton can be used to prove (by state-exploration) that the system is schedulable.

Note that this schedulability property somehow does not only concern the part of the AADL2LUSTRE translator described in this article. But we mention

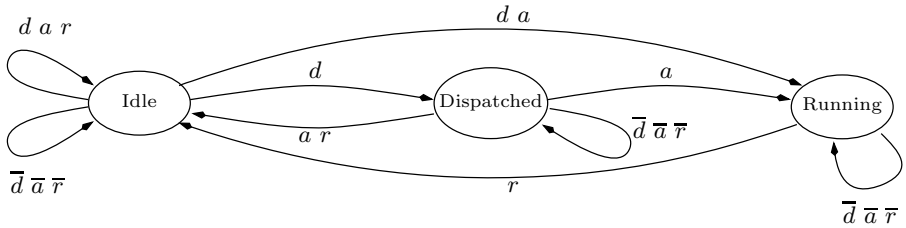


Fig. 5. The automaton recognizing well-scheduled systems. There is one such automaton per thread to schedule.

it here because we believe this analysis is particularly interesting in presence of shared resources.

5 Related Work

The Cheddar tool [14,15] can perform Schedulability analysis over AADL specification, but it ignores the functional aspects of AADL components, and it is more oriented towards quantitative analysis: resource usage, number of preemptions, number of context switches, etc. Cheddar allows users to define dedicated (user defined) schedulers and perform simulations [16].

Using a synchronous framework to model software architectures is not a new idea [9,10,8,11]. Gamatié et al. [9,10] defined a framework that provides a library of components, written in SIGNAL [17] and C++, suitable for modeling systems following the ARINC (Aeronautical Radio Incorporated) 653 standard. They demonstrate how to use the SIGNAL language as an ADL – whereas we translate AADL architecture models into LUSTRE. They mention that model-checking could be possible since the system is described in SIGNAL, but the task scheduler is implemented in C++, which would make its model-checking difficult – they do not pretend to be able to check the scheduler though. Anyway, they do not mention any particular protocol with respect to shared resource handling.

Formal verification of priority inheritance protocols has also been conducted using the PVS theorem prover [18]. The kind of outcome that one obtains using a theorem prover is of course different of what can be achieved with a model-checker. With PVS, Dutertre proves very general property about the PCP correctness. On the contrary, we model-check the protocol together with the system architecture description, plus the functional components. We are therefore able to prove much more fine-grained properties, not only about the whole system behavior, but also about the scheduling protocol itself. Moreover, some protocol properties can be false in the general case; for example, we proved that the second system of Fig. 2 does not deadlock with the BIP.

Penix et al. used a model-checker to verify a rate monotonic scheduler of a real time operating system [19]. But as their scheduler model is very detailed, here again the rest of the architecture is kept abstract. Elaborated protocols for dealing with shared resources are not addressed either.

6 Conclusion

Defining an automated translation from AADL models to a purely Boolean synchronous scheduler, that can be directly model-checked, has many advantages.

- Firstly, the model-checker was very useful to debug our scheduler generator.
- Secondly, we claim it can also be useful for the AADL end-users; for example, the PCP is a refinement of the BIP that has been introduced to avoid deadlocks. However, for some particular topologies of threads and resources, it may happen that deadlocks cannot occur even with the BIP scheduler, and that a model-checker is able to prove it on our model.

- Finally, in presence of shared resources, the analytic schedulability criteria may be too conservative, and reject schedulable systems. Moreover, as soon as the system contains sporadic events (when the thread activation depends on the output of some other thread), the analytic method can be meaningless. Consider for example two components activated by a third one, which both outputs cannot be true at the same instant.

Of course with our technique, one cannot deal with generic properties (i.e., for any number of tasks and resources), but since the generation of models for verification is automatic, the verification can be replayed for each instance.

When the verification problem is too large, an exhaustive verification can be untractable. However, our encoding can still be useful to perform intensive automatic simulations using testing tools like Lurette [20]. The absence of deadlocks, the schedulability, and the non-inversion properties are used as test oracles (i.e., runtime monitors). The assertions on the scheduler inputs (e.g., no rising edges for the asking of a resource by threads that do not have the CPU) are used to constrain the random input generator [21].

Another case where such tests and simulations are the only tractable methods is when the AADL model contains sporadic threads activated by software components that are not implemented in Lustre (or in any other language with formal semantics). A way around this problem would be to have a Lustre abstraction of all the possible behavior of such components; but such an abstraction is not always easy to define.

Acknowledgments

We thank Karine Altisen for fruitful (nearby office) discussions about scheduling.

References

1. Halbwachs, N., Jahier, E., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Seventh International Conference on Embedded Software (EMSOFT 2007), Salzburg, Austria (October 2007)
2. Feiler, P.H., Gluch, D.P., Hudak, J.J., Lewis, B.A.: Embedded system architecture analysis using SAE AADL. Technical note cmu/sei-2004-tn-005, Carnegie Mellon University (2004)
3. SAE: Architecture Analysis & Design Language (AADL). AS5506, Version 1.0, SAE Aerospace (November 2004)
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE* 79(9), 1305–1320 (1991)
5. Liu, C.L., Layland, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM* 20(1), 46–61 (1973)
6. Milner, R.: On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edinburgh Univ. (1981)
7. Baufreton, P.: SACRES: A step ahead in the development of critical avionics applications. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) *HSCC 1999*. LNCS, vol. 1569, p. 1. Springer, Heidelberg (1999)

8. Baufreton, P.: Visual notations based on synchronous languages for dynamic validation of gals systems. In: CCCT 2004 Computing, Communications and Control Technologies, Austin (Texas) (August 2004)
9. Gamatié, A., Gautier, T.: Synchronous modeling of avionics applications using the signal language. In: 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), Toronto, pp. 144–151 (May 2003)
10. Gamatié, A., Gautier, T.: The signal approach to the design of system architectures. In: 10th IEEE Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2003), Huntsville (Alabama), pp. 80–88 (April 2003)
11. Le Guernic, P., Talpin, J.P., Le Lann, J.C.: Polychrony for system design. *Journal for Circuits, Systems and Computers*, Special Issue on Application Specific Hardware Design (April 2003)
12. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers* 39(9), 1175–1185 (1990)
13. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering*, Special Issue on the Specification and Analysis of Real-Time Systems, 785–793 (September 1992)
14. Hugues, J., Zalila, B., Kordon, L.P.F.: Rapid prototyping of distributed real-time embedded systems using the AADL and Ocarina. In: 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP 2007) (2007)
15. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar: a flexible real time scheduling framework. In: McCormick, J.W., Sward, R.E. (eds.) *SIGAda*, pp. 1–8. ACM, New York (2004)
16. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and memory requirements analysis with AADL. In: *SIGAda* (2005)
17. Guernic, P.L., Benveniste, A., Bournai, P., Gautier, T.: SIGNAL, a data flow oriented language for signal processing. *IEEE-ASSP* 34(2), 362–374 (1986)
18. Dutertre, B.: Formal analysis of the priority ceiling protocol. In: *IEEE Real-Time Systems Symposium (RTSS 2000)*, pp. 151–160 (2000)
19. Penix, J., Visser, W., Engstrom, E., Larson, A., Weininger, N.: Verification of time partitioning in the deos scheduler kernel. In: *ICSE*, pp. 488–497 (2000)
20. Jahier, E., Raymond, P., Baufreton, P.: Case studies with Lurette V2. *International Journal on Software Tools for Technology Transfer (STTT) Special Section on Leveraging Applications of Formal Methods* (2006)
21. Raymond, P., Jahier, E., Roux, Y.: Describing and executing random reactive systems. In: *SEFM*, pp. 216–225. IEEE Computer Society, Los Alamitos (2006)

Describing and Analyzing Behaviours over Tabular Specifications Using (Dyn)Alloy

Nazareno M. Aguirre¹, Marcelo F. Frias², Mariano M. Moscato²,
Thomas S.E. Maibaum³, and Alan Wassyng³

¹ Department of Computer Science, FCEFQyN, Universidad Nacional de Rio Cuarto
and CONICET, Argentina

`naguirre@dc.exa.unrc.edu.ar`

² Department of Computer Science, FCEyN, Universidad de Buenos Aires and
CONICET, Argentina

`{mfrias,mmoscato}@dc.uba.ar`

³ Department of Computing and Software, McMaster University, Canada

`tom@maibaum.org, wassyng@mcmaster.ca`

Abstract. We propose complementing tabular notations used in requirements specifications, such as those used in the SCR method, with a formalism for describing specific, useful, subclasses of *computations*, i.e., particular combinations of the atomic transitions specified within tables. This provides the specifier with the ability of *driving* the execution of transitions specified by tables, without the onerous burden of having to introduce modifications into the tabular expressions; thus, it avoids the problem of modifying the object of analysis, which would make the analysis indirect and potentially confusing. This is useful for a number of activities, such as defining test harnesses for tables, and concentrating the analyses on particular, interesting, subsets of computations. Unlike previous approaches, ours allows for the description of a wider class of combinations of the transitions defined by tables, by means of a rich operational language. This language is an extension of the Alloy language, called DynAlloy, whose notation is inspired by that of dynamic logic.

The use of DynAlloy enables us to provide an extra mechanism for the analysis of tabular specifications, based on SAT solving. We will illustrate this and the features of our approach via an example based on a known tabular specification of a simple autopilot system.

1 Introduction

Tabular notations, originally used to document requirements by D. Parnas and others [9], have proved to be a useful means for concisely describing expressions characterizing complex requirements. Indeed, tables have been successfully incorporated into various formalisms for requirements specification, most notably those reported in [12,7]. The central use of tables in the description of software requirements is as a way of organizing formulas that specify the relations that the system must maintain with the environment. Since these formulas would be large and complex in their traditionally linear notation, their division into well

distinguished smaller formulas that are easier to follow, provided by the tabular notation, has great advantages. A tabular specification then consists of a collection of tables, which combined specify a relation R , characterizing the intended behaviour of the system. There exist different classes of tables, but essentially all are descriptions of relations of some form in terms of *guards* and *result values*. The whole specified system is then typically composed of a disjunction of these guarded expressions, describing, intuitively, all transitions.

We are concerned with complementing tabular notations with a way of prescribing specific combinations of transitions defined in tabular descriptions. As we will argue later on, this enables the specifier to *drive* the execution of transitions defined by tables, which is useful for defining test harnesses for tables, and concentrating the analysis activities on particular execution scenarios, namely those corresponding to the prescribed combinations.

Contributions of this paper. The contributions of this paper are twofold. First, we propose a notation for prescribing subsets of the set of all possible executions of a tabular specification. The notation is very expressive, based on an operational language called DynAlloy [4], an extension of the Alloy specification language [10]. This has as an advantage that the specifier can concentrate the analyses on the particular sets of runs he is interested in with a potentially great impact on the efficiency and effectiveness of the analyses. The proposed notation enables the specifier to describe sets of executions by means of *programs* referring to the tabular descriptions, without the need to introduce modifications in the tables. These descriptions are written in a language accessible to the specifier familiar with tabular descriptions.

Second, we provide an additional analysis mechanism for tabular specifications, based on SAT solving, and supporting the above mentioned notation for prescribing sets of executions. This analysis mechanism is based on a translation of the tabular specifications into Alloy and DynAlloy, and the use of the Alloy and DynAlloy Analyzers for performing SAT based analysis.

Related Work. The described tabular notations, and in particular the tabular expressions used in the SCR (Software Cost Reduction) method [7], have associated tool support, which provide different kinds of analysis, ranging from simple syntax checking to theorem proving and model checking of properties [3,8,13]. However, most analyses we are aware of apply to the whole set of behaviours associated with tables; more precisely, most techniques for analyzing properties of behaviours are concerned with the *global* set of “atomic transitions”, described in the tables. Generally, there is a lack of a notation for describing particular combinations of these atomic transitions or tables. An exception to this is the case of the simulator in the SCR toolset [8]. The simulator allows the developer to load specific scenarios and check whether certain associated assertions are violated or not in the particular executions described by the scenarios. Also, in an approach described in [5] and defined for testing purposes in the context of SCR [7], *modes* (essentially classes of states) are exploited as a means for singling out a proper subset of all possible transition sequences allowed by a tabular

specification. These approaches have some limitations. The use of the first alternative becomes impractical if the set of execution scenarios one is interested in is large, since each execution scenario needs to be individually described. On the other hand, the second alternative allows for the description (and analysis) of large proper subsets of the set of executions associated with a tabular description, but all these executions are similarly obtained, essentially by considering all the execution sequences that “go through” (or, more precisely, “end up at”) a given set of modes. This is insufficient if one is interested in more sophisticated execution sequences, resulting from table sequencings not obtainable by “filtering” executions according to some of the existing modes. Of course, one might decide to include new modes and mode classes to enforce these particular sequencings of the transitions modelled by tables; however, this last alternative is, in our opinion, unsuitable, since it would require altering the tables and introducing new modes and mode classes to enforce the sequencing; clearly, this is inappropriate, and potentially dangerous, if the sequencing is not really part of the requirements, but particular behaviours the modeller wants to analyze.

2 An Example of Tabular Specifications

In order to illustrate how transitions are typically specified by tables, we will use the SCR approach to requirements specification. We will describe the notation via an SCR specification, given in [2], of the requirements for a simple autopilot system. This will also serve us as a case study for illustrating our proposal.

In the SCR methodology, tables are used for describing the relationship that the system should induce between monitored and controlled variables. In order to describe this relationship, SCR uses *events*, *conditions*, *mode classes* and *terms*. Events occur when changes in the variables observed by the system take place (these variables include monitored and controlled variables, as well as modes and terms), and conditions are logical expressions referring to these variables. Modes represent classes of states of the system (whose corresponding partition is called a mode class), and terms are functions on the variables of the specification.

The autopilot needs to monitor three environment variables, the aircraft’s altitude, flight path angle and calibrated air speed, represented by monitored variables `mALTactual`, `mFPAactual` and `mCASactual`, respectively. It also monitors the status of some elements in the autopilot’s control panel, which are four switches, represented also by monitored variables `mALTsw`, `mATTsw`, `mCASsw` and `mFPAsw`, and three knobs for changing the desired altitude, flight path angle and calibrated air speed (these values are represented by monitored variables `mALTdesired`, `mFPAdesired` and `mCASdesired`, respectively). The system has to control three displays, which show either the actual or desired altitude, flight path angle and calibrated air speed (displays are represented by controlled variables `cALTdisplay`, `cFPAdisplay` and `cCASdisplay`), depending on the state of the system. The four switches allow the pilot to activate the modes `ATTmode`, `ALTmode`, `FPAmode` and `CASmode` of the system. The displays usually show the current altitude, flight path angle and calibrated air speed, unless the pilot changes

one of these desired values (i.e., “preselects” a value) and activates the corresponding mode. In this case, the display will show the desired value instead of the actual one. Each display will show the corresponding current value (instead of the desired value) when the corresponding mode is manually disengaged, or when the desired value is reached. Modes are engaged/disengaged by setting the corresponding switches, although the system cannot be engaged in more than one of the modes **ALTmode** and **FPAmode**, so entering one of these should disengage the previous mode. There is an extra mode, the attitude control wheel steering, in which the system is set when neither **ALTmode** nor **FPAmode** are engaged. The **CASmode** can be engaged independently of the other modes, at any time.

When the pilot attempts to engage the system into the **ALTmode**, setting the desired altitude to one that is more than 1200 feet above the current altitude, the system will not engage directly in the **ALTmode**; in this situation, the system will switch to the mode **FPAmode**, in an “armed” mode. Then the system will require the pilot to enter a flight path angle (that to follow until the aircraft gets within 1200 feet away from the desired altitude), after which the system will move to an “unarmed” mode. Once the aircraft reaches the point where it is less than 1200 feet from the desired altitude, it engages the mode **ALTmode**. When a mode other than **CASmode** is engaged, the other preselected displays return to show the current value (instead of the desired one).

In the formalization of the autopilot system’s requirements we are reproducing here, the **FPAmode** is splitted into two different modes, **FPAarmed** and **FPAunarmed**, to differentiate the cases in which the **FPAmode** is *armed* (waiting for the flight path angle to be set after the **mALTsw** was switched on at an altitude lower than 1200 feet below the desired altitude) and the case in which the **FPAmode** is unarmed. These two modes together with modes **ALTmode** and **ATTmode** constitute the only mode class of the system, called **mcStatus**. Also, terms **tALTpresel**, **tCASpresel** and **tFPAPresel** are introduced to characterize the states in which the desired altitude, calibrated air speed and flight path angle have been preselected, and the **CASmode** is also represented as a term. An additional term **tNear** is used to characterize the states in which the difference between the desired and actual altitudes is smaller than 1200.

For describing events, SCR provides a simple notation. The notation **@T(c)** **WHEN d** describes the event in which expression **c** becomes true, when **d** is **true** in the current state, i.e., it represents the expression $c' \wedge c \wedge d$, where the primed expression refers to the next state. If **d** is true, the ‘**WHEN**’ section is not written. Also, the event **CHANGED(v)** indicates that *v* has changed, i.e., it represents the expression $v' \neq v$. The table describing the mode transitions, as well as the values for terms **tALTpresel** and **tFPAPresel** is the first one in Figure 1. This table corresponds to the merge of a mode transition table, describing the mode transitions, and two event tables. The values of the displays are defined by the next three small condition tables in Fig. 1. Finally, terms **tCASmode** and **tCASpresel** get defined by the bottom event table in Fig. 1. Term **tNear** has a definition which is independent of modes, conditions and events; its definition is simply $mALTdesired - mALTactual \leq 1200$.

Mode Class = mcStatus				
Old mode	Events	New mode	tALTpresel	tFPAPresel
ATTmode	@T(mALTsw=on) WHEN (tALTpresel AND tNear) @T(mALTsw=on) WHEN (tALTpresel AND NOT tNear) @T(mFPAsw=on) CHANGED(mALTdesired) CHANGED(mFPAdesired)	ALTmode FPAarmed FPAunarmed	false true	false true
ALTmode	@T(mATTsw=on) @T(mFPAsw=on) CHANGED(mALTdesired) CHANGED(mFPAdesired) @T(mALTdesired = mALTactual)	ATTmode FPAunarmed ATTmode	false false false	false false true
FPAarmed	@T(mATTsw=on) OR @T(mFPAsw=on) CHANGED(mALTdesired) CHANGED(mFPAdesired) AND NOT (mFPAdesired' = mFPAactual') @T(tNear) AND (mALTdesired' = mALTdesired) @T(mFPAdesired = mFPAactual)	ATTmode ATTmode ALTmode	false false	false false true false false
FPAunarmed	@T(mALTsw=on) WHEN (tALTpresel AND tNear) @T(mALTsw=on) WHEN (tALTpresel AND NOT tNear) @T(mATTsw=on) OR @T(mFPAsw=on) CHANGED(mALTdesired) CHANGED(mFPAdesired) AND NOT (mFPAdesired' = mFPAactual') @T(mFPAdesired = mFPAactual)	ALTmode FPAarmed ATTmode	false true	false false true false

cALTdisplay =	Conditions
	tALTpresel NOT tALTpresel mALTdesired mALTactual
cCASdisplay =	Conditions
	tCASpresel NOT tCASpresel mCASdesired mCASactual
cFPAdisplay =	Conditions
	tFPAPresel NOT tFPAPresel mFPAdesired mFPAactual

Term = tCASmode		
	Events	tCASmode tCASpresel
NOT tCASmode	@T(mCASsw=on) CHANGED(mCASdesired)	true true
tCASmode	@T(mCASsw=on) @T(mCASdesired = mCASactual) CHANGED(mCASdesired) AND NOT (mCASdesired' = mCASactual')	false false false true

Fig. 1. Tabular specification of the autopilot system

3 The Alloy and DynAlloy Modeling Languages

In the description of the autopilot system, the datatypes associated with the variables are obvious: numeric values range over integers, states for switches can be characterized by boolean values, and the possible values for the mode class `mcStatus` can be the defined modes for the system. In Alloy, these datatypes would be defined by *signatures*. For instance, the datatype associated with mode class `mcStatus` can be straightforwardly defined in Alloy in the following way:

```
abstract sig StatusMode { }
one sig ALTmode, ATTmode, FPAmode extends StatusMode { }
```

Abstract signatures have as their only associated elements, those of their non abstract “subsignatures”. The modifier `one` forces the corresponding signatures to have exactly one element, i.e., to be singletons. Thus, the above Alloy specification defines an enumerated set. More complex data domains can also be defined via signatures, using typed *fields*. For instance, we can define a signature to characterize the *state* associated with the autopilot system, composed of monitored and controlled variables, mode classes and state dependent terms, in the following way:

```

sig State {
  -- Monitored Variables
  mALTactual, mCASactual, mFPAactual, mALTdesired, mCASdesired,
  mFPAdesired: Int,
  mALTsw, mATTsw, mCASsw, mFPAsw: SwitchState,
  -- Controlled variables
  cALTdisplay, cCASdisplay, cFPAdisplay: Int,
  -- Mode classes
  mcStatus: StatusMode,
  -- Terms
  tARMED, tCASmode, tALTpresel, tCASpresel, tFPApresel: Boolean
}

```

Fields, which can have relational types, are interpreted as relations from the set associated with the signature in which the field is defined to the relation given as a type of the field. Thus, for instance, field `mcStatus` in signature `State` is interpreted as a relation from `State` to `StatusMode`.

Using signatures and fields, it is possible to build more complex expressions denoting relations, with the aid of the Alloy operators. Operator $\sim$ denotes relational transposition, $*$ denotes reflexive-transitive closure, and $\wedge$ denotes transitive closure of a binary relation. Operator $+$ denotes union, $\&$ denotes intersection, and dot ($\cdot$) denotes composition of relations, generalized to n -ary relations and having relational image as a special case. In all cases, the typing must be adequate. Formulas are built from expressions. Binary predicate `in` checks for inclusion, while `=` checks for equality. From these (atomic) formulas we define more complex formulas using standard first-order connectives and quantifiers. Negation is denoted by `!`. Conjunction, disjunction and implication are denoted by `&&`, `||` and `=>`, respectively. Finally, quantifications have the form `some a : A |  $\alpha(a)$` and `all a : A |  $\alpha(a)$` . These formulas can be used in order to describe assumed as well as intended properties of the models. Parameterized formulas, which can be used for describing properties, can be written in Alloy using *predicates*. For instance, we can define a predicate for characterizing event `@T(mATTsw = on)`, as follows:

```

pred Ev_TmALTswOn(s,s': State) { s.mALTsw != on && s'.mALTsw = on }

```

Assumed properties of the specified data domains can be given as *facts* in Alloy. We can use facts for characterizing the values of terms or other variables defined via condition tables in a straightforward way. For our presented example, the values associated with controlled variables `cALTdisplay`, `cCASdisplay` and `cFPAdisplay` can be enforced using a fact, in the following way:

```

fact {
  all s: State | (s.tALTpresel = trueValue =>
    s.cALTdisplay = s.mALTdesired else
    s.cALTdisplay = s.mALT) &&
    (s.tCASpresel = trueValue =>
    s.cCASdisplay = s.mCASdesired else
    s.cCASdisplay = s.mCASactual) &&

```



```

(s.tFPAPresel = trueValue =>
    s.cFPADisplay = s.mFPAdesired else
    s.cFPADisplay = s.mFPAactual)
}

```

Intended properties, those to be checked, are defined in Alloy using *assertions*. For instance, we can consider the following assertion, corresponding to a disjointness check for the mode transition table associated with `mcStatus`:

```

assert DisjointnessCheck {
all s, s': State | ! (s.mcStatus = ATTmode &&
(Ev_TmALTswOn[s,s'] && s.tALTpresel = trueValue && s.tNear = trueValue) &&
(Ev_TmALTswOn[s,s'] && s.tALTpresel = trueValue && !(s.tNear = trueValue)))
}

```

In Alloy, operations over the defined domains are specified using *predicates*. DynAlloy, on the other hand, incorporates the notion of *action* for specifying operations. Atomic actions, which are the basic units for specifying state change, are defined via pre- and post-conditions. This kind of description for an action indicates that, for the action to be executed, its precondition must be true, and in this case the state resulting from the execution of the action satisfies the postcondition. As a simple example, consider the following action, which characterizes the change of the monitored variable `mALTactual` (and the arbitrary change of the event-dependent terms and controlled variables):

```

act mALTactualChange[s: State] {
pre { }
post { s'.mALTactual != s.mALTactual && s'.mCASactual = s.mCASactual &&
s'.mFPAactual = s.mFPAactual && s'.mALTsw = s.mALTsw &&
s'.mATTsw = s.mATTsw && s'.mCASsw = s.mCASsw &&
s'.mFPAsw = s.mFPAsw && s'.mALTdesired = s.mALTdesired &&
s'.mCASdesired = s.mCASdesired && s'.mFPAdesired = s.FPAdesired }
}

```

Atomic actions can be composed to form *composite* actions (also called programs). These are built using sequential composition (`;`), nondeterministic choice (`+`), test (`[f]?`, an action that does not modify the state but can only be executed when `f` is true) and iteration (`*`). For example, if we consider atomic actions describing the change of each of the monitored variables (as we did above for monitored variable `mALTactual`), then the following composite action characterizes the change of *one* of the monitored variables:

```

program monitoredVarChange[s: State] {
mALTactualChange[s] + mCASactualChange[s] + mFPAactualChange[s] +
mALTswChange[s] + mATTswChange[s] + mCASswChange[s] + mFPAswChange[s] +
mALTdesiredChange[s] + mCASdesiredChange[s] + mFPAdesiredChange[s]
}

```

This language for describing composite actions is what we are primarily interested in exploiting. DynAlloy also allows the specifier to write *assertions* associated with his programs, i.e., intended properties of the executions of the

programs, to be checked. These properties to be checked are also given in the form of partial correctness assertions, i.e., by pre- and post-conditions. For example, the following DynAlloy assertion expresses that the above program cannot change `mALTactual` and `mCASactual` at the same time:

```
assertCorrectness[s:State] {
pre { }
program monitoredVarChange[s]
post { !(s'.mALTactual != s.mALTactual && s'.mCASactual != s.mCASactual)}
}
```

Alloy assertions can be automatically analyzed using the Alloy Analyzer. The mechanism for analysis is based on SAT solving. Basically, given a system specification and a statement about it, the Alloy tool exhaustively searches for a counterexample of this statement under the assumptions of the system description, by reducing the problem to the satisfiability of a propositional formula. Since the Alloy language is first-order, the search for counterexamples has to be performed up to a certain bound k in the number of elements in the universe of the interpretations. Thus, in order to check an assertion, the user has to provide bounds for the number of elements in the domains (associated with signatures) of the specification. Obviously, this analysis is not a decision procedure, since it cannot be used in general to guarantee the absence of counterexamples for a theory [10]. Nevertheless, it is useful in practice, since it allows one to discover counterexamples of intended properties, and if none is found, gain confidence about our specifications. This is similar in spirit to testing, since one checks the truth of a statement for a number of cases; however, as explained in [11], the scope of the technique is much greater than that of testing, since the space of cases examined (usually in the order of billions¹) is beyond what is covered by testing techniques, and it does not require one to manually provide test cases.

DynAlloy assertions can also be analyzed automatically, by means of the same mechanism. Essentially, the DynAlloy Analyzer translates a DynAlloy assertion into an Alloy specification, which then can be analyzed using the Alloy Analyzer. In order to do so, the DynAlloy Analyzer needs, besides the bounds for the domains, an extra bound for iteration. This extra bound is used by the DynAlloy Analyzer to “unroll” the iterations in the program to be checked.

4 Characterizing Tables in DynAlloy

Part of the characterization of tables in DynAlloy has already been introduced in the previous section. First, type definitions, including the definition of signature `State`, are defined as shown in previous sections. The state of the system is composed of system variables, mode classes and terms. Second, each of the events mentioned in the tables gives rise to a corresponding predicate definition. Third,

¹ This is so because, even for simple specifications and relatively small bounds, the number of bounded possible instances of the model, i.e., the cases to be examined, can be very large, easily reaching billions of possible instances [11].

event independent terms and controlled variables, defined by condition tables, are constrained in Alloy using facts, as shown before for the displays. These facts are automatically synthesized from the condition tables. Fourth, mode transitions, described in a mode transition table, give rise to an Alloy predicate characterizing the transitions. In our case, this predicate is the following:

```
pred NEXTmcStatus(s, s': State) {
  s.mcStatus = ALTmode &&
    (Ev_TmATTswOn[s,s'] || Ev_ChangedmALTdesired[s,s']) =>
      s'.mcStatus = ALTmode else
    (s.mcStatus = ALTmode && Ev_TmFPAswOn[s,s'] => s'.mcStatus = FPAmode else
    (s.mcStatus = ATTmode && Ev_TmALTswOnWhentALTpreselAndtNear[s,s'] =>
      s'.mcStatus = ALTmode else
    (s.mcStatus = ATTmode && (Ev_TmFPAswOn[s,s'] ||
      Ev_TmALTswOnWhentALTpreselAndNottNear[s,s']) =>
      s'.mcStatus = FPAmode else
    (s.mcStatus = FPAmode && (Ev_TmALTswOnWhentALTpreselAndtNear[s,s'] ||
      Ev_TtNearWhentARMED[s,s'])
      => s'.mcStatus = ALTmode else
    (s.mcStatus = FPAmode && (Ev_TmATTswOn || Ev_TmFPAswOn ||
      Ev_ChangedmALTdesiredWhentARMED[s,s'])
      => s'.mcStatus = ATTmode
      else s'.mcStatus = s.mcStatus))))
}
```

Notice that we are making use of the predicates associated with the events. Clearly, predicate `NEXTmcStatus` corresponds to the formula specified using a tabular notation in SCR. We follow a similar process for controlled variables and terms defined by event tables. For instance, we will have a predicate associated with the event table defining `tcASmode`, etc. Finally, using these predicates we define a single DynAlloy action, called `stateChange`, as follows:

```
act stateChange[s: State] {
  pre { }
  post { monitoredVariableChange[s,s'] && NEXTmcStatus[s,s'] &&
    NEXTtARMED[s,s'] && NEXTtcASmode[s,s'] &&
    NEXTtALTpresel[s,s'] && NEXTtcASpresel[s,s'] && NEXTtFPApresel[s,s'] }
}
```

where `monitoredVariableChange` is an Alloy predicate characterizing the change of a monitored variable. Other elements of a tabular specification, such as initial states, are also straightforwardly characterized in DynAlloy. The important point here is that the generation of the DynAlloy specification corresponding to the tabular descriptions is fully automated.

With the DynAlloy characterization of the tabular specification, we can do various analyses. For instance, we can use the Alloy Analyzer for checking disjointness and completeness associated with tables (an example of this is assertion `DisjointnessCheck` given in the previous section). We can also check properties of *all* executions, using the DynAlloy Analyzer. For instance, we could check that, whenever the system is in the `ALTmode`, the altitude display shows the desired altitude. This is specified using the following DynAlloy assertion:

```

assert basicProgram {
  assertCorrectness[s: State] {
    pre = { initialState[s] }
    program = { stateChange[s]* }
    post = { (s'.mcStatus = ALTmode) => (s'.cALTdisplay = s'.mALTdesired) }
  }
}

```

where `initialState` is a predicate describing the initial state for the system (also synthesized from the tabular specification).

5 Specifying and Analyzing Sets of Executions via DynAlloy Programs

Most of the analysis techniques associated with tabular specifications, including the SAT based analysis described in the previous section, have their efficiency tied to the size of the specification (or, put in a different way, to the number of possible runs associated with the specification). We propose here a notation for describing subsets of the executions of a tabular specification. The notation is essentially that of DynAlloy programs, complemented with a way of restricting action `stateChange`, our only atomic action, representing a change in the state of the system as defined by the tables. Intuitively, the *conditioned* atomic action definition:

$$\text{stateChange} \ll f[s, s'] \gg [s]$$

where $f[s, s']$ is a formula referring to the pre and post states, corresponds to action

`stateChange` occurring, with f also taking place in the transition. For example, the following actions:

$$\text{stateChange} \ll \text{@T(mALTsw = on)} \gg [s] \quad \text{stateChange} \ll \text{NOT @T(mALTsw = on)} \gg [s]$$

correspond to action `stateChange`, restricted to the facts that the `mALTsw` switch must be switched on in the change, and must *not* be switched on in the change, respectively. If an action a is defined by pre and postconditions `pre_a` and `post_a`, then its semantics is the relation associated with the formula `pre_a[s]  $\wedge$  post_a[s, s']`. Action $a \ll f[s, s'] \gg$ has instead as its semantics the relation associated with the formula `pre_a[s]  $\wedge$  post_a[s, s']  $\wedge$  f[s, s']`. Since the restrictions for atomic action `stateChange` are provided by the user, the SCR notation is employed for expressing them. In this way, the specifier used to the tabular notation will not need to deal directly with (Dyn)Alloy. The idea is, of course, that (Dyn)Alloy should be used as a backend for analysis.

Let us consider as a first example of the use of the notation the following. If none of the switches in the panel are switched, then the mode remains being that of the initial state, namely `ATTmode`. This is expressed as follows:

```

assert SimpleCheck {
  assertCorrectness[s: State] {

```

```

pre = { initialState[s] }
program = { stateChange<<! $\text{@T}(\text{mATTsw} = \text{switchOn})$ 
            AND  $\text{@T}(\text{mFPAsw} = \text{switchOn})$  AND  $\text{@T}(\text{mALTsw} = \text{switchOn})$ >>[s]* }
post = { s'.mcStatus = ATTmode }
}

```

Another example is the following. Suppose that one wants to check if, once the `ALTmode` is on, if no switch is switched afterwards then either the current mode is `ALTmode`, or is back to mode `ATTmode` (i.e., the `ALTmode` is deactivated because the desired altitude was reached). This is expressed as follows:

```

assert ALTmodeDisengagedWhenALTReached {
  assertCorrectness[s: State] {
    pre = { initialState[s] }
    program = { stateChange[s]* ; [ s.mcStatus = ALTmode]? ;
               stateChange<<! $\text{@T}(\text{mATTsw} = \text{switchOn})$  AND  $\text{@T}(\text{mFPAsw} = \text{switchOn})$ 
               AND  $\text{@T}(\text{mALTsw} = \text{switchOn})$ >>[s]* }
    post = { s'.mcStatus = ALTmode || s'.mcStatus = ATTmode }
  }
}

```

These examples use conditioned actions, test actions, iteration and sequential composition. Let us now consider the following assertion: If the `mALTsw` is pressed below 1200 from the desired altitude, then if the panel is not touched and the airplane reaches an altitude higher than or equal to the desired altitude, the airplane moves to `ALTmode` and the altitude display shows the current altitude. This is expressed in the following way:

```

assert BackToALTWhenALTpassed {
  assertCorrectness[s: State] {
    pre = { initialState[s] }
    program = { stateChange[s]* ;
               [ s.mALTdesired - s.mALTactual > 1200 ]? ;
               stateChange<<! $\text{@T}(\text{mALTsw} = \text{switchOn})$ >>[s] ;
               stateChange<<! $\text{@T}(\text{mATTsw} = \text{switchOn})$  AND  $\text{@T}(\text{mFPAsw} = \text{switchOn})$ 
               AND  $\text{@T}(\text{mALTsw} = \text{switchOn})$  AND
               NOT Changed(mALTdesired) AND
               NOT Changed(mFPAdesired) AND
               NOT Changed(mCASdesired)>>[s]* ;
               [ s.mALTactual > s.mALTdesired]? ; }
    post = { s'.mcStatus = ALTmode && s'.cALTdisplay = s'.mALTactual }
  }
}

```

Via programs, we *externalize* the specification of control flow from tabular specifications. Notice that the number of possible executions of these programs if iterated just a few times is huge, due to the various different branches these can follow in different iterations; this is beyond the scope of what can practically be done by manually defining executions for simulation, for instance by using the SCR toolset's simulator. Notice that tables in a specification describe

a labeled transition relation, and a system's behaviour is understood as the reflexive-transitive closure of this relation. For these relations to adequately describe the wanted behaviours associated with any of the properties to be checked given in this section, it would be necessary to include control variables (or new modes) whose values rule out undesired control flows. Some of these variables can be deemed unnecessary by using an action language allowing us to externally define complex behaviours, as the one we propose.

Although in this paper we analyze *programs* over tabular specifications using SAT-solving, via the (Dyn)Alloy and DynAlloy analyzers, the notation is not limited to this usage. For instance, it is relatively straightforward to translate assertions of the kind shown in this section to the input languages of other analysis tools, in particular model checking tools.

6 Synthesis for Conditioned Atomic Actions

According to the semantics of conditioned atomic actions, it is clear that we can straightforwardly synthesize new DynAlloy atomic actions for each of the conditioned atomic actions used by the specifier. For instance, conditioned atomic action `stateChange<<@T(mcStatus = ATTmode)>>[s,s']` would lead to the following DynAlloy atomic action:

```
act stateChangeCondEv_TmcStatusATTmode[s: State] {
pre { } post { postStateChange[s,s'] && Trans(@T(mcStatus = ATTmode)) }
}
```

where `postStateChange` is the original postcondition of action `stateChange` and `Trans(@T(mcStatus = ATTmode))` corresponds to the mapping of `@T(mcStatus = ATTmode)` into Alloy's syntax (and the predicates associated with events introduced). However, this has some disadvantages. Conditioned actions, which should be restrictions of the original atomic actions, are actually more complex. This has a negative impact with respect to analysis, as it will be made clearer in the next section. Therefore, we consider an alternative mechanism for generating the corresponding DynAlloy definition of a conditioned atomic action. This mechanism makes use of the information associated with the wellformedness of tabular specifications (in particular, disjointness), and the semantics of tables.

Assuming that we have already checked the wellformedness of our tabular specifications, as defined in [6], the process for synthesizing a DynAlloy atomic action from a conditioned action `stateChange[e]` works by identifying a subset of the events used in the tables, the set of *incompatible* events with respect to `e`. We restrict the construction to atomic events of the form `@T(v = c)`, `@F(v = c)`, `CHANGED(x)`, `@T(v = c) WHEN cond`, `@F(v = c) WHEN cond`, and combinations of these using conjunction, disjunction and negation. Furthermore, in order to make the process efficient, it is based on the sole syntactic analysis of the events used in the tables, i.e., without resorting to the use of SAT solving to check whether two events are incompatible or not. We will describe the rules for constructing the set of incompatible events for `@T(x = c)`, but the reader can straightforwardly generalize the principles used in this construction to the other

events. Let $\mathbf{e}$ be $\text{@T}(\alpha)$. The events incompatible with $\mathbf{e}$ are the following:

- (i) $\text{@F}(\alpha)$ is incompatible with $\mathbf{e}$.
- (ii) If α is of the form $\mathbf{x} = \mathbf{c}$, with $\mathbf{x}$ a variable and $\mathbf{c}$ a constant, then $\text{@T}(\mathbf{x} = \mathbf{c}')$ and $\text{@T}(\mathbf{x} = \mathbf{c}') \text{ WHEN } \text{cond}$ are incompatible with $\mathbf{e}$, for every constant $\mathbf{c}'$ different from $\mathbf{c}$.
- (iii) If α is of the form $\mathbf{x} = \mathbf{c}$, with $\mathbf{x}$ a variable, term or mode class, then then $\text{@T}(\mathbf{y} = \mathbf{c}')$, $\text{@T}(\mathbf{y} = \mathbf{c}') \text{ WHEN } \text{cond}$, $\text{@F}(\mathbf{y} = \mathbf{c}')$, $\text{@F}(\mathbf{y} = \mathbf{c}') \text{ WHEN } \text{cond}$ and $\text{CHANGED}(\mathbf{y})$ are incompatible with $\mathbf{e}$, for every variable, term or mode class $\mathbf{y}$ which is not a predecessor nor a successor of $\mathbf{x}$ in the symbol dependency graph, and for every expression $\mathbf{c}'$.
- (iv) Conjunctions involving any of the above cases are also incompatible with $\mathbf{e}$.
- (v) Disjunctions whose all composing disjuncts correspond to the above cases are incompatible with $\mathbf{e}$.

These incompatible events are guaranteed not to occur simultaneously with $\mathbf{e}$. The reason for this in the first and second of the above cases is obvious. The third is based on the observation that, if $\mathbf{x}$ changed its value, then all the symbols which do not depend on $\mathbf{x}$ and of which $\mathbf{x}$ does not depend, cannot have changed. This has to do with the assumption that events start with the change of a single monitored variable, and the propagation of changes to the symbols depending on this variable. Notice also that the third of the above rules has as a special case that, if a monitored variable changes in a transition step, then none of the other monitored variables changes in the same transition (recall that we assume that the tabular specification is valid, implying that the symbol dependency graph is acyclic). The reason for the last two rules are obvious due to the semantics of conjunction and disjunction, respectively.

Once the set of incompatible events is constructed for $\mathbf{e}$, the process is straightforward. First, we remove the cases in the tabular specifications involving events incompatible with $\mathbf{e}$. Second, in the tables where disjointness conditions apply, the alternatives to $\mathbf{e}$ are removed. We then generate the “stateChange” formula corresponding to the resulting tables. These formulas can be much simpler than the original **stateChange** postcondition. For instance, the combined mode transition/event table given previously, restricted according to **stateChange**($\text{@T}(\text{mATTsw=on})$) is reduced to the following:

Mode Class = mcStatus				
Old mode	Events	New mode	tALTpresel	tFPAPresel
ALTmode	$\text{@T}(\text{mATTsw=on})$	ATTmode	false	false
FPAarmed	$\text{@T}(\text{mATTsw=on}) \text{ OR } \text{@T}(\text{mFPAsw=on})$	ATTmode	false	false
FPAunarmed	$\text{@T}(\text{mATTsw=on}) \text{ OR } \text{@T}(\text{mFPAsw=on})$	ATTmode	false	false

The resulting tables might not satisfy some of the wellformedness conditions (e.g., when we remove a row, we might lose completeness); however, this is not a concern, since the resulting tables are guaranteed to be equivalent to the original, if restricted to the occurrence of event $\mathbf{e}$, used to “restrict” the tables.

7 Analyzing Programs over Tabular Specifications

In order to assess our approach, we have conducted some case studies, including two different versions of the autopilot system, taken from [1,2], and a tabular specification of a mini FM radio. In order for the Alloy Analyzer to be able to handle

integer values, we had to consider abstractions of the altitude, etc, using smaller integers (smaller than 16). Although this is a limitation particularly associated with the Alloy Analyzer, this kind of abstraction process is also typical of other automated analysis techniques, such as model checking. The results of the experiments, which were carried out using an Intel Core 2 Duo of 2.2Ghz with 2GB of RAM, running the Alloy Analyzer 4.1.8 over Mac OS X, were positive. In the cases of the somewhat more preliminary specification of the autopilot system taken from [1] and the specification of the mini FM radio, the Alloy Analyzer allowed us to find various errors in the specifications, including consistency errors, errors related to misinterpretations of events, and transcription mistakes. For the better developed specification of the autopilot system in [2], we carried out experiments involving the execution programs given above. We compared the straightforward (SG) and based on event compatibility (ECG) approaches to generating definitions for conditioned atomic actions, with the latter showing a better performance compared to the former, as expected. We summarize the analysis times for two of the assertions, namely **SimpleCheck** and **BackToALTWhenALTPassed**, in the table below (times are in seconds). We found out using the (Dyn)Alloy Analyzer that this last property is, contrary to what we expected, invalid. The first counterexample obtained had to do with the altitude changing arbitrarily, and was solved by requiring it to be increased/decreased in units (which must divide the representation of 1200, used in **tNear**). The second counterexample obtained exhibited the following situation: if the airplane gets into **FPAunarmed** mode after the desired altitude has been altered, the system will stop considering that the altitude has been preselected; thus, when **mALTsw** is pressed, the event is ignored and the system will continue to be in **FPAunarmed** mode. Obviously, this has to do with a misinterpretation of the relationship between **tNear** and **tALTpresel**, which we can correct in the program by requiring in the intermediate test action not only that **tNear** be false, but also that **tALTpresel** is true. **SimpleCheck** and the corrected **BackToALTWhenALTPassed** (also included in the table) properties are valid within the provided bounds, requiring exhaustive explorations of cases for the corresponding bounds. Also, **BackToALTWhenALTPassed** contains two loops, leading to longer runs for the corresponding loop unrolls and longer analysis times, as it can be observed in the table.

Loop unrolls	SimpleCheck		BackToALTWhenALTPassed		BackToALTWhenALTPassed corrected	
	SG	ECG	SG	ECG	SG	ECG
5	.563	.602	2.740	1.233	17.442	16.181
10	1.855	1.255	6.253	6.305	106.018	122.003
20	6.184	4.510	84.421	33.518	> 60'	> 60'
40	46.749	21.648	576.402	170.946	> 60'	> 60'
80	422.722	285.784	> 60'	> 60'	> 60'	> 60'

8 Conclusions

We have proposed a formal notation for describing behaviours over tabular specifications of requirements. This formalism enabled us to describe classes of *computations*, in the sense of particular combinations of the atomic transitions specified by tables. As opposed to previous approaches, our approach allows for

the description of a wider class of combinations of the transitions defined by tables, by means of a rich operational language. This language is based on DynAlloy, an extension of the Alloy formal specification language that incorporates actions, both atomic and composite, in order to specify state change in a suitable way. Our approach comes equipped with some tool support, since, as we showed in the paper, the DynAlloy Analyzer [4] can be used to validate partial correctness assertions over tabular specifications via a SAT based mechanism (by indirectly employing the Alloy Analyzer).

The proposed notation for characterizing particular combinations of the transitions specified by tables is not restricted to SAT-based analysis. The powerful tool support associated with existing tabular notations for requirements [7,3,8,13] might benefit from the presented approach. In particular, starting from the DynAlloy assertions (accompanied by programs) of the kind presented in the paper, one can generate corresponding specifications in the input languages of model checkers. Thus, one could apply model checking, in the way that the SCR toolset applies it, to analyze the behaviours corresponding to these programs. We are confident that this would contribute to the efficiency of the analysis and the convenience of the user, by allowing the specifier to concentrate verification on particular sets of runs (specified by programs over tables) in a declarative manner. It is our aim and part of our work in progress to apply some of these analysis techniques with a focus on the runs specified by programs in our notation, rather than on the global set of behaviours.

References

1. Bharadwaj, R., Heitmeyer, C.: Applying the SCR Requirements Specification Method to Practical Systems: A Case Study. In: 21st Software Engineering Workshop, NASA GSFC (1996)
2. Bharadwaj, R., Heitmeyer, C.: Applying the SCR Requirements Method to a Simple Autopilot. In: Proc. of the Fourth NASA Langley Formal Methods Workshop (1997)
3. Bultan, T., Heitmeyer, C.: Analyzing Tabular Requirements Specifications using Infinite State Model Checking. In: Proc. of MEMOCODE 2006 (2006)
4. Frias, M., Galeotti, J.P., López Pombo, C., Aguirre, N.: DynAlloy: Upgrading Alloy with Actions. In: Proc. of ICSE 2005. ACM Press, New York (2005)
5. Gargantini, A., Heitmeyer, C.: Using Model Checking to Generate Tests from Requirements Specifications. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, p. 146. Springer, Heidelberg (1999)
6. Heitmeyer, C., Bull, A., Gasarch, C., Labaw, B.: SCR*: A Toolset for Specifying and Analyzing Requirements. In: Haverlaen, M., Dahl, O.-J., Owe, O. (eds.) Abstract Data Types 1995 and COMPASS 1995. LNCS, vol. 1130. Springer, Heidelberg (1996)
7. Heitmeyer, C., Jeffords, R., Labaw, B.: Automated consistency checking of requirements specifications. ACM Trans. on Soft. Eng. and Methodology 5(3) (1996)
8. Heitmeyer, C., Archer, M., Bharadwaj, R., Jeffords, R.: Tools for constructing requirements specifications: the SCR Toolset at the age of nine. Computer Systems: Science & Engineering 20(1) (2005)

9. Heninger, K., Kallander, J., Parnas, D., Shore, J.: Software Requirements for the A-7E Aircraft, NLR Memorandum Report 3876, US Naval Research Lab. (1978)
10. Jackson, D.: Alloy: a lightweight object modelling notation. *ACM Trans. on Soft. Eng. and Methodology* 11(2) (2002)
11. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge (2006)
12. Leveson, N., Heimdahl, M., Hildreth, H., Reese, J.: Requirements Specifications for Process-Control Systems. *IEEE Trans. on Software Engineering* 20(9) (1994)
13. Owre, S., Rushby, J., Shankar, N.: Analyzing Tabular and State-Transition Specifications in PVS. In: Brinksma, E. (ed.) *TACAS 1997*. LNCS, vol. 1217. Springer, Heidelberg (1997)

Reducing the Costs of Bounded-Exhaustive Testing

Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov

Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

{vbangal2,lee467,bdaniel3,marinov}@cs.uiuc.edu

Abstract. Bounded-exhaustive testing is an automated testing methodology that checks the code under test for *all inputs* within given bounds: first the user describes a set of test inputs and provides test oracles that check test outputs; then the tool generates all the inputs, executes them on the code under test, and checks the outputs; and finally the user inspects failing tests to submit bug reports. The costs of bounded-exhaustive testing include machine time for test generation and execution (which translates into human time waiting for these results) and human time for inspection of results. This paper proposes three techniques that reduce these costs. Sparse Test Generation skips some tests to reduce the time to the first failing test. Structural Test Merging generates a smaller number of larger test inputs (rather than a larger number of smaller test inputs) to reduce test generation and execution time. Oracle-based Test Clustering groups failing tests to reduce the inspection time. Results obtained from the bounded-exhaustive testing of the Eclipse refactoring engine show that these three techniques can substantially reduce the costs while mostly preserving fault-detection capability.

1 Introduction

Testing is an important but expensive part of software development, estimated to take more than half of the total development cost [1]. One approach to reducing the cost is to automate testing. Bounded-exhaustive testing is an automated approach that checks the code under test for *all inputs* within given bounds [2, 3, 4, 5, 6]. The rationale is that many faults can be revealed within small bounds [7, 8], and exhaustively testing within the bounds ensures that no “corner case” is missed. Bounded-exhaustive testing has been used in both academia and industry to test several real-world applications, with some recent examples including testing of refactoring engines [5] and a web-traversal code [6].

Bounded-exhaustive testing consists of three activities. First, the user describes a set of test inputs and provides test oracles that check test outputs. Second, the tool generates all the inputs, executes them on the code under test, and checks the outputs using the oracles. Third, the user inspects failing tests to submit bug reports or debug the code; typically, bounded-exhaustive testing produces a large number of *failures* for each *fault* found. Two key costs in this

context are machine time for test generation and execution (which also translates into human time for waiting for these results [9,10]) and human time for inspection of failures. Previous experience shows that bounded-exhaustive testing can discover important faults [3,4,5,11] but also can have high costs.

This paper proposes, and evaluates on a case study, three novel techniques that reduce these costs of bounded-exhaustive testing. The three techniques address various costs and can be used individually or synergistically.

Sparse Test Generation (STG): We present a new technique that reduces the time to first failure (abbreviated *TTFF*), i.e., the time that the user has to wait after starting a tool for bounded-exhaustive testing until the tool finds a failing test. Note that in this context there is usually a large number of failing tests (say, hundreds or even thousands) or no failing test (if the code under test reveals no fault for any generated test). *TTFF* measures only the time to the *first* failure (not all failures). It is an important practical metric that captures the user idle time. Previous research shows, in a related context of regression testing, that reducing the time to failure can significantly help in development [9,10]. *STG* works by making two passes through test generation. The first, *sparse*, pass skips some tests in an attempt to reduce *TTFF*. While this pass is related to test suite minimization/reduction/prioritization [12,13,14,15,16,17], the main challenge is to skip tests while they are being generated and not to select some tests only after all have been generated. The second, *exhaustive*, pass generates all the tests to ensure exhaustive checking within the given bound. Effectively, *STG* trades off (substantially) decreasing *TTFF* for (slightly) increasing the total time.

Structural Test Merging (STM): We present a new technique that reduces the total time for test generation and execution. In bounded-exhaustive testing, users typically describe a test set with a *large number of small tests*, while we advocate considering test sets with a *smaller number of larger tests*. Our technique is inspired by the work on test granularity [18,19] which studied the cost-benefit trade-offs in using a larger number of smaller tests versus a smaller number of larger tests. That work mostly considered manually written tests for regression testing, while we focus on automatically generated tests. Moreover, that work considered cases where larger tests can be automatically built from smaller tests by simply *appending* (e.g., if each test is a sequence of commands, a longer test sequence can be obtained by simply appending a number of shorter test sequences), while we consider cases where it is harder to build larger tests from smaller tests (e.g., simply appending two test input programs together while testing a compiler or a refactoring engine would likely result in a compilation error as these programs have program entities with the same name; moreover, renaming would reduce the opportunity of speeding up test execution). Instead of simply appending tests, our technique *merges* them based on their structure, hence the name *STM*.

Oracle-Based Test Clustering (OTC): We present a new technique that reduces the human time for inspection of failing tests. Bounded-exhaustive testing can produce a large number of failing tests, and a tester/developer has to map

these failures to distinct faults to submit bug reports or debug the code under test. Our technique builds on the ideas from test clustering [20, 21, 22, 23, 24, 25] where the goal is to split (failing) tests into groups such that all tests in the same group are likely due to the same underlying fault. Previous work mostly considered manually written tests or actual programs runs, and clustering was based on *execution profiles* obtained from monitoring test execution. In contrast, we consider automatically generated test inputs, and our technique exploits information from oracles. Typically, an oracle only states *if* some test passed or failed, i.e., the output from an oracle is a boolean. However, in some domains oracles also state *how* the result is incorrect, i.e., the output from an oracle is an error *message*. OTC splits tests based on oracle messages, and our results suggest that it is beneficial to build such oracles whenever possible. The key to our technique is *abstracting* messages and not comparing them directly.

Case Study: We implemented our three new techniques in the ASTGen framework for bounded-exhaustive testing of refactoring engines [5]. We chose ASTGen for three reasons: it had enabled finding actual faults in real large software (we had found a few dozens of new faults in the refactoring engines of Eclipse and NetBeans, two popular IDEs for Java [5]); we were familiar with the framework; and we personally experienced the costs of using ASTGen. We evaluated the techniques on testing 6 refactorings with 9 generators (explained later in the text). The results show that (1) STG can reduce TTFF almost 10x (an order of magnitude) when there is a failure, while increasing the total test generation and execution time only 10% when there is no failure; (2) STM can reduce the total time 2x-6x (in one instance from over 6 hours to 70 minutes) and even more (but with some reduction of the fault-detection capability); and (3) OTC can reduce the number of tests to be inspected by clustering hundreds of failing tests into a few groups (up to 11) such that almost all tests within the group are due to the same fault. In summary, the results show that the new techniques can substantially reduce both the machine time and the human time without reducing the fault-detection capability.

2 Example

To illustrate how our techniques reduce the costs of bounded-exhaustive testing, we discuss testing of the PullUpMethod refactoring in the Eclipse refactoring engine using the ASTGen framework. We first describe what PullUpMethod is. We then describe how to use ASTGen for bounded-exhaustive testing of this refactoring. We finally discuss how our new techniques improve on ASTGen.

Each refactoring is a program transformation that changes the program code but not its external behavior [26]. Programmers undertake refactorings to improve design of their programs. For example, PullUpMethod is a refactoring that moves a method from some class into one of its superclasses (usually because the same method is useful for other subclasses of that superclass). Figure 1 shows a simple application of the PullUpMethod refactoring. Note that moving the

<pre>// Before refactoring class A { int f; } class B extends A { void m() { super.f = 0; } }</pre>	<pre>// After refactoring class A { int f; void m() { this.f = 0; } } class B extends A { }</pre>	<pre>// Before refactoring class A { } class B extends A { int f; void m() { this.f = 0; } }</pre>	<pre>// Refactoring engine // warning: // Cannot pull up: // method 'm' // without pulling up: // field 'f'</pre>

Fig. 1. Example applications of the PullUpMethod refactoring

method also requires properly updating the references within the method body, i.e., replacing `super.f` with `this.f`.

Refactoring engines are development tools that automate applications of refactorings. They are an important part of modern IDEs such as Eclipse [27]. To apply PullUpMethod, the developer instructs the engine which method to move to which superclass in the *input* program. The engine first checks whether the move is permitted (e.g., PullUpMethod should not move a method to a superclass if the superclass already has a method with the same signature). If it is, the engine appropriately transforms the program. The *output* is either a transformed program or a set of warning messages that indicate why the move would not be permitted, as illustrated in Figure 1.

Testing the implementation of PullUpMethod requires generating a number of input programs, invoking the refactoring engine on them, and checking whether it gives the appropriate output (either a correctly transformed program or an expected set of warning messages). Testers can have good intuition about which input programs could reveal a fault. For instance, PullUpMethod may have faults if the subclass and superclass have some additional relationship, e.g., being an inner or a local class or being related through a third class. Also, there may be faults for some expressions and statements that include field and method references from the body of the method being pulled up or to the method being pulled up. However, it is time-consuming and error-prone to manually generate a large number of such input programs.

We previously developed the ASTGen framework for bounded-exhaustive testing of refactoring engines [5]. ASTGen allows the tester to write *generators* that can automatically produce a (large) number of (small) input programs for testing refactorings. ASTGen generates *all* these inputs, executes the refactoring engine on them, runs several oracles to validate the outputs, and reports failures.

For instance, to test PullUpMethod, we can use a generator that produces programs with three classes in various relationships. For this specific case, ASTGen generates 1,152 input programs, of which 160 result in failing oracles. A detailed inspection of these failures shows that they reveal 2 distinct faults. While finding these faults is clearly positive, there are costs. Test generation and execution (including oracles) take about 27 minutes (on a typical desktop), and the time to find the first failure is about 9 minutes. Also, identifying the 2 distinct faults among 160 failing tests is labor-intensive and tedious.

This paper proposes three techniques that reduce these costs. *STG* addresses the time to first failure (TTFF) by first sampling some inputs rather than

exhaustively generating all inputs from the beginning. For our specific example, TTFF is on average reduced almost an order of magnitude, from about 9 minutes to 1 minute. *STM* addresses the total time for test generation and execution. Instead of testing PullUpMethod for 1,152 (small) programs that exercise various features in isolation, STM builds larger programs that combine some of the features, e.g., combine several expressions or statements that include field and method references to/from the method being pulled up. The tester can choose how many features to combine. In this example, the least aggressive combination reduces the total time from 27 minutes to about 4 minutes, and the most aggressive combination reduces the total time further to under 1 minute. *OTC* addresses the cost of failure inspection. It clusters the failing tests into groups that are likely to be due to the same fault, and thus the tester can inspect only one or a few tests from these “equivalence classes”. Our clustering is based on oracle messages and can consider more or fewer details of the messages. The basic clustering splits 160 failing tests into 127 clusters, but our best clustering splits them into just 3 clusters that reliably find the 2 faults. In contrast, random sampling could miss faults, e.g., one of our experiments shows that it finds on average 1.77 out of 2 faults in this case.

3 Background: ASTGen

We now describe in more detail two parts of the ASTGen framework that are relevant to present the three techniques introduced in this paper. ASTGen allows the testers to write *generators*—pieces of code that implement a specific interface—which ASTGen runs to automatically generate input programs. ASTGen then applies refactorings on these inputs and runs the *oracles* on the outputs.

Generators: Each generator is a piece of Java code that produces elements of Java abstract syntax trees (ASTs), which can be pretty-printed as Java source. Conceptually, generators are close to grammar-based generation [28, 29], but ASTGen uses Java code rather than a grammar formalism as explained elsewhere [5]. ASTGen provides (1) a large library of basic generators, (2) several mechanisms to compose and link simpler generators into more complex generators, and (3) customization of generators using Java code. Some of the generators that ASTGen provides include:

Field Declaration Generator produces many different field declarations that vary in terms of type (`int`, `byte`, `boolean`, array or non array, etc.), visibility (`private`, `public`, etc.), and name of the declared field.

Field Reference Expression Generator is linked to the Field Declaration Generator and produces different expressions that reference the declared field in various ways, including field accesses and operations (`this.f`, `new A().f`, `super.f`, `f++`, `!f`, etc.).

Single Class Field Reference Generator is composed on top of the Method Declaration Generator and produces classes with one field (obtained from the Field Declaration Generator) and one method that references the field in various ways.

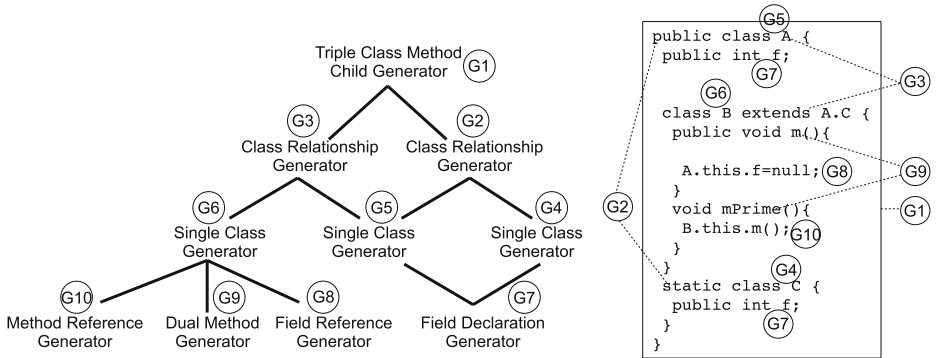


Fig. 2. Triple Class Method Child Generator structure and a generated test input

Dual Class Relationship Generator is composed upon generators that produce classes (e.g., Single Class Field Reference Generator) and produces two classes with various relationships between them (inheritance, inner class, local class, etc.).

While the main purpose of generators is to actually produce the test inputs, they also encode the space of all inputs to be produced. Consider this scenario:

Inputs: Programs with three classes A, B, and C.

- B extends C; B has a method `m` and a method `mPrime` that invokes `m`.
- C and A each have a field `f` that may be referenced by `m`.

Test: Pull up method `m` from class B to class C.

The user can generate all these inputs by writing a generator that composes and links several library generators. Figure 2 shows the overall structure of a generator, called Triple Class Method Child Generator, that encodes this input space. The figure also shows a sample test input produced by this generator and how the input sub-parts match the sub-generators responsible for producing them. By iterating through all the variations of the sub-generators, the Triple Class Method Child Generator produces 1,152 test inputs.

Oracles: While generators are the core of ASTGen and help testers to produce a large number of input programs for testing refactorings, it would be impractical that the testers check the result of each refactoring application. Oracles automate checking of the results so that the testers only have to inspect a smaller number of tests that fail the oracles (and likely detect faults). ASTGen provides two generic oracles and allows the users to write refactoring-specific oracles:

Compilation Failure Oracle flags tests where the refactored program has a compilation error: if the input program compiles, then the output program should also compile.

Erroneous Warning Oracle flags tests where the refactoring engine raised a warning about a refactoring application, but ignoring that warning results in a refactored program with no compilation errors (or custom failures).

Custom Oracles are specific to the refactoring being applied. For example, a custom oracle for `RenameMethod` checks that renaming a method, say `m` to `p`, and then renaming back, `p` to `m`, results in the same program.

The output of traditional oracles are only booleans (pass or fail), but the AST-Gen oracles can provide additional information about the failure, e.g., messages from the compiler or warnings from the refactoring engine.

4 Sparse Test Generation (STG)

Generators can encode and produce all the test inputs within defined bounds. Bounded-exhaustive testing checks the code under test for all these inputs. This usually consumes a large amount of machine time since the number of inputs generated is fairly large. For example, ASTGen generators can generate thousands of test inputs, and it can take hours of machine time to execute the refactorings on all those inputs. Additionally, this time translates into human time required by the developer to wait for the execution of the tests to complete.

Note that as soon as a tool reports a failure, the developer can start inspecting it to file a bug report or to debug the fault that caused the failure. In theory, the time the tool takes for generation and testing after the first failure is not important since the developer does not have to idle. For this reason, we consider the Time to First Failure (TTFF) as the key metric in interactive bounded-exhaustive testing. If no generated test input results in a failure, the developer has to wait for the entire generation and testing process to complete.

STG is our technique that aims to reduce the TTFF. STG has two phases:

Sparse Generation is motivated by our observation that failing test inputs are often located closely together in the sequence of inputs produced by a generator, and thus, to find a failure, it is often not necessary to exhaustively generate all the inputs but only one input from a closely located group. Therefore, this phase makes “jumps” through the generation sequence. The jump length is not constant (since the failing tests may be in a stride that a constant jump would miss) but *each jump is (uniformly) random within some length limit*. The key is to determine an appropriate limit: a lower limit increases the overhead of STG compared to the basic, *dense* bounded-exhaustive testing, while a higher limit decreases the chance that Sparse Generation finds a failure (and thus increases the TTFF). We use the limit of 20 as it provides a good trade-off: the expected jump is of length $(1+20)/2$, which increases the total time by less than 10% when there is no failure. If Sparse Generation finds a failing test, it usually does so quickly; the results from Section 7 show that STG reduces the TTFF by an order of magnitude in most cases compared to the dense generation. However, STG is a heuristic and, in general, could keep missing failures until the very end while dense generation would have found those failures at the very beginning.

Exhaustive Generation follows Sparse Generation and does basic bounded-exhaustive testing (1) to ensure that a failing test input will be found if one exists and (2) to find all the failing tests that Sparse Generation missed (which can help in clustering failures or debugging [20, 23, 25]).

<pre> public class A { public int f; class B extends C { private void m(){ this.f=0; } void mPrime(){ m(); } } } class C { public int f; } </pre>	<pre> public class A { public int f; class B extends C { private void m(){ new A().f=0; } void mPrime(){ m(); } } } class C { public int f; } </pre>	<pre> public class A { public int f; class B extends C { private void m(){ super.f=0; } void mPrime(){ m(); } } } class C { public int f; } </pre>

Fig. 3. Unmerged test inputs

5 Structural Test Merging (STM)

TTFF is an important metric in bounded-exhaustive testing. Another important metric is the total time for test generation and execution. This time can be very long when generators produce a large number of inputs, which is the case for typical top-level ASTGen generators. For example, consider the number of inputs for the Triple Class Method Child Generator shown in Figure 2. Each of its sub-generators has a small number of variations—G2 has 2 (inner, outer); G3 has 3 (inner, method inner, outer); G4, G5, and G6 have 1; G7 has 2 (public, private); G8 has 6 (f, new A().f, A.this.f, etc.); G9 has 4 (public, private, same/different signature); and G10 has 4 (m(), new B().m(), this.m(), B.this.m())—but the top-level generator produces $2 \times 3 \times 1 \times 1 \times 1 \times 2 \times 6 \times 4 \times 4 = 1152$ combinations.

STM reduces the number of test inputs while still aiming to preserve their exhaustiveness: instead of producing a large number of small input programs, STM produces a smaller number of larger input programs by *merging* appropriate program elements. For example, the Triple Class Method Child Generator produces the three inputs shown in Figure 3. The only difference between the three are the highlighted statements, generated by the Field Reference sub-generator (G8). Figure 4 shows an input that contains all these three statements. This single, merged input encodes the same input space as the three unmerged inputs. This structural merging transformation is the crux of our STM technique.

STM exploits the compositional structure of the sub-generators to produce merged test inputs. Figure 4 shows an alternative structure for the Triple Class Method Child Generator: a new Field Reference Merging Single Class Generator (G8M) merges together all the program elements produced by the original Field Reference Generator (G8). While figures 2 and 4 show a generator before and after a *single* application of the structural merging transformation, it is possible to apply the transformation *multiple* times within the hierarchical structure of a generator. Each application leads to a multiplicative reduction in the number of generated inputs. For example, the original Method Reference Generator (G10) can also be modified to a generator G10M that merges together all the different method invocation statements. Together, G8M and G10M produce inputs that merge *both* field references and method references. We refer to the number of

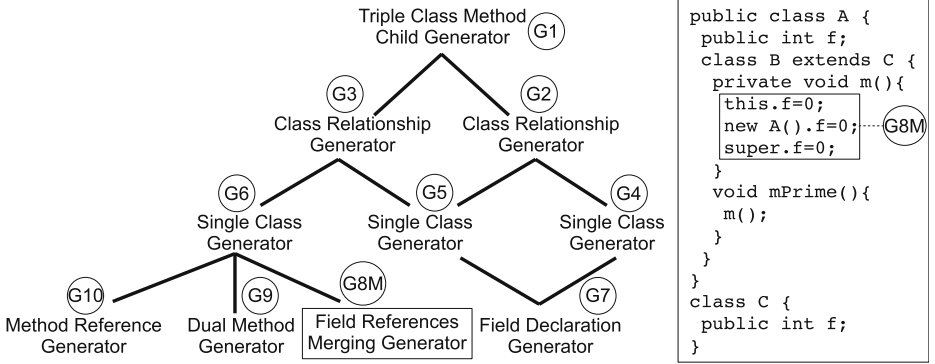


Fig. 4. Merged generator structure and a generated merged test input

transformation applications as *merging level*: for the Triple Class Method Child Generator, merging level M1 has only G8M, and merging level M2 has both G8M and G10M. The unmerged generator produces 1,152 inputs, and levels M1 and M2 reduce the number of inputs to 192 and 48, respectively.

While STM achieves significant time savings, it is important to note its two potential drawbacks. One potential drawback is that larger inputs, through the interference of program elements, can mask some test failures [18,19]. Consider, for example, merging together all the different field references (as in Figure 4). There may be a failure triggered by one of the field reference statements which gets masked by the presence of the other field reference statements. However, this interference can also go the other way: larger inputs may trigger new failures that smaller inputs do not trigger. The other drawback is the effect of larger inputs on debugging. STM produces fewer larger inputs rather than more smaller inputs, but (failing) smaller inputs typically make it easier to perform fault localization. We could take two approaches to address this. One approach is to reduce inputs by applying Delta Debugging [30] on the larger failing input to try to isolate the part of the input that triggers the failure. Another approach, enabled by the fact that larger inputs are produced by merging generators, is to regenerate the small inputs that represent the larger failing input.

6 Oracle-Based Test Clustering (OTC)

The experience with bounded-exhaustive testing in academia and industry shows that it can find faults in real code [4,3,5,11] but also produces a large number of failures. Identifying a few faults out of many failures is a challenging task. OTC is a new technique that helps in this task by splitting failing tests into groups such that all tests in the same group are likely due to the same fault.

OTC exploits information from oracles. Recall that ASTGen oracles provide messages about the failures, e.g., if a refactored program does not compile, ASTGen reports the compilation error provided by the compiler. We use these messages to cluster the failing tests by grouping together those tests that have

exactly the same messages. (A test can produce multiple messages, which our experiments compare as lists, not bags or sets.) However, directly using *concrete* messages provided by the compiler can result in a large number of small clusters, e.g., two compilation errors may differ only in line or column numbers, say, “3:8:field f not visible” and “2:6:field f not visible”. Instead, we use *abstract* messages that ignore some details such as line and column numbers. One can further consider ignoring exact messages and clustering based on *which* oracle failed, *not how* it failed. The trade-off is that creating too many clusters increases inspection effort, while creating too few clusters increases the chance to miss a fault. Our evaluation compares four clustering options: Concrete Message, Abstract Message, Oracle Name, and Random Selection (a base case with no clustering).

7 Case Study

We evaluated our three new techniques in the ASTGen framework for bounded-exhaustive testing of refactorings engines. We tested 6 refactorings using 9 generators listed in Figure 5. For each generator and several merging levels, we tabulate the number of inputs generated, various times and APFD metric (described below), the number of failing inputs, and the number of distinct faults. We previously tested these refactorings with these generators and found a number of faults [5]. The goal of this study was to evaluate whether the new techniques reduce the testing costs, but due to OTC, we also found a new fault in the PushDownMethod refactoring, previously missed [5] due to random sampling. We ran all experiments in Eclipse 3.3.2 on a dual core 3.4GHz machine.

Sparse Test Generation (STG): Figure 5 shows the time results for ASTGen with and without STG. The ‘Dense’ subcolumns show the total time and time to first failure (TTFF) for bounded-exhaustive testing without STG. If no failure exists, TTFF shows ‘n/a’. The ‘Sparse’ column shows average values for TTFF if a failure exists (roughly the top half of the table) and the total time if no failure exists (the bottom half of the table). These times are averaged over 20 random seeds, with the jump limit of 20, as discussed in Section 4. The main questions about STG are how it affects TTFF and the total time.

STG reduces TTFF in all cases where the dense TTFF was significant (a minute or more): the speedup ranges from 9.00x to 10.58x, with an average of an order of magnitude. In a few cases with very small dense TTFF, STG had a slowdown of at most 1 sec. Recall the two phases of STG; the reduction in TTFF implies that the sparse phase found a failure before the exhaustive phase.

STG increases the total time, as expected. With the jump limit of 20, the overhead of the additional sparse phase is expected to be slightly under 10% of the total time for dense generation. Our experiments confirm that this is indeed the case: the slowdown ranges from -6.48% to -9.60%. In summary, STG achieves a 10x speedup in TTFF for only a 10% slowdown in the total time.

We further evaluated STG using the Average Percentage Fault Detected (APFD) metric introduced by Rothermel et al. [12] to compare techniques for

Refactoring	Generator	ML	Inputs	Dense		Sparse	APFD [%]		Failures	Faults
				Time	TTFF		Dense	Sparse		
PushDown-Field	DualClass-FieldReference	M0	7416	133:32	7:33	0:47	74.98	98.16	1074	2
		M1	1236	22:43	0:01	0:02	99.23	88.62	179	1
		M2	12	0:21	0:00	0:01	95.83	74.17	4	1
		M3	3	0:13	0:00	0:00	83.33	63.33	1	1
Encapsulate-Field	DualClass-FieldReference	M0	23760	427:09	73:34	7:14	58.03	97.59	486	3
		M1	3960	71:50	12:03	1:11	69.82	97.77	354	3
		M2	72	1:19	0:13	0:03	74.31	80.56	31	2
		M3	18	0:26	0:06	0:03	58.33	73.15	8	2
	SingleClass-FieldReference	M0	8576	155:15	0:22	0:03	75.37	97.61	836	4
		M1	2144	39:04	0:21	0:03	66.86	97.59	242	4
		M2	1072	19:35	0:09	0:02	84.25	93.04	144	3
		M3	268	4:55	0:02	0:02	72.70	88.11	62	3
PushDown-Method	DualClass-MethodParent	M0	960	22:19	11:28	1:05	43.91	93.59	180	3
		M1	192	4:07	2:07	0:14	41.75	91.89	38	3
		M2	48	0:45	0:28	0:21	40.63	87.85	2	1
		M3	16	0:17	0:00	0:01	96.88	84.06	1	1
PullUp-Method	TripleClass-MethodChild	M0	1152	27:02	9:09	1:01	13.19	95.77	160	2
		M1	192	3:57	1:25	0:09	48.18	95.36	96	2
		M2	48	0:47	0:17	0:02	56.25	89.58	24	2
	DualClass-MethodChild	M0	576	13:22	n/a	14:14	n/a	n/a	0	0
		M1	96	1:49	n/a	1:55	n/a	n/a	0	0
		M2	24	0:21	n/a	0:22	n/a	n/a	0	0
Rename-Field	DualClass-FieldReference	M0	23760	629:01	n/a	689:17	n/a	n/a	0	0
		M1	3960	107:26	n/a	117:48	n/a	n/a	0	0
		M2	72	1:56	n/a	2:04	n/a	n/a	0	0
		M3	18	0:34	n/a	0:34	n/a	n/a	0	0
	SingleClass-FieldReference	M0	8576	229:00	n/a	250:59	n/a	n/a	0	0
		M1	2144	57:28	n/a	62:56	n/a	n/a	0	0
		M2	1072	28:44	n/a	31:28	n/a	n/a	0	0
		M3	268	7:15	n/a	7:57	n/a	n/a	0	0
Rename-Method	SingleClass-MethodReference	M0	9540	173:32	n/a	190:11	n/a	n/a	0	0
		M1	4900	89:26	n/a	98:05	n/a	n/a	0	0
		M2	140	2:37	n/a	2:50	n/a	n/a	0	0
		M3	80	1:31	n/a	1:37	n/a	n/a	0	0

Fig. 5. Sparse Test Generation and Structural Test Merging Results

Legend: ML = Merging Level, TTFF = Time to First Failure, All times in minutes:seconds

test prioritization and extended by Walcott et al. [16] for test selection. APFD measures the number of faults detected in terms of the number of tests executed, whereas TTFF is based on the first failure (not all faults) and actual time (not number of tests) as TTFF aims to capture the waiting time for testers in interactive bounded-exhaustive testing, similar to recent extensions of APFD [10]. APFD ranges between 0 and 100%, with higher values being better. Figure 5 shows APFD, with ‘Sparse’ averaged over 20 random seeds. The results show that STG improves APFD in all cases where the dense TTFF was significant.

Structural Test Merging (STM): Figure 5 shows the results for STM for several merging levels of each of the generators. The merging level number (e.g., 3 in M3) represents the number of structural merging transformations applied to the unmerged generator (labeled M0) to obtain the corresponding merged generator, as discussed in Section 5. The main questions about STM are how it affects times (total and TTFF) and the number of failures/faults detected.

Each merging level reduced both the total time and TTFF compared to its previous level and thus to M0. On average, level M1 achieved 5x speedup, and level M2 achieved 130x speedup compared to M0 for the total time. The merged

Refactoring	Generator	ML	Random		Oracle		Abstract		Concrete	
			FD	NC	FD	NC	FD	NC	FD	NC
PushDownField	DualClassFieldReference	M0	1.99	1	2	2	2	5	2	68
		M1	1	1	1	1	1	3	1	59
		M2	1	1	1	1	1	2	1	4
		M3	1	1	1	1	1	1	1	1
EncapsulateField	DualClassFieldReference	M0	2.24	1	2.24	2	3	4	3	51
		M1	2.05	1	2.31	2	3	4	3	112
		M2	1.73	1	2	2	2	4	2	8
		M3	1.75	1	2	2	2	3	2	3
	SingleClassFieldReference	M0	2.84	1	3.11	2	4	4	4	71
		M1	2.48	1	2.46	2	4	4	4	73
		M2	2.26	1	2.26	1	3	4	3	58
		M3	2.30	1	2.30	1	3	5	3	24
PullUpMethod	TripleClassMethodChild	M0	1.77	1	1.77	1	2	3	2	127
		M1	1.56	1	1.56	1	2	2	2	84
		M2	1.62	1	1.62	1	2	2	2	24
PushDownMethod	DualClassMethodParent	M0	2.19	1	3	2	3	11	3	20
		M1	2.56	1	2.54	2	3	10	3	16
		M2	1	1	1	1	1	1	1	2

Fig. 6. Oracle-Based Test Clustering Results

Legend: ML = Merging Level, FD = Faults Detected, NC = Number of Clusters

generators also substantially reduced the TTFF: on average, level M1 achieved 80x speedup, and level M2 generators achieved 150x speedup compared to M0.

Merging did not expose any new faults, but aggressive merging did mask some faults. In particular, level M1 masks only one fault (in PushDownField), but levels M2 and higher mask a much larger number of faults. However, even the highest level of merging finds at least one fault (when there is a fault at M0). Additionally, if one considers TTFF as the most important metric, masking faults at the higher merging levels is not detrimental but actually beneficial: the user can start the exploration from a high level, quickly find failures, and start inspecting them, while the tool continues the exploration at a lower level. In summary, STM can substantially improve total time and TTFF while somewhat reducing the fault-detection capability of bounded-exhaustive testing.

Oracle-Based Test Clustering (OTC): Figure 6 shows the results for the four clustering options discussed in Section 6. For each option, we present the number of clusters formed and distinct faults detected by inspecting a number of randomly selected tests from each cluster. The results are averaged over 1000 random seeds. For this experiment, we needed to choose a sampling strategy [20], which determines how many tests to select and from which clusters. The basic strategy, one-per-cluster, selects one test for each cluster; we used this strategy for Abstract Message and Concrete Message. For Random Selection and Oracle Name, which have fewer clusters, we used a strategy that selects more tests per cluster, specifically selects at least as many tests as Abstract Message selects (i.e., the number of clusters that Abstract Message has) and at most 1% of all failing tests. The main questions about OTC are how it affects the number of failures that need to be inspected and the number of faults detected.

To measure the number of faults detected by a set of selected tests, we had to map failing tests to the fault(s) they detect and also had to determine which

faults are *distinct*. We performed two steps. First, a researcher (the second paper author) manually inspected all tests from each cluster (based on Abstract Message) with less than 30 tests and inspected at least 10 tests from each cluster with more than 30 tests. Since all inspected tests from each cluster detected the same fault(s), we extrapolated that all tests in a cluster can detect the same fault(s). We also patched 6 of these faults in Eclipse and confirmed their results from the first step. Second, we asked a researcher (unaware of the details of this study but with a multi-year experience with Eclipse refactorings) to label the faults collected in the first step as potential duplicates of each other or non-faults. This resulted in 12 distinct faults that we used in our experiments.

Abstract Message substantially reduces the number of tests to be inspected to find all the faults, e.g., PullUpMethod for M0 has 160 failing tests, but Abstract Message splits them into 3 clusters, and selecting any 3 tests, one from each cluster, always reveals all 2 faults. The results show that Abstract Message finds *all faults* that Concrete Message finds but requires inspection of much fewer tests, up to over an order of magnitude for lower merging levels. Also, Abstract Message finds *more faults* than Random Selection and Oracle Name while the same number or even fewer tests are inspected. In summary, Abstract Message was the most effective OTC option among the four we compared.

8 Related Work

There is a large body of work on automated testing. Our focus is on bounded-exhaustive testing [2, 3, 4, 5, 6, 11] that tests the code for all inputs within given bounds. Previous work considered how to describe a set of inputs (using declarative [2, 4] or imperative [5] approaches) and how to efficiently generate them. Bounded-exhaustive testing has been successfully used to reveal faults in several real applications [3, 4, 5, 11], but it has costs in machine time for test generation and execution and human time for inspection of failures. This paper presents three new techniques that reduce the costs of such testing.

STG is related to work on test selection/reduction/prioritization [12, 13, 14, 15, 16, 17, 31, 32] whose goal is to reduce the testing cost or to find faults faster by selecting a subset of tests from a test suite and/or ordering these tests. The previous techniques mostly consider regression testing where a test suite exists a priori, and the simplest techniques can randomly select or order these tests. In contrast, STG selects tests while they are being generated, and generation proceeds in a particular order, so arbitrary random sampling is not possible. Finally, STG does not compromise the fault-finding ability [32].

STM is related to work on test granularity [18, 19] which studied the cost-benefit trade-offs in testing with a larger number of smaller tests versus a smaller number of larger tests. The key difference is that previous work considered tests that can be easily *appended* while we consider tests that need to be *merged*. Note that appending tests only saves setup and teardown costs [19], while merging can also reduce test execution cost (e.g., merging 1,152 input programs into 192 input programs requires only 192 applications of the PullUpMethod refactoring).

However, the results are similar in both contexts: larger tests reduce the testing time, but too large tests may miss faults.

OTC is related to work on test clustering/filtering/indexing [20, 21, 22, 23, 24, 25]. Previous work performed clustering based on *execution profiles*, obtained from monitoring test execution. The main novelty of our technique is to exploit information-rich oracles, rather than execution profiles, to cluster failing tests. Our goal is to cluster failing tests to help in identifying the underlying faults. Dickinson et al. [20] present an empirical study that evaluates somewhat different techniques whose goal is to find failures among executions by using cluster analysis of execution profiles. Effectively, those techniques use cluster analysis as approximate oracles. Their results show that cluster filtering of executions can find failures more effectively than random sampling, and that clustering of executions can distinguish failing executions from passing ones.

9 Conclusions

Bounded-exhaustive testing checks the code for all inputs within given bounds. It can find faults but at potentially high costs, including machine time to generate and run tests, and human time to wait for the test results and to inspect failures. We presented three techniques that reduce these costs: Sparse Test Generation skips some tests to reduce the time to first failure by an order of magnitude; Structural Test Merging generates larger tests to reduce test generation and execution time by order(s) of magnitude; and Oracle-based Test Clustering groups failing tests to reduce the inspection time by order(s) of magnitude.

Acknowledgments. We thank Danny Dig for inspecting the faults we found in Eclipse, the anonymous reviewers for useful comments, and the students from the Fall 2008 Advanced Topics in Software Engineering class at our department for their feedback on this work. This material is based upon work partially supported by the NSF under Grant Nos. CCF-0746856, CNS-0615372, and CNS-0613665.

References

1. Beizer, B.: Software Testing Techniques (1990)
2. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: ISSTA (2002)
3. Sullivan, K., Yang, J., Coppit, D., Khurshid, S., Jackson, D.: Software assurance by bounded exhaustive testing. In: ISSTA (2004)
4. Khurshid, S., Marinov, D.: TestEra: Specification-based testing of Java programs using SAT. *Auto. Soft. Eng. Jour.* (2004)
5. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: ESEC/FSE (2007)
6. Misailovic, S., Milicevic, A., Petrovic, N., Khurshid, S., Marinov, D.: Parallel test generation and execution with korat. In: ESEC/FSE (2007)
7. Marinov, D., Andoni, A., Daniliuc, D., Khurshid, S., Rinard, M.: An evaluation of exhaustive testing for data structures. Technical report, MIT CSAIL (2003)

8. Jackson, D.: *Software Abstractions: Logic, Language and Analysis* (2006)
9. Saff, D., Ernst, M.D.: Reducing wasted development time via continuous testing. In: ISSRE (2003)
10. Do, H., Rothermel, G.: An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In: ESEC/FSE (2006)
11. Stobie, K.: Model based testing in practice at Microsoft. *Electr. Notes Theor. Comput. Sci.* 111, 5–12 (2005)
12. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Test case prioritization: An empirical study. In: ICSM (1999)
13. Elbaum, S., Malishevsky, A., Rothermel, G.: Incorporating varying test costs and fault severities into test case prioritization. In: ICSE (2001)
14. Kim, J.M., Porter, A.: A history-based test prioritization technique for regression testing in resource constrained environments. In: ICSE (2002)
15. Srivastava, A., Thiagarajan, J.: Effectively prioritizing tests in development environment. In: ISSTA (2002)
16. Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.: Time-aware test suite prioritization. In: ISSTA (2006)
17. Yu, Y., Jones, J.A., Harrold, M.J.: An empirical study of the effects of test-suite reduction on fault localization. In: ICSE (2008)
18. Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P., Davia, B.: The impact of test suite granularity on the cost-effectiveness of regression testing. In: ICSE (2002)
19. Rothermel, G., Elbaum, S., Malishevsky, A.G., Kallakuri, P., Qiu, X.: On test suite composition and cost-effective regression testing. In: ACM TOSEM (2004)
20. Dickinson, W., Leon, D., Podgurski, A.: Finding failures by cluster analysis of execution profiles. In: ICSE (2001)
21. Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B.: Automated support for classifying software failure reports. In: ICSE (2003)
22. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: Sober: statistical model-based bug localization. In: ESEC/FSE (2005)
23. Jones, J.A., Harrold, M.J., Bowring, J.F.: Debugging in parallel. In: ISSTA (2007)
24. Runeson, P., Alexandersson, M., Nyholm, O.: Detection of duplicate defect reports using natural language processing. In: ICSE (2007)
25. Liu, C., Zhang, X., Han, J., Zhang, Y., Bhargava, B.K.: Indexing noncrashing failures: A dynamic program slicing-based approach. In: ICSM (2007)
26. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: *Refactoring: Improving the Design of Existing Code* (1999)
27. Eclipse Foundation, T.: Eclipse project, <http://www.eclipse.org>
28. Duncan, A.G., Hutchison, J.S.: Using attributed grammars to test designs and implementations. In: ICSE (1981)
29. Maurer, P.M.: Generating test data with enhanced context-free grammars. *IEEE Soft* (1990)
30. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. *IEEE Trans. Soft. Eng.* (2002)
31. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. *ACM TOSEM* (1997)
32. Heimdahl, M.P.E., Devaraj, G.: Test-suite reduction for model based tests: Effects on test quality and implications for testing. In: ASE (2004)

Logical Testing

Hoare-style Specification Meets Executable Validation

Kathryn E. Gray and Alan Mycroft

University of Cambridge Computer Laboratory
{Kathryn.Gray, Alan.Mycroft}@cl.cam.ac.uk

Abstract. Software is often tested with unit tests, in which each procedure is executed in isolation, and its result compared with an expected value. Individual tests correspond to Hoare triples used in program logics, with the pre-conditions encoded into the procedure initializations and the post-conditions encoded as assertions. Unit tests for procedures that modify structures in-place or that may terminate unexpectedly require substantial programming effort to encode the postconditions, with the post-conditions themselves obscured by the test programming scaffolding. The correspondence between Hoare logic and test specifications suggests directly using logical specifications for tests. The resulting tests then serve the dual purpose of a formal specification for the procedure.

We show how logical test specifications can be embedded within Java and how the resulting test specification language is compiled into Java; this compilation automatically redirects mutations, as in software transactional memory, to support imperative procedures. We also insert monitors into the tested program for coverage analysis and error reporting.

1 Introduction

A unit test comprises a statement of the initial conditions, a statement or expression to evaluate, and a statement of concrete expectations for the result — in essence, a Hoare triple. The similarity of these components and their roles suggests that test specifications can draw from logical specifications in syntax and semantics. With similar specifications, tests serve as both runtime and logical validation; however, directly encoding tests into a standard programming language complicates the specification of some tests, making the connection to logical specifications needlessly complicated.

In discussing test specifications, we separate procedures into two broad classes: *generative* procedures that construct new data and *imperative* procedures that modify data structures and redirect bindings. As generative procedures do not modify any values, the associated test specifications closely resemble Hoare-style post-conditions that only involve the current values of variables. Post-conditions for imperative procedures in general need to refer to an initial value, commonly referred to as *old*, as well as the current, modified, value. Providing access to both values in previous executable tests obscures the relationship to

logical specifications and can cause crucial post-conditions to be omitted from test specifications.

Accessing the old value in an executable test requires a copy of the initial value that will not be modified. For unstructured data, such as integers, making the copy is simple. For structured data, such as an object with fields, copying the initial value is difficult and can lead to test specifications that incorrectly encode the post-conditions. The object-copy should copy each field, to identify sub-structure modifications, but heap-level equality is lost.

Given these problems in test specifications for imperative procedures, as well as additional problems caused by non-standard termination (i.e. exceptions), we bypass the programming language and use a specialized extension of Java expressions for test-specification. We extend Java with testing forms based on Java Modeling Language [11] (JML) guarantee specifications. Our compiler generates Java programs from test specifications that provide references to both unmodified and modified versions of objects using software transactional memory (STM) [12].

Snapshot Testing

In STM, modifications to a transactional variable affect the value read for some clients while others read the initial value. Modifications to the transaction are stored until a command commits them to the value, exposing them to all clients, unless a conflict causes the modifications to be aborted. We use these techniques to isolate modifications to our initial value instead of copying structural data.

Consider a method that imperatively inserts an item into a list with the post-condition that after insertion the list is a strict superset of the initial list. Copying the list requires making a copy of each element of the list, which may themselves be complex data structures. A copy can be memory intensive, may require duplication of externally-shared resources that should not be copied (such as network sockets), and prohibits pointer-equality comparisons. For some post-conditions, such a copy may invalidate the test.

Instead of copying values, we introduce the idea of *snapshot tests* that automatically preserve the original data structure while also supporting access to the modified version. In databases and file-system backups, access to older versions of the system are provided through snapshots, inspiring our name. Due to storage space involved, it is infeasible for these systems to perform a bit-for-bit copy and instead store a delta of the changes made. Similarly, our snapshots store the modifications made to the value during a test and we treat the original value as a transaction to record the modifications. During the test command, reads to the snapshot value use the transaction's record (a 'log'). The post-conditions can read either the log or the original value. Subsequently, we commit the changes.

Logical Specifications and Assertions

Hoare-style logical annotations have been incorporated into Java both as formal specifications (i.e. using JML), and assertions. While these specifications, as generalized descriptions of a method's behavior, complement verification

techniques, they do not fill all of the needs of a test. A test sets out all specific initial conditions for evaluation and concisely connects these to the specific outcomes; a formal specification does not necessarily provide sufficient information to evaluate a command.

While JML can be used for verification and static analysis, such as in tools like ESC/Java [7], it also comprises the syntax of tools that convert the specifications into runtime assertions. These tools have taken the first steps toward integrating formal specifications and execution-based validation. However, their formal specifications do not support all of the standard aspects of test development; nor do these implementations fully support accessing initial values after a mutation.

The Jass Java-extension [2] embeds a pre- and post-condition language into Java comments, using a subset of JML. These conditions are compiled into assertions within the method; however, a user must still provide specific method calls in a separate location to create tests. The separation between method call and post-conditions increases development efforts and reduces the clarity of a test. Jass specifications support access to both the initial and modified values bound to a variable, using an `Old` designation. However, the implementation for this feature uses the Java `clone` method, which duplicates top-level field bindings but does not necessarily recur into structures to clone the reachable data. Depending on the semantics of any comparisons, this clone can either not provide sufficient duplication or can break pointer-level comparisons. Our snapshot representation dynamically preserves both qualities, so that the test specification can freely describe the expected behavior.

Another similar system, the Cheon and Leavens [6] system, generates JUnit test specifications based on JML specifications but requires programmers to modify the generated test cases with specific parameters and sometimes results, also provides connections between formal specifications and runtime validation. However, like the Jass implementation, this system does not support testing imperative methods.

Existing hybrid systems either preclude the concrete executions of a test case or split test specification into multiple locations. Also they do not properly support the semantics of accessing the initial data after an imperative method. Our work builds on this background to provide a specification framework that matches standard test development and supports more accurate specification semantics.

Roadmap

Our test specification forms, presented in Section 2, allow test specifications to support both logically valid assertions with specific executions, combining the best of existing formal specifications and validating test environments. We demonstrate the ease of specification, compared with the industry-standard test specification library JUnit in Section 3. With our specialized test specification forms, we can provide additional information regarding test evaluation and failure than standard test engines, described in Section 4.1. Section 4.2 outlines compiling these test specifications into standard Java expressions, with a specific emphasis on compiling snapshots in Section 5.

2 Test Specifications

Hoare-logic based specifications, as typified by JML, annotate method definitions with requirements and guarantees. These specifications encode a generalized view of the method's behavior. In contrast, a typical test specification examines a concrete evaluation of a method call and encodes the specific behavior for the circumstance. Due to the different requirements, the test specifications must examine individual evaluations – typically a unit test examines one method call, so we present this circumstance.

We present our test specifications relative to a class modeling a board game

```
class Board {
    Board(String size, Piece initial) ...
    void slide(Piece p, char d) ...
}
abstract class Piece { Posn left; }
```

A typical test first establishes a set of bindings – the precondition.

```
Piece p = new LShape();
Board b = new Board("small", p);
```

A test precondition comprises a set of variable declarations and data initializations, as above. A test *command* uses the variable declarations

```
b.slide(p, 's')
```

Following this imperative method call, the test specification confirms that necessary post-conditions are met; our extended Java expression language encodes these specifications in JML-style syntax, for example

```
modifies(b) && old(p).left.y == p.left.y + 1
```

This *TestExpression* ensures that the `slide` method changed the internal board structure and updated the specified piece's left corner to a lower coordinate.

To deal with imperative test specifications, *TestExpressions* extend the Java expression form with three variants, see Figure 2¹. Each *TestExpression* produces a boolean indicating the success or failure of the individual post-condition. The `old` designation allows the specification to access the snapshot of value bound to the specified variable; the `modifies` predicate determines whether the test command has performed any mutations involving the value bound to the specified variable; and the `modifiesOnly` predicate determines if the test command has performed a mutation involving only the specified fields of the specified variables, leaving other bindings unmodified.

Each *TestExpression* evaluates with respect to a particular tested command, in our example the `slide` method call. To identify the tested command, our expression test form combines the command with the post-conditions, e.g.

```
b.slide(p, 's') ensure modifies(b) && old(p).left.y == p.left.y + 1;
```

¹ We accept any Java expression with nested *TestExpressions* as a *TestExpression*.

```

TestExpression ::= [TestExpression / Expression]
                  | old(Variable)
                  | modifies(Variable)
                  | modifiesOnly(Variable.Name[, Variable.Name]* )
                  | throws Name
Expression ::= ...
              | Expression ensure TestExpression

```

Fig. 1. Test Specification Forms²

The infix³ **ensure** keyword is drawn from the JML keyword for method requirements, and identifies the scope of the post-conditions. The *TestExpressions* include the existing expressions, where *TestExpressions* can occur within the expressions, and each produce booleans. The **throws** form of *TestExpression* determines whether the tested command terminated with the named expression, while the others provide a means of describing modifications to data structures. Further post-conditions can be evaluated in conjunction with a tt throws test, effectively merging the **ensures** and **exsures** keywords from JML.

2.1 Language Additions

While the *TestExpressions* theoretically suffice to encode test specifications, for practical purposes (including error reporting) we further extend the Java syntax to support test development. We also provide test-specific top-level grouping constructs to simplify test specifications, discussed in greater detail in Section 3.2, as well as further *TestExpression* forms that provide more convenient specifications. Figure 2 presents all of our additions.

```

TopDef ::= ...
          | test Variable { DefMember }
DefMember ::= ...
            | testcase Variable { MethodBody }
Expression ::= ...
              | Expression ensure TestExpression'
TestExpression' ::= [TestExpression' / TestExpression]
                  | TestExpression
                  | result
                  | TestExpression' === TestExpression'
                  | TestExpression' memberOf Expression

```

Fig. 2. Test Specifications

The **result** binding, following the JML specifications, provides the result value of the test command on the right-hand side of the **ensure** expression. We

² The form $[a/b]$ denotes all the right-hand sides of productions for b , but with occurrences of b replaced by a .

³ **ensure** follows the same precedence as the $?$ operator in conditional expressions.

restrict binding local variables within a *TestExpression* for clarity and compilation concerns, therefore we must automatically provide a binding for this value. A *TestExpression* may only occur within a `test` definition. The final two forms perform common comparisons; a structural comparison of two values using `===`, and a comparison between the stated value and one of an array of values in `memberOf`. Programmers could write these specifications themselves, but it is convenient to incorporate such forms into the language.

3 Comparison with JUnit

Encoding tests as Hoare-style specifications rather than as stylized calls within a test suite leads to clearer test specifications as well as greater connections between executable validation techniques and formal verification. To explore the improvements for specifications, we compare our test specifications with test specifications written in the most prevalent testing package for Java, JUnit. These sample test specifications evaluate an extension of our board game example from Section 2.

3.1 Individual Tests

A JUnit test specification uses assertion methods, such as `assertEquals`, to validate test properties, often augmented with logging information to aid in test reports. These methods are provided from a class `Test`, that programmers extend. Programmers may add specialized assertion methods that combine the initial assertion methods.

Comparing Non-Structured Data. Comparing non-structured values, such as integers, in JUnit can use the standard assertion method. This method accepts two values and uses either numeric equality or the Java `equals`, which defaults to pointer equality, method to compare them.

```
assertEquals("board size", b.size(), 30);
```

The `Test` class provides an appropriate `assertEquals` method for all values.

Comparing values in *TestExpressions* uses `result` to access the test command's returned value.

```
b.size() ensure result == 30
```

Test specifications may require multiple checks on a given value, in which case the JUnit test must also store the resulting value in a variable, for example when a partly random result is expected:

```
int size = b.size();
assertTrue("board size", (size > 20 && size < 50));
```

versus our specification

```
b.size() ensure (result > 20 && result < 50)
```

These simple specifications are not significantly different in either implementation, due to their simplicity. As a language extension, failure reports between our system and JUnit differ, these differences are discussed in Section 4.1.

Checking Imperative Methods. With imperative methods, comparisons in JUnit test specifications require the programmer to either explicitly copy an existing value or manually compare individual fields to constant values. The following example uses both techniques

```
Piece p2 = new Square();
Board bOld = copy(b);
b.place(p2, 7, 9);
assertTrue(bOld.grid.subset(b.grid));
assertEquals(p2.left.x, 7);
assertEquals(b.numMoves, bOld.numMoves);
assertEquals(b.maxSide, bOld.maxSide);
...
```

This test specification correctly validates the performance of `place` provided a) the `copy` procedure properly copies all the field values so that the two boards do not share values, particularly the grid, b) the `copy` method preserves the necessary information for the other fields to satisfy the equality methods, and c) all of the fields are listed. As a program develops over time, the likelihood of the test correctly validating the method decreases as fields are added and implementations change.

Our specification uses a snapshot of `b` and `p2` to validate the post-conditions.

```
Piece p2 = new Square();
b.place(p2, 7,9) ensure (modifiesOnly(b.grid, p.left) &&
                        old(b).grid.subset(b.grid))
```

The `modifiesOnly` form checks that for the listed bindings, namely `p` and `b`, the referenced fields are modified without changes to any other fields in these objects. The `old` form allows the specification to access the initial grid value.

Checking for Exceptions. Testing methods using JUnit that may cause an exception requires either explicitly placing the method call in a `try` block or passing any exception along to the test engine. For simple tests and intricate interactions, this can be too large a burden for programmers to implement correctly.

The `ensure` expression removes the need to explicitly handle exceptions within the specification, with the `throws` test clause providing a per-call means to evaluate error handling implementations.

In JUnit, a test specification that anticipates an exception can omit a `try` block if restricted to one method call and exception per test case, as shown in the following example

```
@Test(expected = IllegalMove.class) void place() throws IllegalMove {
    Board b = new Board("small");
    b.place(new Square(), -10, 0)
}
```

The annotation at the method definition, contained within the test class, indicates that an exception must halt the execution of this test method and any

other behavior is an error. Grouping this test with previous test specifications manipulating the board is problematic; additionally, testing that a method has performed any mutations prior to raising an exception cannot be handled in this style.

To show the potential problems caused in using JUnit, the following protocol⁴ requires that multiple exceptions be tested within one method, to ensure that proper side-effects occur during exception handling.

```
Piece p1 = ..., p2 = ...; Posn c = ...;
b.place(p1,c);
try {
    b.place(p2,c);
} catch( ContestedPosition e) {
    try {
        b.location(p1);
    } catch( UnplacedPiece e) {
        return;
    }
    fail("UnplacedPiece not thrown");
}
fail("ContestedPosition not thrown");
```

The second call to `place` attempts to overlay one piece on another, which causes an exception and changes the state in `p1` and `b`. The call to `location` should now fail due to the state changes caused in modifying the board due to the encountered error. Correctly developing such nested `try` blocks can lead to errors with omitted `return` statements or misplaced calls.

An equivalent test specification using our `ensure` expressions follows

```
Piece p1= ..., p2= ...; Posn c = ...;
b.place(p1,c);
(b.place(p2,c) ensure throws ContestedPosition &&
 b.location(p1) ensure throws UnplacedPiece)
```

This eliminates the need for nested `try` statements and the second test specification clearly relies on the first.

Comparing Objects. For object comparisons, JUnit's comparison method uses the inherited `equals` method, which does not always perform the necessary structural comparison, so the programmer must develop an independent comparison. These cases are most problematic with arrays and with classes without source, where a structural comparison is necessary but unavailable. In JUnit, such a comparison follows

```
boolean comp(Piece[] a1, Piece[] a2) {
    boolean res = a1.length == a2.length;
    if (res)
        for(int i; i< a1.length; i++)
```

⁴ Inspired by a test seen in an open-source text-editing project.

```

        res &= a1[i].equals(a2[i]);
    return res; }

```

```

    assertTrue(comp(b.getPieces(3), new Piece[]{new LShape(), ...}));

```

The `comp` method correctly assess whether the two `Piece` arrays are equivalent. Depending on the comparison method for a `Piece`, different array comparison methods may be required. The `===` comparison simplifies these test specifications by providing a structural comparison for any two values.

```

    b.getPieces(3) ensure result === new Piece[]{new LShape(), ...}

```

Comparisons of objects with private fields highlights an additional benefit offered with `===`; writing such a comparison can be problematic for a programmer, relying on reflection and security accesses. We leverage compile-time information to provide a structural comparison in all circumstances.

3.2 Test Organization

Test specifications typically occur in a separate Java package from the primary implementation, following the JUnit style. Individual classes are tested by an extension of the JUnit `Test` class that contains a set of methods which each test the behavior of a particular method in the implementation class. Thus in our example, a test-specification `Board` class would extend the `Test` class and primarily test the operations within the `Board` implementation.

We mirror this organization within our testing specifications, to accommodate the expectations of test developers. Instead of deriving a particular class, however, we provide a third top-level form `test` that serves as the grouping mechanism for testing class implementations. The additional form allows us to restrict the placement of the `ensure` keyword, so that standard Java programs are unaffected.

The methods within a JUnit test class are annotated with an `@Test` attribute, signaling that these methods test a particular facet of the implementation. These methods must take no parameters and return no values (conditions which are checked via reflection at runtime).

We again mirror this organization, but use a specialized form that omits the possibility of dynamic signature errors. Test methods use a `testcase` modifier, a la `abstract`, and cannot specify attributes. A `testcase` may only appear within a `test`.

These additional forms provide static checks of test organization while not requiring test developers to modify their test organization strategies.

4 Implementation

The *TestExpression* and enclosing `test` forms can all be compiled to standard Java and during compilation can be automatically integrated with a report mechanism. This further simplifies test development and allows a program to be tested on a standard JVM implementation.

4.1 Integration with Test Reports

Each compiled *TestExpression* includes source information as well as annotations that provide further tools for evaluating test performance. Using the source information, a failing test report can identify the tested command and any values involved in the computation as well as the nature of the failure, for example if a method triggers a different exception than one declared in a **throws** clause. This information can aid the programmer in eliminating mistakes, in either their specification or the program. Using a system like JUnit, programmers manually annotate test specifications with this information, complicating the development.

By identifying a test specification, with the combination of the **test** and **testcase** designations and the **ensure** keyword, we can provide test-specialized coverage information when compilation is extended with coverage-tracking features. The compiler identifies the start of each test and inserts calls to select the coverage information collected during evaluation of the test command, or of the **testcase**. This information can be used to assess the scope of a test, aiding in debugging failed tests and determining the benefit of more test development. Other analyses, such as memory accounting, could also be automatically incorporated into test suites to improve program assessment.

4.2 Implementation

The **ensure** expression compiles into a Java method call that returns a boolean value, with the two expressions involved compiled into an anonymous inner class that evaluates the test command and the post-conditions. Figure 3 contains the compilation target for a general **ensure** expression, where the test command generates a value. Other than **result**, the variables within the **value** method are fresh to avoid name capture.

Targeting an anonymous class allows the tested command to expand into a sequence of statements that initialize the snapshots used within the assessment. Additionally, this expansion permits the test command to evaluate within a controlled framework where exceptions can be caught and evaluations can be run in a separate thread, controlled by the **addTest** method. Due to the inner class, local variable declarations must be treated as final within the *TestExpression*, so that these values can be passed into the inner class. This safety restriction does not restrict program validation, as mutating abstract variables within the post-condition does not provide validation on the correctness of the test command.

Each *TestExpression* compiles into an expression that evaluates the condition and provides information to the test report engine. The comparison expression expands into a method call to a comparison function within the test harness that inspects all of the fields of an object, whether private or accessible; the throws expression expands into an **instanceof** statement using the specified class and the **exn** binding. The expansion for the snapshot-specific expressions, are addressed in Section 5 with the explanation of how a snapshot is implemented.

A standard test specification

`Expr1 ensure Expr2`

compiles into

```
test.addTest( new TestClosure() {
    public boolean value() {
        Object result = null;
        boolean ans = false;
        Throwable exn = null;
        << set snapshots as indicated by Expr2 >>
        try {
            result = Expr1;
        } catch (Throwable t) {
            exn = t;
        }
        ans = Expr2 << with compiled TestExpressions >> ;
        << unset snapshots >>
        return ans;
    }})
```

Fig. 3. Compilation Template for **ensure** with Value Generating Test

5 Snapshot Tests

Our snapshots mimic the effect of creating a copy in memory, but do not actually copy any values, which preserves the semantics of both pointer-level equality and structural comparisons. We divert mutation operations into a per-object log to preserve the original data-structure, and redirect accesses to the log during tests.

5.1 Taking a Snapshot in Restricted Java

Redirecting field accesses can be more easily explained in a restricted subset of Java than in the full language; therefore, we initially prohibit field accesses outside of specialized methods – fields may only be read in a field-specific **get** method and may only be modified in a **put** method. All methods, including the constructors, may only use these accessor methods when referring to a field binding. Both methods may read one additional field we add to each object – a boolean **stmOn**⁵ field, with initial value of **false**.

While **stmOn** is **false**, field reads and writes proceed as normal. Using the old designation or either modification predicate changes the referenced object's **stmOn** to **true** prior to evaluating the test expression. While **stmOn** is **true**, field reads and writes pass through a log and do not modify the binding.

We represent the modification log with a hash-table, where the field name (combined with the class) is the key. On calls to the **put** method, the hash-table entry is updated with the provided value for the field binding. On calls to the **get**

⁵ The **stmOn** field's actual name is generated to avoid collisions.

method, the field value is read when no entry exists in the hash-table, otherwise the hash-table value is used.

Before returning from the test call, all snapshots are reverted to normal by changing the `stmOn` to `false` and the modifications contained in the hash-table are committed to the respective field values.

This solution correctly redirects field accesses for object snapshots when all fields refer to unstructured values (i.e. integers, characters, or booleans). However, when a field refers to a structured value and the test command modifies the structure of this value (i.e. a modification of the form `p.left.y = 5`, where `p` is from our previous examples), the modification has not passed through a log as the `left` field is not a snapshot.

We must prohibit modifications on a snapshot from affecting the value until after the test, therefore we propagate the modification to `stmOn` on each initial read of a field value of a snapshot. On the first access of a field binding within a test, the value referred to by the field becomes a snapshot and modifications cannot affect the stored value. Further on each call to `put`, if the provided value is not yet a snapshot, `put` first sets the `stmOn` before storing the value.

While this Java subset is too restrictive for standard Java programs, a translation from any Java program into this subset is straightforward – the primary concern is to select a fresh name for the field methods and to preserve the shadowing of inherited fields.

5.2 Taking a Snapshot with Reduced Costs

Using a method call to access field values is not uncommon in Java programs and can typically be in-lined by an optimizing compiler, removing dispatch overhead. However, with the addition of the `stmOn` parameter, an optimizer may not be able to identify that field accesses can be safely in-lined. Thus the compilation strategy for taking a snapshot could negatively impact performance of all programs – this should be avoided.

For each field, compilation generates the accessor methods described above but only includes the implementation for the case where `stmOn` is false. This permits an optimization pass to remove the indirection for all field accesses. We incorporate the snapshot log by extending each class with a test-aware version. These classes each override the accessor methods of the original class with the body contents described in Section 5.1. The next step is in replacing instances of the original class with instances of the test-aware class in snapshot-contexts.

If we replace all constructor calls in the test program with their test-aware counterparts, than any objects initialized within the test specification can be preserved within a snapshot. However, this does not affect constructor calls not made within the test but made externally, such as a constructor that itself calls other constructors to initialize internal state. Therefore, without whole-program modifications, we cannot redirect the constructor calls and thus cannot turn every object into a snapshot-variant using this strategy.

A snapshot reflects the state of a value, without regard for the origin of this value. Therefore our snapshot-aware class extensions embed an instance of their parent class; each field access method dispatches to the embedded class instance and other methods defer to the super class. During a field access, the snapshot implementation defers to the test-aware field implementation, while protecting the initial value, while dynamic method dispatch selects the correct implementation to test.

For each value requiring a snapshot, we modify the binding to refer to the test-aware extension of the class with the initial value embedded within. Due to dynamic method dispatch, we cannot statically determine the test-aware class to instantiate. We use reflective techniques to identify the class and create the instance dynamically.

Uses of `old` within the post-conditions translate into an access of the embedded value. The `modifies` and `modifiesOnly` predicates translate into method calls which examine the hash-table of the specified snapshot. By examining the log, we differentiate unmodified values from a modification that has restored the original value before termination. Before returning, we record the modifications found in each hash-table into the embedded value to commit the changes.

5.3 Supporting Binary Libraries

The previous translation converts source files, ignoring (JVM) binary compilations. We advocate compiling from source where possible, to provide source-level information in test reports and analyses; however, this cannot always occur when using external (pre-compiled) libraries. External libraries must be made test-aware through byte-code transformations to redirect field accesses. We must rely on aspect-oriented style rewriting to modify the `getfield` and `putfield` byte-code instructions into appropriate method calls; however, this modification has not yet been implemented.

Using an aspect-oriented ‘advice’ to generate test-aware byte-code highlights the similarity between our transformations, which inject transactions into existing Java applications as well as test-report monitors, and generalized aspect-oriented programs that inject additional functionality into existing programs.

5.4 Problems with Snapshots

Due to the indirections on field accesses and the additional memory consumption, using snapshots has an impact on the performance of tested methods. While the run-time impact of snapshots has not been measured, we believe that any run-time overhead should be minimized to avoid increasing the cost of test suite evaluation. In future versions of our compiler, we intend to use a combination of ownership types and liveness analysis to determine whether each field in a class requires a snapshot to accurately validate a method’s performance. When no snapshot is necessary, we will either use a copy or ignore the variable.

6 Related Work

In addition to the existing work joining executable validation with formal specification discussed in Section 1, much effort has been put into improving the organization and execution of tests. An early effort in this regards is the SUnit [3] system for Smalltalk, which supports organizing groups of related tests into classes with individual test methods containing multiple test cases evaluating a single implementation method. Individual evaluations use assertion methods, that may compare or assess values. Libraries following the SUnit philosophy now exist to support similar organizations for most programming languages, including JUnit [4] for Java, SchemeUnit for Scheme [13], and LIFT for Lisp [10].

JUnit provides a library of assertion methods, derivable classes, and an integrated report interface. New tests extend a class from the JUnit library and contain test methods, indicated either with a `@Test` attribute[9] or by appending `test` to the name of the method. Programmers use `String`-valued fields and parameters to document test cases. Both the SchemeUnit and LIFT systems use macros to extend the language with specific test forms, improving error reporting. Both provide language forms for test specifications, but encodings for imperative procedure remain difficult.

More modern efforts, such as TestNG [5], attempt to refine the test organization strategies of JUnit to support flexible test evaluation at the level of test methods but do not provide additional support for test specifications within the language as we do.

As with the macro-based libraries, the jMock [8] test engine embeds a test specification language into Java – via strings. These test specifications do not provide support for imperative post-conditions and the form of embedding can increase the difficulty of reading the test suites, as the programmer must distinguish between evaluated strings and documentary strings.

The Fortress programming language [1] contains built-in forms for declaring test procedures. An individual test case is marked using a `test` modifier, and a library provides functions to report and terminate failing tests. This language-based support does not extend to representing the test specifications, only benefiting the test organization and execution.

7 Conclusions and Further Work

We noted that existing unit test specifications correspond to restricted Hoare-logic formulae. We then observed that a more relaxed set of post-conditions, including `old(X)` (value of `X` during the precondition), `throws` (holds when an exception is thrown) and the like greatly simplify the code required for tests (using JUnit as a base for comparison), especially for imperative methods, which modify structures in-place. The extended forms of expressions *TestExpression* used in post-conditions can be implemented using a variant of Software Transactional Memory (STM).

While the mapping of our unit test constructs can be seen as a simple extended Java-to-Java translation, we show that this translation can also be done at the JVM level for pre-compiled libraries.

The system described is implemented in two systems, one with an alternate syntax and restricted functionality, which can both be downloaded from www.professorj.org/testing.

More expressive Hoare-logic formulas could also correspond to test specifications (for example, universal quantification might reasonably be treated as random testing) to further enhance test development. We leave this to future work.

Acknowledgements. We thank Matthias Felleisen for helpful conversations as we began our explorations of test development. This work was supported by (UK) EPSRC grant GR/F033060 “Linguistic Support for Test Development”.

References

1. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G.L., Tobin-Hochstadt, S.: The Fortress language specification. Technical report, Sun (2007)
2. Bartezko, D., Fischer, C., Moller, M., Wehrheim, H.: Jass – Java with assertions. In: Workshop on Runtime Verification (2001)
3. Beck, K.: Simple smalltalk testing with patterns. The Smalltalk Report (1994)
4. Beck, K., Gamma, E.: Test-infected: programmers love writing tests. Java Report (1998)
5. Beust, C., Suleiman, H.: Next Generation Java Testing: TestNG and Advanced Concepts. Addison-Wesley, Reading (2007)
6. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, p. 231. Springer, Heidelberg (2002)
7. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. PLDI (2002)
8. Freeman, S., Pryce, N.: Evolving an embedded domain-specific language in Java. In: Proc. OOPSLA (2006)
9. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley, Reading (2005)
10. King, G.: LIFT — the lisp framework for testing. Technical Report 01-25, U. Mass. CS (2001)
11. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design, ch. 12. Kluwer, Dordrecht (1999)
12. Shavit, N., Touitou, D.: Software transactional memory. In: Principles of Distributed Computing (1995)
13. Welsh, N., Solsona, F., Glover, I.: SchemeUnit and SchemeQL: Two little languages. In: Proc. Scheme Workshop (2002)

Cross-Entropy-Based Replay of Concurrent Programs^{*}

Hana Chockler¹, Eitan Farchi¹, Benny Godlin¹, and Sergey Novikov^{2,**}

¹ IBM Haifa Research Laboratories
Haifa University, Mount Carmel
Haifa 31905, Israel
hanac, farchi, godlin@il.ibm.com
² Department of Computer Science
Weizmann Institute, Israel
sergey.novikov@weizmann.ac.il

Abstract. *Replay* is an important technique in program analysis, allowing to reproduce bugs, to track changes, and to repeat executions for better understanding of the results. Unfortunately, since re-executing a concurrent program does not necessarily produce the same ordering of events, replay of such programs becomes a difficult task. The most common approach to replay of concurrent programs is based on analyzing the logical dependencies among concurrent events and requires a complete recording of the execution we are trying to replay as well as a complete control over the program's scheduler. In realistic settings, we usually have only a partial recording of the execution and only partial control over the scheduling decisions, thus such an analysis is often impossible. In this paper, we present an approach for replay in the presence of partial information and partial control. Our approach is based on a novel application of the cross-entropy method, and it does not require any logical analysis of dependencies among concurrent events. Roughly speaking, given a partial recording R of an execution, we define a performance function on executions, which reaches its maximum on R (or any other execution that coincides with R on the recorded events). Then, the program is executed many times in iterations, on each iteration adjusting the probabilistic scheduling decisions so that the performance function is maximized. Our method is also applicable to debugging of concurrent programs, in which the program is changed before it is replayed in order to increase the information from its execution. We implemented our replay method on concurrent Java programs and we show that it consistently achieves a close replay in presence of incomplete information and incomplete control, as well as when the program is changed before it is replayed.

1 Introduction

Software testing and debugging are nowadays the primary means of checking the correctness of programs. Repeated re-execution, or *replay*, is a widely accepted technique

^{*} This work is partially supported by the European Community under the Information Society Technologies (IST) program of the 6th FP for RTD - project SHADOWS contract IST-035157. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

^{**} Most of this work was performed when author was at IBM Haifa Research Laboratories.

for debugging sequential deterministic programs. In these programs, a specific input vector always generates the same execution, so the task of replay is relatively easy, provided that the input vector is recorded. This approach, of recording the input vector and re-executing the program, does not work on concurrent programs, where the same input vector can generate many executions of the program, as a result of the different scheduling of concurrent events. *Replay of a concurrent execution from its recorded trace* is a widely used and extensively researched approach for analyzing and debugging concurrent behaviors. To overcome the problem of different scheduling, replay of a concurrent program is achieved by first recording an execution, and then enforcing the same order of events during the replay.

Existing work on replay of concurrent programs is based on analyzing the logical order between concurrent events and assigning time-stamps to events that need to be executed in a particular order. The idea of applying time-stamps to express logical order between events was first introduced in the seminal work of Lamport [15]. Lamport introduced the *happened-before* relation between concurrent events, which imposes a partial order on these events. A system of logical clocks is then used in order to capture this relation by assigning time-stamps to all concurrent events in the computation, consistent with the logical dependencies between these events. The computation of time-stamps for ordering of concurrent events is studied and improved upon in several works (see, for example, [3,14,21,18,19,1]), where the order of events determined by time-stamps is used for replay of programs. Replay of multi-threaded Java programs, which are inherently concurrent, is studied by Choi et al., who suggest methods for efficient recording of concurrent events and control of the Java scheduler in order to replay the recorded execution (see, for example, [5]). Here also, the logical dependencies between events are analyzed in order to construct a correct logical schedule, which is then used for replay.

The major drawback of the existing approaches to replay of concurrent programs is that they all study *deterministic replay*, that is, replay that has full control over the scheduler and follows a complete recording of a previous execution. In order to implement deterministic replay, it is necessary to record the events that are crucial for defining the logical order between all concurrent events. There are several works that attempt to reduce the size of the trace as much as possible [16,20]. However, there exists a minimal subset of events that is crucial for the correct logical order, and this subset has to be recorded in order to allow replay. Moreover, in replay, there has to be a full control over the scheduler, since even a slight change in the order of concurrent events can lead to deviation from the recorded execution and inability to resume replay. In realistic settings, both requirements, of full recording and full control over the scheduling, are often impractical. Software today incorporates third party code and library code of which we have no control, code is loaded at runtime, and due to standardization and open environments it is impossible to have control over all concurrent events. Moreover, due to space and performance considerations and the fact that often recordings of executions are created by the customer, getting a recording of an execution that is complete enough to determine logical order is unlikely. Finally, these approaches do not address a common debugging scenario, in which the user adds informative statements to the program before replaying it (usually print statements). These statements

affect the existing happened-before relation and the control over the program, and thus the program cannot be replayed based on the previously computed order of concurrent events.

In this paper, we study the problem of *approximate replay*, that is, replay in presence of incomplete recording and only partial control over the concurrent events, and where a program under test is modified before it is replayed. We present a novel approach to replay that is based on the cross-entropy method and the adaptation of this method to testing of concurrent programs. The *cross-entropy* (CE) method is a generic approach to rare event simulation [24]. It derives its name from the cross-entropy (or the Kullback-Leibler distance), which is a fundamental concept of modern information theory [13]. It is an iterative approach based on minimizing the cross-entropy or the Kullback-Leibler distance between two probability distributions. The CE method was motivated by an adaptive algorithm for estimating probabilities of rare events in complex stochastic networks [22]. Then, it was realized that a simple modification allows to use this method also for solving hard combinatorial optimization problems, in which there is a performance function associated with the inputs. The CE method in optimization problems is used to find a set of inputs on which the performance function reaches its global maximum, where the input space is assumed to be too large to allow exhaustive exploration. The method is based on an iterative sampling of the input space according to a given probability distribution over this space. Each iteration consists of the following phases:

1. Generate a random sample of the inputs according to a specified probability distribution.
2. Update the parameters of the probability distribution based on the sample to produce a “better” sample in the next iteration, where “better” is chosen according to the predefined performance function.

The initial probability distribution is assumed to be a part of the input. A sample is evaluated according to a predefined performance function. The procedure terminates when the “best” sample, that is, a sample with the maximal value of the performance function (or, if the global maximum is unknown in advance, with a sufficiently small relative deviation), is generated. The CE method is used in many areas, including buffer allocation [2], neural computation [7], DNA sequence alignment [10], scheduling [17], and graph problems [23].

In [4], we adapted the cross-entropy method to testing of concurrent programs. Informally, such programs induce a large control-flow graph with many branching points, which allows us to view the testing setting as a variation of a graph optimization problem, with the input space being the space of all possible paths on the graph. While a serialized program can, in theory, have many points with non-deterministic decisions (for example, statements conditional on the result of coin-tossing), the most common example of such programs is concurrent programs. In concurrent programs, decisions about the order of execution of concurrent threads are made by the scheduler, and thus can be viewed as non-deterministic when analyzing the program. Our tool, ConCenter, is based on the cross-entropy method, and was shown to be effective in finding rare bugs in multi-threaded Java programs. The most natural initial probability distribution over the space of all possible paths (and the one we use in [4]) is the uniform distribution

over edges on each node, meaning that at each decision point each enabled thread can make a step with equal probability.

In this paper, we adapt the cross-entropy method to replay of concurrent programs as follows. We define the *distance* $D(e_1, e_2)$ between two executions e_1 and e_2 in a way that reflects the distance between e_1 and e_2 on the control graph of the program. Then, we define the performance function $S(e)$ over executions as $-D(e, e_{rec})$, where e_{rec} is the recorded execution. Since the distance is always non-negative, the performance function $S(e)$ reaches its global maximum on executions that are as close to the recorded execution as possible. Then, we apply the cross-entropy-based testing method (implemented in ConCEnter) in order to get an approximate replay of the recorded execution. In our approach, having only a partial recording of the execution or not having full control over the executions is handled naturally without any adjustments. The distance is computed based on the recorded events, and the probabilities of concurrent events over which we have control are adjusted according to the cross-entropy minimization computation. Moreover, our experimental results show that our method works even if the program is changed before replaying it, for example, by adding print statements (a common scenario for debugging).

As we discuss in Section 4.2, in replay we can use the recorded execution in order to compute an initial probability distribution over the space of all executions in a way that significantly improves the running time of ConCEnter compared to executing it on the same examples with the initial uniform distribution. We give edges that appear in the recorded execution the initial probability that is higher than their probability under the uniform distribution, and we adjust the probabilities of other edges accordingly.

We discuss the problem of losing diversity of the sample and the reasons why this problem is more pronounced in replay than in applications of cross-entropy based testing to bug searching. Essentially, the performance function based on the distance from a recorded execution has a very narrow global maximum (in fact, when the complete recording is available, the global maximum is reached on exactly one order of recorded events). This particular shape of the performance function increases the probability of missing the global maximum by converging to a local maximum or even to some random point. This problem, of losing diversity of the sample (and as a result, converging to a wrong point), is inherent for the cross-entropy method as well as for other methods that use iterative adjustments of the probability distribution, and it becomes more pronounced in replay, compared to testing for rare bugs. To overcome this problem, we use *random injection* at some of the iterations of the cross-entropy execution. Our method is inspired by *simulated annealing*, introduced in [11]. Simulated annealing is a general approach to finding good approximations for global optimization problems of functions in large search spaces. Essentially, in each iteration, simulated annealing replaces the current solution by a random “nearby” solution, chosen with a probability that depends on the number of previous iterations. The randomization decreases during the process, thus allowing it to converge. Roughly speaking, simulated annealing slows the convergence process by introducing an additional dimension of randomness. The effect is two-fold: first, by slowing the convergence process, it increases the probability of it converging to the correct maximum¹; and second, by introducing more

¹ The reason for this phenomenon in simulated annealing is the same as in cross-entropy.

randomness, it increases coverage of the search space, thus increasing the probability that a good solution will be eventually drawn. We achieve a similar effect by introducing random injection at some iterations, where the decision of whether to introduce a random injection depends on the relative standard deviation of the current sample.

We implemented replay with random injection in ConCenter and tested it on multi-threaded Java programs². Our experimental results show that we are able to efficiently produce a close replay even when we have only a partial recording of the execution. We compare the performance of ConCenter with the performance of ConTest, a randomized tool for testing of concurrent programs developed at IBM [8].

2 Preliminaries

2.1 The Cross-Entropy Method in Optimization Problems

In this section we present a brief overview of the cross-entropy method for optimization problems. The reader is referred to [4] for more formal explanation and to the book on cross-entropy for the complete description of the method[24]. In our setting, we use the application of the cross-entropy method to graph optimization problems.

The cross-entropy (CE) method for optimization problems searches for a global maximum of a function S (called a *performance function*) defined on a very large probability space. Since the exhaustive search is impossible due to the size of the space, the method works in iterations, each time drawing a sample from the space and adjusting the probability distribution for the next iteration according to the values of S on the sample.

In graph optimization problems, we are given a (possibly weighted) graph $G = \langle V, E \rangle$, and the probability space is defined on the set of paths in G represented by the sets of traversed vertices. The probability distribution is defined by assigning probabilities to edges or vertices of the graph (depending on how the sample is drawn). This setting matches, for example, the definitions of the traveling salesman problem and the Hamiltonian path problem in the context of CE method. This is also the setting which we use in this paper.

2.2 Programs as Graphs

We view concurrent programs under test as *control flow graphs*, with nodes being synchronization points. We start with the definition of a control flow graph of a single thread. We define a *program location (PL)* as a line number in the code of the program, and we assume that it uniquely defines the state of a thread. In particular, this means that all loops are unwound to the maximal number of iterations and all function calls are embedded in the code of the main function.³ We use t for the number of threads in the program.

² The executable of ConCenter with some test programs is available from the authors on request.

³ The unwound code of even a small program can be very large. In the previous paper, we reduced the size of the unwound code by using multi-dimensional modulo counters [4]. Unfortunately, when using modulo counters, we lose the one-to-one correspondence between the trace and its representation on the graph. Thus, we do not use this method in replay.

Definition 1 (CFG_i). A control flow graph (CFG_i) of thread i ($i \in [t]$) is a directed graph $G_i = \langle L_i, E_i, \mu_i \rangle$ where L_i is the set of all program locations in the unwound code of the thread, E_i is the set of edges such that $\langle v, u \rangle \in E_i$ if a statement at location u can be executed immediately after the statement at location v , and $\mu_i \in L$ is the initial program location of the thread.

Definition 2 (PLV). Program location vector (PLV) v is a t -dimensional vector such that for each $i \in [t]$, $v_i \in L_i$.

We say that at a given time m during the execution of the program, the execution is at PLV v iff for each $i \in [t]$, v_i is the next program location to be executed in thread i .

Clearly, the set of all PLVs is equal to the cross-product of the L_i s.

Definition 3 (JCG). The joint control graph (JCG) of the program under test is a graph $\langle V, E \rangle$ whose vertices are the PLVs. There is an edge in the JCG between vertices u and w if there exists an execution path in which w is the immediate successor of u .

Note that at each step only one thread executes. Therefore, the branching degree of each vertex is at most t . Since the code is unwound, every statement in it is executed at most once and the statements are executed in the increasing order of their program locations. Therefore, JCG is a finite directed acyclic graph (DAG). The initial node of the JCG is a PLV which is composed of the initial PL μ_i for each thread i .

Definition 4 (PF). Probability function $PF : V(JCG) \times [t] \mapsto [0, 1]$ such that the probability sum over the outgoing edges of each vertex is 1.

This function defines for each vertex v and each of its outgoing edges e_i the probability of the thread i to advance when the execution reaches v . If not all threads are enabled at v , we take the *relative probabilities* of the enabled threads. If $T_{en} \subseteq [t]$ is the set of the enabled threads at this moment then the relative probabilities are:

$$RP(v, i) \doteq \begin{cases} \frac{PF(v, i)}{\sum_{j \in T_{en}} PF(v, j)} & \text{if } i \in T_{en} \\ 0 & \text{otherwise} \end{cases}$$

For a single execution of the program, we call the sequence of vertices of JCG that it visits an *execution path* in JCG.

2.3 The Cross-Entropy Method for Replay

We describe the problem of replay in concurrent programs as a graph optimization problem, where the program is represented as a graph, executions are paths on the graph, and the performance function reaches its maximum on the recorded execution. In replay, an input to the procedure is a (partially) recorded execution. The performance function is based on the *distance* between two executions. Our distance metric is somewhat similar to L_1 distance, also known as *taxicab distance* or *rectilinear distance* (see [12]), and it expresses the distance between the paths on the control graph that correspond to the execution.

Definition 5. For two executions e_1 and e_2 of the same program P represented by paths π_1 and π_2 on the JCG of P , respectively, the distance $D(e_1, e_2)$ between e_1 and e_2 is defined as the number of nodes that are present in one execution and absent from another. Formally, we have the following definition:

1. Let L_P be the vector of all nodes of the JCG of P (note that each node can appear only once in an execution because of unwinding). Let n be the length of L_P .
2. Let $L(\pi_k)$ be the binary vector of length n such that $L(\pi_k)[i] = 1$ iff $L_P[i] \in \pi_k$, for $k = 1, 2$.
3. The distance $D(e_1, e_2)$ is defined as $H(L(\pi_1), L(\pi_2))$, where $H()$ is the Hamming distance (Hamming distance is defined in [9]).

The replay performance function $S^r(e)$ from the recorded execution r to an execution e is defined as $-D(r, e)$. Clearly, it reaches its maximum value 0 at $e = r$. When we have a partial recording of the execution, the distance is measured only with respect to the recorded nodes, even if they do not form a connected path on the graph. Thus, in case of a partial recording, there is a set of traces that have the distance 0 from the recorded execution.

We define the probability distribution on the set of executions by assigning probabilities to edges of the control flow graph. The initial probability distribution is either uniform or biased towards the recorded execution (see Section 3.1 for the discussion on the initial distribution). In each iteration i , we sort the sample $\mathcal{X}_i = \{X_1, \dots, X_N\}$ generated in this iteration in ascending order of their performance function values. That is, $S(X_1) \leq S(X_2) \leq \dots \leq S(X_N)$. For some $0 < q \ll 1$, let

$$Q(\mathcal{X}_i) = \{X_{\lfloor (1-q)N \rfloor}, X_{\lfloor (1-q)N + 1 \rfloor}, \dots, X_N\}$$

be the best q -part of the sample. The probability update formula in our setting is

$$f'(e) = \frac{|Q(e)|}{|Q(v)|}, \quad (1)$$

where $e \in E$ is an edge of the control flow graph that originates in the vertex v , $Q(v)$ are the paths in $Q(\mathcal{X}_i)$ which go through v and $Q(e)$ are the paths in $Q(\mathcal{X}_i)$ which go through e . Intuitively, the edge e “competes” with other edges that originate in v and participate in paths in $Q(v)$. We continue in the next iteration with the updated probability distribution f' . The procedure terminates when a sample has a relative standard deviation below a predefined threshold parameter (usually between 1% and 5%), or an exact replay is achieved.

Smoothed updating and its importance for replay In optimization problems involving discrete random variables, such as graph optimization problems, the following equation is used in updating the probability function instead of Equation 1:

$$f''(e) = \alpha f'(e) + (1 - \alpha)f(e), \quad (2)$$

where $0 < \alpha \leq 1$ is the *smoothing parameter* (clearly, for $\alpha = 1$ we have the original updating equation). According to [6], when the shape of the performance function makes convergence to the global maximum hard, it is advisable to use very low

smoothing parameters, around 0.01, thus decreasing the rate of update of the probability distribution and therefore also the convergence rate. Slowing the convergence rate increases the probability of a sample to find the global maximum, and hence is better for “difficult” functions. In our experiments, setting the smoothing parameter to a very low value increased the running time by several orders of magnitude, thus rendering the method impractical for large programs. In this work, we use the method of random injection (see Section 3.2) to tackle the problem of convergence to local maximum, thus allowing us to use a relatively high smoothing parameter (in this work we are using $0.8 \leq \alpha \leq 0.9$).

3 Algorithm for Approximate Replay of Concurrent Programs

In this section we describe the algorithm that uses cross-entropy for approximate replay of concurrent programs and discuss random injection - the main change from the traditional cross-entropy method.

3.1 Algorithm

Given a (partially) recorded execution r of a concurrent program P , the algorithm outputs a set of executions of P that are the closest to r . In each iteration, the algorithm generates a set of executions based on the probability distribution table. The probability distribution table contains probability distribution on edges and is updated at each iteration. We consider two options for the initial probability table:

1. Uniform distribution, as in other applications of cross-entropy for testing.
2. Probability distribution biased toward the recorded execution. Here, we perform preprocessing of the probability distribution, assigning higher probability to edges that participate in the recorded execution.

In the classic cross-entropy setting, the initial probability distribution on an input space is a part of the input. In our setting, the probability space is on the set of all possible executions of the concurrent system under test, and there is no predefined probability distribution on this space. In [4], we start with the initial uniform distribution over the edges of the control flow graph, since this is the distribution that most accurately reflects the situation where the program runs without any intervention. In replay, however, the situation is different because the recorded execution is a part of the input. Thus, the initial distribution of executions for ConCenter does not need to be uniform – it can be biased toward the recorded execution. The uniform initial distribution has the advantage of not requiring any special preprocessing before the start of the execution. An obvious disadvantage of a biased initial probability distribution is that it requires a preprocessing, however, as we show in Section 4.2, it significantly improves the convergence rate of the algorithm.

We assume that P is partially instrumented (this is needed for recording executions) by adding callbacks at synchronization points. This enables the algorithm to stop the execution at these points and decide which edge is going to be traversed next (that is, which thread makes a move) according to the probability distribution table. At each iteration, the algorithm performs the following tasks:

1. The instrumented program P is executed a number of times sufficient to collect a meaningful sample. Executions are forced to perform scheduling decisions according to the current probability distribution on edges.
2. The executions are used in order to compute the new probability distribution (see Equation 1 and Equation 2).
3. If the current iteration satisfies the criteria for random injection (see Section 3.2), the algorithm computes the probabilities of edges as a weighted average of the probabilities according to the cross-entropy method and a random injection of uniform distribution, where the weight of the injection is between 0.01 and 0.1.
4. The best execution e (the one with the maximum value of $S^r(e)$) is compared with the best execution obtained so far and a new best execution is chosen.

The algorithm terminates when the collected sample of the current iteration has a sufficiently small relative standard deviation (between 1% and 5%), or, alternatively, when we get the best replay, that is, an execution e for which $S^r(e) = 0$. We note that there can be several best replays, depending on the level of instrumentation of the program and the level of control over the executions. Step 4 of the algorithm is needed in order not to miss an exact replay if it is drawn before the best quantile of the sample converges.

3.2 Random Injection

The concept of using random injection in order to prevent convergence to local maximum is not new and is used widely, for example, in simulated annealing [11]. Essentially, without a (sporadic) random injection, the sample might become uniform too soon and not reach the global maximum. The most common case of converging to a wrong result is when the sample converges to a local maximum. In replay, due to the unique shape of the performance function with a very narrow global maximum, there is also the phenomenon of converging to some value which is neither a local nor a global maximum. In this work, we examine several ways to add random injection to the algorithm without compromising its convergence. We describe our experiments in more detail in Section 4 and compare the effect experimental of applying different methods on the convergence and the convergence rate of the program.

4 Implementation and Experimental Results

We use our cross-entropy-based testing tool ConCenter for checking our approach to replay of concurrent programs. We briefly describe ConCenter and the way replay is implemented in it in Section 4.1. We describe the experimental setting in Section 4.2 and present our experimental results in Section 4.3.

4.1 Implementation

The cross-entropy-based testing tool ConCenter is written in Java. Its structure is reflected in Figure 1, and we briefly describe each part of it below.

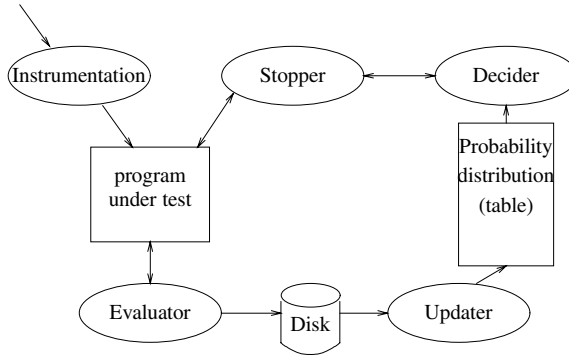


Fig. 1. Parts of ConCenter

- **Instrumenter** is an instrumentation tool that adds callbacks at control points.
- **Decider** receives a node v of the JCG of the program under test and chooses which thread is allowed to execute according to current relative probabilities $RP(v)$ on the control graph edges.
- **Stopper**: on callbacks from the instrumented code it stops the currently running thread using a mutex designated for this thread. Then, it calls notify() on the mutex of the thread that can execute next (based on Decider's decision).
- **Evaluator** collects the edges of the JCG traversed by the execution path. At the end of each execution it computes the S value of the execution.
- **Updater** updates the probability distribution table for the next iteration based on the computations of Evaluator.

During the execution, Decider and Evaluator collect big amounts of data. To minimize the sizes of the memory buffers for this data, it is periodically written to disk.

4.2 Description of the Experimental Setting

We performed several experiments on different metrics and different types of bugs. The tests were written in Java version 1.5.0 and executed on a 4 CPU machine 64bit “Dual Core AMD Opteron(tm) Processor 280” with clock rate of 2.4GHz and 1MB cache size each. The total memory of the machine is 8GB. The operation system it runs is GNU/Linux 2.6.9-42.0.3.ELsmp. In all examples, we executed the program once and recorded it using the instrumentation provided by ConTest. We then attempted to replay this execution with ConCenter. For each program, we repeated the experiment of recording and replay 10 times; the reported numbers are the average numbers calculated on these executions.

Tuning of parameters for ConCenter We checked the influence of different parameters on the success and the convergence rate of replay. These parameters control the execution of ConCenter as follows:

1. Target relative standard deviation is the stopping condition: when the relative standard deviation of the sample reaches the target, the execution terminates.

Table 1. Best Parameters for Replay (After Tuning)

target relative standard deviation	0.01
smoothing parameter	0.8
number of runs per series	200
quantile size q	0.2
target weight in initial distribution	0.9
injection threshold	0.05
injection factor	0.05
change of injection	1
injection frequency	3

2. Smoothing parameter is the weight of the new probability distribution in the weighted average with the previous probability distribution, as explained in Section 2.3.
3. Number of runs per series is the number of executions drawn from the space of all executions according to the current probability distribution on edges at each iteration.
4. Quantile size q is the best quantile of the sample according to the performance function (i.e., for example, the best 10% of the sample used to recompute the probability distribution for the next step).
5. Target weight in initial distribution is the initial bias of the distribution toward the recorded execution, as discussed in Section 3.1.
6. Injection threshold is the relative standard deviation of the best q -part of the sample that triggers random injection in the next iteration.
7. Injection factor is the weight of the random injection in the sample.
8. Change of injection is the multiplicative factor applied to the weight of the random injection at each iteration (if the change of injection is 1, then the weight of the injection is constant during the whole execution).
9. Injection frequency is x if a random injection is added to each x -th iteration, provided that it satisfies the criterion of the injection threshold.

In our setting, we performed the tuning of parameters manually, and the best values are presented in Table 1. Clearly, manual tuning of parameters is time-consuming. On the other hand, our experiments show that the best values of these parameters are approximately the same for all programs we applied our replay algorithm to, so we conjecture that in most cases tuning can be viewed as a one-time preprocessing task. It is also possible to let the algorithm to adjust these parameters automatically during the execution as described in Chapter 5 of [24].

4.3 Experimental Results

In this section we describe the concurrent programs on which we checked the convergence and performance of ConCenter and discuss the meaning of the experimental results.

Programs under test

Toy examples We start with two toy examples:

- A standard producer-consumer program, in which the producer threads fill the buffer, and the consumer threads empty the same buffer. The program launches two threads of each kind running concurrently, and the recorded execution contains a buffer overflow. Since placing a request in the buffer and removing it from the buffer require approximately the same time, a buffer overflow is reproduced only on executions on which only the producer threads run until the buffer is full. It is easy to see that if all threads are enabled all the time, the probability of reproducing a buffer overflow in a random execution is $O(1/2^n)$, where n is the size of the buffer.
- A “Push-pop” example, in which there are two types of threads, A and B , and each thread either pushes its name on top of the stack or pops the top element of the stack depending on the value of the current top element: if the value of the top element is equal to the name of the current thread, the thread pops the value, otherwise it pushes its name. The recorded execution contains stack overflow. It is easy to see that it can only be reproduced in executions in which threads constantly alternate, and thus, similarly to the previous example, the probability of reproducing it is $O(1/2^n)$, where n is the size of the stack.

ConCEnter consistently achieved the perfect replay of recorded executions with the values of parameters as specified in Table 1. We also checked the effect of biased initial distribution on the second example (see the results below).

Java 1.4 Collection Library Our real-life example is the Java 1.4 collection library that was used as a case study in [25]. This is a thread-safe collection framework implemented as a part of `java.util` package of the standard Java library provided by Sun Microsystems. We ran our experiments for all tests in this collection. The simplest tests *ITest* and *MTLinkedListInfiniteLoop* converged in less than 5 iterations to 100% replay. The other tests in the library, *MTListTest*, *MTSetTest*, and *MTVectorTest*, converged to 90% replay after less than 5 iterations, and to 95% replay after less than 10 iterations.

The effect of biased initial distribution. We experimented with the bias in the initial distribution ranging from 1 (probability 1 to choose the recorded execution), to 0.5 in steps of 0.2 and compared the results with the uniform initial distribution (that is, no bias toward the recorded distribution). The results presented in Table 2 are the average results over 10 executions, and it is easy to see that preprocessing of the probability table that assigns higher probabilities to edges that are present in the recorded execution decreases the number of iterations and significantly improves the probability of achieving a good replay. The experiments were done on the push-pop example.

We note that while the best results were achieved with the initial distribution that gives the recorded edges probability 1, in real-life examples and especially when the program was modified before it is replayed, we should not start with such a distribution. This is because giving the recorded edges the probability 1 effectively eliminates the

Table 2. Results for biased initial distribution, push-pop example

Bias weight	total replay success	good replay (> 95%) success	local maximum
1	100%	–	–
0.9	100%	–	–
0.7	90%	10%	–
0.5	100%	–	–
unbiased	30%	30%	40%

Table 3. Adding print statements before replay (Java 1.4 Collection Library)

Test name	number of iterations to 90% replay
MTListTest	2 – 3
MTSetTest	2 – 4
MTVectorTest	2

chance of choosing other edges. If the program is modified before it is replayed, the recorded execution might not be enabled at all, and thus the probability of choosing other edges should be greater than 0 to allow an approximate replay.

Changing the weight of injection. In our experiments, we checked the influence of injection on convergence of the cross-entropy based replay by varying the weight of injection from 0.01 to 0.1. We also checked the effect of different injection frequency, from 1 (each iteration) to 10. The best results, on all examples, were obtained with the weight of injection between 0.02 and 0.03, and injection frequency either 2 or 3.

Replay after program modification. In order to simulate a common debugging scenario, we introduced print statements after each action on the shared variables in the examples from Java 1.4 Collection Library. The print statements are conditional on a random bit, i.e., executed with the probability 50% at each execution, and they were introduced after the execution was recorded, which corresponds to the scenario where an engineer attempts to debug a program by introducing print statements and then reproducing the bug. Table 3 shows that a replay that is 90% close to the recorded execution was achieved after less than 5 iterations.

Comparison with ConTest. We compared our results to ConTest, where replay can be attempted by using the same random seed as in the recorded execution. We executed ConTest on the examples above, saved the seed of the random noise generator's decisions and attempted to replay the execution 100 times for each example. To compare the executions in ConTest, we checked the number of elements in the buffer after the execution terminates (that is, a looser criterion than an identity between executions). In buffer overflow, ConTest succeeded in 4 attempts, and in push-pop in 16 attempts (out of 100), even with such a permissive criterion of equivalence between executions. On the collection of program from the Java 1.4 Collection Library, ConTest did not achieve

even an approximate replay. This is hardly surprising, since the same random seed does not guarantee the same scheduling of concurrent events.

5 Conclusions and Future Work

We presented an application of cross-entropy method to replay of concurrent programs. To the best of our knowledge, this is the first approach that does not require a full recording of the concurrent events in the execution and also allows only partial control over the repeated executions. Our approach also accommodates program changes between the recording and the replay. For example, adding print statements to the program before replaying it does not interfere with the replay. In future work, we will integrate other techniques into replay, such as clustering algorithms, simulated annealing, and other methods for updating the probability distribution. We also plan to integrate techniques from genetic algorithms, as there seems to be quite a significant common ground between them and cross-entropy, especially when cross-entropy is applied to reproducing a single event, such as in replay.

References

1. Agarwal, A., Garg, V.K.: Efficient dependency tracking for relevant events in concurrent systems. *Distributed Computing* 19(3), 163–182 (2006)
2. Alon, G., Kroese, D.P., Raviv, T., Rubinstein, R.Y.: Application of the cross-entropy method to buffer allocation problem in simulation-based environment. *Annals of Operations Research* (2004)
3. Carver, R.H., Tai, K.C.: Replay and testing for concurrent programs. *IEEE Software* 8(2), 66–74 (1991)
4. Chockler, H., Farchi, E., Godlin, B., Novikov, S.: Cross-entropy based testing. In: *Proceedings of Formal Methods in Computer Aided Design (FMCAD)*, pp. 101–108. IEEE Computer Society, Los Alamitos (2007)
5. Choi, J.-D., Srinivasan, H.: Deterministic replay of multithreaded java applications. In: *ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pp. 48–59 (1998)
6. Costa, A., Jones, O.D., Kroese, D.: Convergence properties of the cross-entropy method for discrete optimization. *Operations Research Letters* 35(5), 573–580 (2007)
7. Dubin, U.: The cross-entropy method for combinatorial optimization with applications. Master Thesis, The Technion (2002)
8. Edelstein, O., Farchi, E., Nir, Y., Ratzaby, G., Ur, S.: Multithreaded java program test generation. *IBM Systems Journal* 41(3), 111–125 (2002)
9. Hamming, R.W.: Error detecting and error correcting codes. *Bell System Technical Journal* 26(2), 147–160 (1950)
10. Keith, J.M., Kroese, D.P.: Rare event simulation and combinatorial optimization using cross entropy: sequence alignment by rare event simulation. In: *Proceedings of the 34th Winter Simulation Conference: Exploring New Frontiers*, pp. 320–327. ACM, New York (2002)
11. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. *Science* 220(4598), 671–680 (1983)
12. Krause, E.F.: *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*, Dover (1987)
13. Kullback, S., Leibler, R.A.: On information and sufficiency. *Annals of Mathematical Statistics* 22, 79–86 (1951)

14. Kumar, R., Garg, V.K.: Modeling and control of logical discrete event systems. Kluwer Academic Publishers, Dordrecht (1995)
15. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7), 558–565 (1978)
16. Leblanc, T.J., Mellor-Grummy, J.M.: Debugging parallel programs with instant replay. *IEEE Transactions on Computers* 36(4), 471–481 (1987)
17. Margolin, L.: Cross-entropy method for combinatorial optimization. Master Thesis, The Technion (2002)
18. Mittal, N., Garg, V.K.: Debugging distributed programs using controlled re-execution. In: *ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 239–248 (2000)
19. Mittal, N., Garg, V.K.: Finding missing synchronization in a distributed computation using controlled re-execution. *Distributed Computing* 17(2), 107–130 (2004)
20. Netzer, R.H.B.: Optimal tracing and replay for debugging shared-memory parallel programs. In: *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*; also available as *ACM SIGPLAN Notices* 28(12), 1–11 (1993)
21. Paik, E.H., Chung, Y.S., Lee, B.S.: Chae-Woo Yoo. A concurrent program debugging environment using real-time replay. In: *Proc. of ICPADS*, pp. 460–465 (1997)
22. Rubinstein, R.Y.: Optimization of computer simulation models with rare events. *European Journal on Operations Research* 99, 89–112 (1997)
23. Rubinstein, R.Y.: The cross-entropy method and rare-events for maximal cut and bipartition problems. *ACM Transactions on Modelling and Computer Simulation* 12(1), 27–53 (2002)
24. Rubinstein, R.Y., Kroese, D.P.: The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning. In: *Information Science and Statistics*. Springer, Heidelberg (2004)
25. Sen, K., Agha, G.A.: Cute and jcute: Concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) *CAV 2006*. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)

Control Dependence for Extended Finite State Machines

Kelly Androutsopoulos¹, David Clark¹, Mark Harman¹, Zheng Li¹,
and Laurence Tratt²

¹Department of Computer Science, King's College London, Strand, London,
WC2R 2LS, United Kingdom

{kalliopi.androutsopoulos,david.j.clark,mark.harman,zheng.li}@kcl.ac.uk

²Bournemouth University, Poole, Dorset, BH12 5BB, United Kingdom
laurie@tratt.net

Abstract. Though there has been nearly three decades of work on program slicing, there has been comparatively little work on slicing for state machines. One of the primary challenges that currently presents a barrier to wider application of state machine slicing is the problem of determining control dependence. We survey existing related definitions, introducing a new definition that subsumes one and extends another. We illustrate that by using this new definition our slices respect Weiser slicing's termination behaviour. We prove results that clarify the relationships between our definition and older ones, following this up with examples to motivate the need for these differences.

Keywords: extended finite state machines, reactive systems, control dependence, slicing.

1 Introduction

Program slicing is a source code analysis technique that identifies the parts of a program's source code which can affect the computation of a chosen variable at a chosen point in it. The variable and point of interest constitute the slicing criterion. There are many variations on the slicing theme. For instance, slices can be constructed statically (with respect to all possible inputs), dynamically (with respect to a single input) or within some spectrum in-between. Program slicing has proved to be widely applicable, with application areas ranging from program comprehension [HBD03] to reverse engineering and reuse [CCD98].

However, despite thirty years of research, several hundred papers and many surveys on *program* slicing [BH04, De 01, Tip95], there has been comparatively little work on slicing at the *model* level. This paper tackles slicing at the model level, particularly static slicing of Finite State Machines (FSMs).

FSMs are a graphical formalism that have become widely used in specifications of embedded and reactive systems. Their main drawback is that even moderately complicated systems result in large and unwieldy diagrams. Harel's Statecharts [Har87] and Extended Finite State Machines (EFSMs) are two of the many attempts over past decades to address FSM's disadvantages.

Work on slicing FSM models began with the work of Heimdahl et al. in the late 1990s [HW97, HTW98], followed by Wang et al. [WDQ02] and then in 2003 by the work of Korel et al. [KSTV03] and more recently by Langenhove and Hoogewijs [LH07] and by Labbé et al. [LGP07, LG08].

One of the challenges facing any attempt to slice an EFSM is the problem of how to correctly account for control dependence. It is common for state machines modelling such things as reactive systems not to have a final computation point or ‘exit node’. To overcome this problem, Ranganath et al. [RAB⁺05] recently introduced the concept of a control sink and associated control dependence definitions for reactive programs. A control sink is a strongly connected component from which control flow cannot escape once it enters. Building on this, Labbé et al. [LGP07, LG08] introduced a notion of control dependence and an associated slicing algorithm for EFSMs that was non-termination sensitive. However, they introduce a syntax dependent condition and thus cannot be applied to any FSM.

However, traditional control dependence, as used in program slicing [HRB90] is non-termination *insensitive*, with the consequence that the semantics of a program slice dominates the semantics of the program from which it is; slicing may remove non-termination, but it will never introduce it. In moving slicing from the program level to the state based model level, an important choice needs to be made

“should EFSM slicing be non-termination sensitive or insensitive?”

Recent work on control dependence has only considered the non-termination sensitive option [LGP07, LG08]. The non-termination insensitive option was explored by Korel et al. [KSTV03], but only for the restricted class of state machines that guarantee to have an exit state. Heimdahl et al. [HW97, HTW98] have a different notion of control dependence which is not a structural property of the graph of FSMs but is based on the dependency relation between events and generated events. This could lead to slices being either non-termination sensitive or insensitive depending on the specification. The definition of control dependence given in [WDQ02, LH07] is for UML statecharts with nested and concurrent states and is the same as that of data dependence when applied to EFSMs that do not have concurrent and/or nested states. This leaves open the question of how to extend control dependence to create non-termination insensitive slicing for general EFSMs in which there may be no exit node.

This problem is not merely of intellectual curiosity as it also has implications for the applications of slicing. In the literature on traditional program slicing, a non-termination sensitive formulation was proposed as early as 1993 by Kamkar [Kam93], but has not been taken up in subsequent slicing research. Non-termination sensitive slicing tends to produce very large slices, because all iterative constructs that cannot be statically determined to terminate must be retained in the slice, no matter whether they have any effect other than termination on the values computed at the slicing criterion. These ‘loop shells’ must be retained in order to respect the definition of non-termination sensitivity. Furthermore, for most of the applications of slicing listed above, it turns out that it is perfectly acceptable for slicing to be non-termination insensitive.

In this paper, we introduce a non-termination insensitive form of control dependence for EFSM dependence analysis, that can be applied to any FSM, and a slicing algorithm based upon it. Like Labbé et al., we build on the recent work of Ranganath et al. [RAB⁺05], but our definition is non-termination insensitive. Also, unlike Korel's definition, our development of the recent work of Ranganath et al. allows us to handle arbitrary EFSMs. We prove that our definition of control dependence is backward compatible with traditional non-termination insensitive control dependence outside of control sinks. Furthermore, we prove that our definition agrees with the non-termination sensitive control dependence of Labbé et al. inside control sinks. Finally we demonstrate the type of slices produced with our definition.

2 Extended Finite State Machines

We formally define an EFSM as follows.

Definition 1 (Extended Finite State Machine). *An Extended Finite State Machine (EFSM) $E=(S, T, Ev, V)$ where S is a set of states, T is a set of transitions, Ev is a set of events, and V is a store represented by a set of variables. Transitions have a source state $source(t) \in S$, a target state $target(t) \in S$ and a label $lbl(t)$. Transition labels are of the form $e_1[g]/a$ where $e_1 \in Ev$, g is a guard, i.e. a condition (we assume a standard conditional language) that guards the transition from being taken when an e_1 is true, and a is a sequence of actions (we assume a standard expression language including assignments). All parts of a label are optional.*

EFSMs are possibly non-deterministic. States of S are atomic. Actions can involve store updates or generation of events or both. A transition t may have a successor t' whose source is the same as the target of t . Two or more distinct transitions which share the same source node are said to be *siblings*. A final transition is a transition whose target is an exit state and an exit state is a state which has no outgoing transitions. An ε -transition is one with no event or guard.

3 Survey

In this section we survey several existing definitions of control dependence and discuss their strengths and weaknesses.

Ranganath et al.'s control dependence definitions [RAB⁺05, RAB⁺07] are defined for programs of systems with multiple exit points and / or which execute indefinitely, and therefore form the basis for subsequent state machine control dependence definitions. We exclude from this discussion the control dependence definition as given in [WDQ02, LH07] because it is defined in terms of concurrent states and transitions and EFSMs do not have concurrent states and transitions. Moreover, when applied to states and transitions that are not concurrent, it is the same as data dependence as in Definition 13.

Korel et al [KSTV03], Ranganath et al. and Labbé et al. [LGP07, LG08] definitions of control dependence are given in terms of execution paths. Since a path is commonly presented as a (possibly infinite) sequence of nodes, a node is in a path if it is in the sequence. A transition is in a path if its source state is in the path and its target state is both in the path and immediately follows its source state. A *maximal path* is any path that terminates in an end node or final transition, or is infinite.

3.1 Control Flow for RSML

Heimdahl et al. [HW97, HTW98] present an approach for slicing specifications modelled in the Requirements State Machine Language (RSML) [LHHR94], a tabular notation that is based on hierarchical finite state machines. Transitions have events, guards and actions; events can generate events as actions, which are broadcast in the next step of execution. Heimdahl et al. were the first to present a control dependence-like definition for FSMs; it differs from the traditional notion as it defines control flow in terms of events rather than transitions.

Definition 2 (Control flow for RSML (CF) [HTW98]). *Let E be the set of all events and T the set of all transitions. The relation $\text{trigger}(T \rightarrow E)$ represents the trigger event of a transition. The relation $\text{action}(T \rightarrow E^2)$ represents the set of events that make up the action caused by executing a transition. $\text{follows}(T \rightarrow T)$ is defined as: $(t_1, t_2) \in \text{follows}$ iff $\text{trigger}(t_1) \in \text{action}(t_2)$.*

CF can be applied to non-terminating systems that have multiple exit nodes. However, it depends on transitions being triggered by events and being able to generate events as actions and therefore cannot be applied to any finite state machine, such as EFSMs that do not generate events.

3.2 Control Dependence for EFSMs

Korel et al. [KSTV03] present a definition of control dependence for EFSMs in terms of post dominance that requires execution paths to lead to an exit state.

Definition 3 (Post Dominance [KSTV03]). *Let Y and Z be two states and T be an outgoing transition from Y .*

- *State Z post-dominates state Y iff Z is in every path from Y to an exit state.*
- *State Z post-dominates transition T iff Z is on every path from Y to the exit state though T . This can be rephrased as Z post-dominates $\text{target}(T)$.*

Definition 4 (Insensitive Control Dependence (ICD) [KSTV03]). *Transition T_k is control dependent on transition T_i if:*

1. *$\text{source}(T_k)$ post-dominates transition T_i (or $\text{target}(T_i)$), and*
2. *$\text{source}(T_k)$ does not post-dominate $\text{source}(T_i)$.*

This definition is successful in capturing the traditional notion of control dependence for static backward slicing. However it can only determine control dependence for state machines with exactly one end state, failing if there are multiple exit states or if the state machine is possibly non-terminating.

3.3 Control Dependence for Non-terminating Programs

Ranganath et al. [RAB⁺05, RAB⁺07] address the issue of determining control dependence for programs utilising Control Flow Graphs (CFGs). A CFG is a labelled, directed graph with a set of nodes that represent statements in a program and edges that represent the control flow. A node is either a *statement node* (which has a single successor) or a *predicate node* (which has two successors, labelled with *T* or *F* for the true and false cases respectively). A CFG has a start node n_s (which must have no incoming edges) such that all nodes are reachable from n_s ; it may have a set of end nodes that have no successors.

Two versions of control dependence definitions are described: *non-termination sensitive* and *non-termination insensitive* control dependence. The difference between these definitions lies in the choice of paths. Non-termination sensitive control dependence is given in terms of maximal paths.

Definition 5 (Non-termination Sensitive Control Dependence

(NTSCD)). In a CFG, $N_i \xrightarrow{NTSCD} N_j$ means that a node N_j is non-termination sensitive control dependent on a node N_i iff N_i has at least two successors N_k and N_l such that: for all maximal paths π from N_k , where $N_j \in \pi$; and there exists a maximal path π_0 from N_l where $N_j \notin \pi_0$.

Non-termination insensitive control dependence is given in terms of *sink-bounded paths* that end in *control sinks*. A control sink is a region of the graph which, once entered, is never left. These regions are always SCCs, even if only the trivial SCC, i.e. a single node with no successors.

Definition 6 (Control Sink). A control sink, $\mathcal{K}$, is a set of nodes that form a strongly connected component such that, for each node n in $\mathcal{K}$ each successor of n is in $\mathcal{K}$.

Definition 7 (Sink-bounded Paths). A maximal path π is sink-bounded iff there exists a control sink $\mathcal{K}$ such that:

1. π contains a node from $\mathcal{K}$;
2. if π is infinite then all nodes in $\mathcal{K}$ occur infinitely often.

The second clause of Definition 7 defines a form of fairness and hence we refer to it as the *fairness condition*. $\text{SinkPaths}(N)$ denotes a set of sink-bounded paths from a node N . We now define Ranganath et al. [RAB⁺05] non-termination *insensitive* version of control dependence.

Definition 8 (Non-termination Insensitive Control Dependence

(NTICD)). In a CFG, $T_i \xrightarrow{NTICD} N_j$ means that a node N_j is non-termination insensitive control dependent on a node N_i iff N_i has at least two successors N_k and N_l such that:

1. for all paths $\pi \in \text{SinkPaths}(N_k)$ where $N_j \in \pi$;
2. there exists a path $\pi_0 \in \text{SinkPaths}(N_l)$ where $N_j \notin \pi_0$.

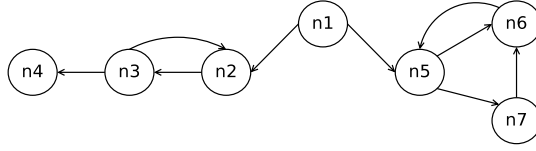


Fig. 1. A CFG with multiple exit points and which is potentially non-terminating

The difference between paths in NTSCD and NTICD is shown in Figure 1. According to Definition 5, $n1 \xrightarrow{\text{NTSCD}} n2$ and $n1 \xrightarrow{\text{NTSCD}} n3$ but not $n1 \xrightarrow{\text{NTSCD}} n4$ because $n4$ is not in all maximal paths as there is a maximal path with an infinite loop, i.e. $\{n2 \rightarrow n3 \rightarrow n2 \dots\}$. However, $n1 \xrightarrow{\text{NTICD}} n2, n3, n4$ since $n2, n3$ and $n4$ occur on all sink-bounded paths from $n2$ (the control sink for these paths is $n4$) and there exists a sink bounded path from $n5$ (the control sink consists of $n5, n6, n7$) which does not include $n2, n3$ and $n4$. Compared to NTSCD, NTICD cannot calculate any control dependencies within control sinks. For example, in Figure 1, $n5 \xrightarrow{\text{NTSCD}} n7$ but no such dependency exists for NTICD. Some programs (e.g. servers) are a global control sink and as such there would be NTSCD, but no NTICD, dependencies.

3.4 Control Dependence for Communicating Automata

Labbé et al. [LG08]¹ adapt Ranganath et al.’s NTSCD definition for communicating automata, in particular focusing on Input/Output Symbolic Transition Systems (IOSTS) [GGRT06].

Definition 9 (Labbé et al.- Non-Termination Sensitive Control Dependence (LG-NTSCD) [LG08]). *For an IOSTS S , a transition T_j is control dependent on a transition T_i if T_i has a sibling transition T_k such that:*

1. T_i has a non-trivial guard, i.e. a guard whose value is not constant under all variable valuations;
2. for all maximal paths π from T_i , the source of T_j belongs to π ;
3. there exists a maximal path π_0 from T_k such that the source of T_j does not belong to π_0 .

FSM models differ from CFGs in several ways. For example, FSMs can have multiple start and exit nodes, more than two edges between two states and more than two successors from a state. Moreover, in CFGs, decisions (Boolean conditions) are made at the predicate nodes while in state machines they are made on transitions. Labbé et al. take such differences into account when adapting NTSCD. For example in Figure 2 $T2 \xrightarrow{\text{NTSCD}} T3$ and $T3 \xrightarrow{\text{NTSCD}} T2$ because according to the second clause in Definition 5 the maximal paths start from *start*. However these control dependencies are non-sensical because $T2$ and $T3$ are

¹ Labbé et al.’s definition of control dependence in [LGP07] differs slightly from Labbé et al. [LG08], so we evaluate the most recent.

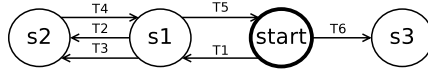


Fig. 2. If NTSCD or NTICD is applied, undesired dependences are produced

sibling transitions. Using LG-NTSCD these control dependencies do not exist because in the third clause of Definition 9 the maximal paths start from $s1$.

The first clause of LG-NTSCD concerning the non-triviality of guards is introduced in order to avoid a transition being control dependent on transitions that are executed non-deterministically even though they are NTSCD control dependent. Furthermore, because this is a syntax dependent clause, the definition cannot be applied to many FSMs, such as the FSM for the elevator system in Figure 3 that contains transitions with trivial guards.

4 New Control Dependence Definition: UNTICD

We define a new control dependence definition by extending Ranganath et al.'s NTICD definition and subsuming Korel et al.'s definition in order to capture a notion of control dependence for EFSMs that has the following properties. First, the definition is general in that it should be applicable to any reasonable FSM language variant. Second, it is applicable to non-terminating FSMs and / or those that have multiple exit states. Third, by choosing FSM slicing to be non-termination insensitive (in order to coincide with traditional program slicing) it produces smaller slices than traditional non-termination sensitive slicing.

Following [RAB⁺05], the paths that we consider are sink-bounded paths, i.e. those that terminate in a control sink as in Definition 6. Unlike NTICD, the sink-bounded paths are unfair, i.e. we drop the fairness condition in Definition 7. For non-terminating systems this means that control dependence can be calculated within control sinks.

Definition 10 (Unfair Sink-bounded Paths). *A maximal path π is sink-bounded iff there exists a control sink $\mathcal{K}$ such that π contains a transition from $\mathcal{K}$.*

Note that a transition is in a path if its source state is in the path and its target state is both in the path and immediately follows its source state.

Definition 11 (Unfair Non-termination Insensitive Control Dependence (UNTICD)). $T_i \xrightarrow{\text{UNTICD}} T_j$ means that a transition T_j is control dependent on a transition T_i iff T_i has at least one sibling T_k such that:

1. for all paths $\pi \in \text{UnfairSinkPaths}(\text{target}(T_i))$, the source(T_j) belongs to π ;
2. there exists a path $\pi \in \text{UnfairSinkPaths}(\text{source}(T_k))$ such that the source(T_j) does not belong to π .

UNTICD is in essence a version of NTICD modified to EFSMs (rather than CFGs) and given in terms of unfair sink-bounded paths. This means that, unlike in the second clause of Definition 8, sink-bounded paths start from the source of T_k rather than from the target of T_k because EFSMs can have many transitions between states and Definition 8 would lead to non-sensical dependences, e.g. in Figure 2 $T2 \xrightarrow{\text{NTICD}} T3$ while according to our definition $T2$ does not control $T3$.

5 Properties of the Control Dependence Relation

We prove the following properties for UNTICD: UNTICD subsumes ICD; the transitive closure for the NTICD relation is contained in the transitive closure for the UNTICD relation; and for an EFSM M , UNTICD and NTSCD dependences for all transitions within control sinks are identical.

5.1 UNTICD Subsumes ICD

Proposition 1 *Definition 4 (ICD) is a special case of Definition 11 (UNTICD).*

Proof. Definition 4 is given in terms of post dominance which considers every path to a unique exit state. Definition 11 is given in terms of sink-bounded paths that terminate in control sinks. The final transition that leads to the exit state is a trivial strongly-connected component that has no successors, and hence is a control sink. Therefore, the paths in ICD are contained in the paths of NTICD, but NTICD is not restricted to these. Moreover, the clauses of definition 4 are the same as the clauses of definition 11. $\square$

5.2 Relation between NTICD and UNTICD's Transitive Closures

In Theorem 2 we show that the transitive closure of $\xrightarrow{\text{NTICD}}$ is contained in the transitive closure of $\xrightarrow{\text{UNTICD}}$. This shows that UNTICD does not introduce any additional dependences other than NTICD outside of the control sinks (see Lemma 2) but introduces dependences within control sinks. In order to prove this theorem, we first need to identify the regions in the state machine where dependencies can occur and we do that by considering all the cases in which a transition $t1$ controls another transition $t2$, where $K, K1, K2$ are control sinks:

$$\forall K. t1 \notin K \wedge t2 \notin K \quad (1)$$

$$\exists K. t1 \in K \wedge t2 \in K \quad (2)$$

$$\exists K1, K2. t1 \in K1 \wedge t2 \in K2 \quad (3)$$

$$\forall K. t1 \notin K \wedge \exists K. t2 \in K \quad (4)$$

$$\forall K. t2 \notin K \wedge \exists K. t1 \in K \quad (5)$$

In case (1) both $t1$ and $t2$ are not in any control sink K . In case (2) both $t1$ and $t2$ are in the same control sink K . In case (3) $t1$ is in a control sink $K1$ and $t2$ is in another control sink $K2$. In case (4) $t1$ is not in any control sink and $t2$

belongs to a control sink. In case (5) t_2 does not belong to any control sink and t_1 belongs to a control sink. We introduce Definition 12 that defines a descendant of a transition and the Lemma 1 so that we can discard any impossible cases.

Definition 12 (Descendent). *A descendant of t is a transition related to t by the closure of the successor relation.*

Lemma 1. *For all transitions t in a control sink K , all descendants of t belong to K .*

Proof. By Definition 6 of the control sink and Definition 12 of the descendant relation. $\square$

By Lemma 1, cases (3) and (5) are not possible since t_1 can only control t_2 if t_2 is a descendant of t_1 . Therefore, we only consider cases (1), (2), and (4). When t_1 NTICD controls t_2 , then for each case we write case1^F , case2^F , and case4^F . Similarly when t_1 UNTICD t_2 , then we write case1^U , case2^U , and case4^U .

Lemma 2 shows that the control dependences produced by applying UNTICD to transitions outside of the control sink are the same as those produced when applying NTICD, i.e. $\text{case1}^U = \text{case1}^F$.

Lemma 2. *For an EFSM M , NTICD and UNTICD dependences for transitions T outside of the control sink K (where $t \in T$ and $t \notin K$), are the same.*

Proof. Let us assume that in an EFSM M , T_j is NTICD control dependent on T_i ($T_i \xrightarrow{\text{NTICD}} T_j$) and that T_i and T_j are outside of the control sink. From Definition 8, T_i has a sibling transition T_k such that there exists a path $\pi_k \in \text{SinkPaths}(T_k)$ where the $\text{source}(T_j)$ does not belong to π_k .

Now suppose that the fairness condition in the definition of sink bounded paths is removed, i.e. Definition 11 holds, then this affects the transitions within the control sink only in that they do not occur infinitely often. The source of T_j still remains on all paths from T_i as these are outside of the control sink and π_k still exists. Therefore, NTICD and UNTICD dependences of transitions outside of the control sink are the same. $\square$

The pairwise intersection of case (1), (2), (4) are empty. Therefore the relations can be partitioned as follows:

$$\begin{aligned} \text{case1}^F \cup \text{case2}^F \cup \text{case4}^F &= \xrightarrow{\text{NTICD}} \\ \text{case1}^U \cup \text{case2}^U \cup \text{case4}^U &= \xrightarrow{\text{UNTICD}} \end{aligned}$$

In Theorem 2 we show that the transitive closure of NTICD dependences between transitions within a control sink and between a transition outside of the control sink and a transition within a control sink is a subset of the transitive closure of UNTICD dependences between transitions within a control sink and between a transition outside of the control sink and a transition within a control sink, i.e. $(\text{case2}^F \cup \text{case4}^F)^* \subseteq (\text{case2}^U \cup \text{case4}^U)^*$. First we prove the following lemma.

Lemma 3. *Let $A \cap B = C \cap D = \emptyset$ and $X = A \cup B$ while $Y = C \cup D$. If $A = C$ then $X^* \subseteq Y^*$ if $B^* \subseteq D^*$. All relations are over the same base set.*

Proof. $(x_1, x_2) \in X^*$ iff there exists a path $\pi \in (x_1, x_2), (x_2, x_3), \dots, (x_{n-1}, x_n)$ so that for two successive members (x_i, x_j) and $(x_k, x_l) \in \pi$, $x_j = x_k$, and for all $(x_i, x_j) \in X$. This constructs the smallest transitive closure of X .

We show $X^* \subseteq Y^*$ by induction on the length of the path π in X^* .

Base Case: $\text{length}(\pi) = 1$ then either $(x_0, x_1) \in A = C \subseteq (C \cup D)^* = Y^*$ or $(x_0, x_1) \in X^*$ because $(x_0, x_1) \in B \subseteq B^* \subseteq D^* \subseteq (C \cup D)^* = Y^*$

Induction Case: (Inductive Hypothesis (IH)) Let xX^*y because there exists a path $\pi \in xX^*x_1X^*x_2\dots X^*y$ of length N in X^* . Then there exists a path π_1 in Y such that xY^*y .

Let xX^*z because there exists a path π of length $N + 1$ in X . Then $\exists y, z$. xX^*yXz and by IH xY^*y by the same arguments for the base case of yXz then yY^*z hence xY^*z . $\square$

Theorem 2. *The transitive closure of NTICD, is contained in the transitive closure of UNTICD. $\xrightarrow{\text{NTICD}^*} \subseteq \xrightarrow{\text{UNTICD}^*}$*

Proof. $\xrightarrow{\text{NTICD}^*} \subseteq \xrightarrow{\text{UNTICD}^*}$ can also be expressed as the transitive closure for all of the cases: $(\text{case1}^F \cup \text{case2}^F \cup \text{case4}^F)^* \subseteq (\text{case1}^U \cup \text{case2}^U \cup \text{case4}^U)^*$ which is true if:

- $\text{case1}^F = \text{case1}^U$, i.e. that NTICD and UNTICD dependences between transitions that are not in a control sink are the same, by Lemma 2, and
- $(\text{case2}^F \cup \text{case4}^F)^* \subseteq (\text{case2}^U \cup \text{case4}^U)^*$, i.e. that the transitive closure of NTICD dependences between transitions within a control sink and a transition outside of the control sink, and between transitions within a control sink is a subset of the transitive closure of UNTICD dependences between transitions within a control sink and a transition outside of the control sink, and between transitions within a control sink. This is true because of Lemma 3. $\square$

5.3 NTSCD and UNTICD Dependencies within Control Sinks

Finally, we show that UNTICD and NTSCD are compatible in control sinks.

Theorem 3. *For every $T_i \in K$ and $T_j \in K$ where K is a control sink in EFSM M , $T_i \xrightarrow{\text{UNTICD}} T_j$ iff $T_i \xrightarrow{\text{NTSCD}} T_j$.*

Proof. In a control sink K , if $T_i \in K$ and $T_j \in K$, then according to Definition 11 sink-bounded paths are reduced to maximal paths, since transitions in K do not occur infinitely often (fairly). This coincides with Definition 5. Therefore, the control dependences produced by UNTICD and NTSCD for transitions within control sinks are equivalent. $\square$

6 Comparison of UNTICD with Existing Definitions

Figure 3 illustrates an EFSM of the door control component, a subcomponent of the elevator control system [SW99]. The door component controls the elevator

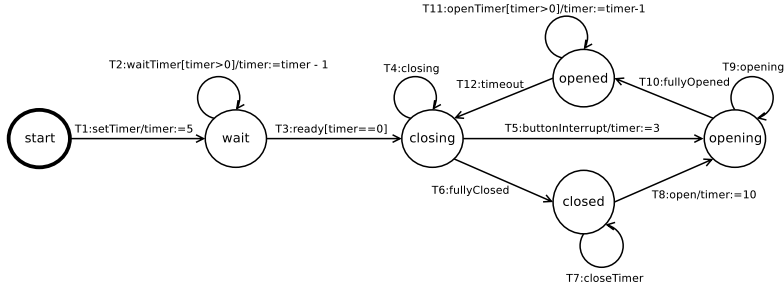


Fig. 3. An EFSM specification for the door control of the elevator system

CF	No dependences as the EFSM does not have generated events	
ICD	Not applicable as the EFSM does not have a unique exit state	
NTSCD	$wait \rightarrow closing$	$closing \rightarrow closed$
	$closed \rightarrow opening$	$opening \rightarrow opened$
	$opened \rightarrow closing$	$closing \rightarrow opening$
NTICD	No dependences	
LG-NTSCD	$T3 \rightarrow T4, T5, T6$	
UNTICD	$T5 \rightarrow T9, T10$	$T6 \rightarrow T7, T8$
	$T8 \rightarrow T9, T10$	$T10 \rightarrow T11, T12$
	$T12 \rightarrow T4, T5, T6$	

Fig. 4. Control dependences computed by new and existing definitions for Figure 3

door, i.e. it opens the door, waits for the passengers to enter or leave the elevator and finally shuts the door. In this section we compute all the control dependencies for this EFSM using the existing and new definitions for the purpose of comparison, as given in Figure 4.

CF cannot be applied to the EFSM in Figure 3 because it is given in terms of the relationship between events and generated events and according to the syntax of EFSMs, events cannot be generated.

ICD cannot be applied to the the EFSM in Figure 3 because it does not have a unique exit state. For EFSMs that lead to a unique exit state the control dependences computed for both ICD and UNTICD are the same. For example, in Figure 2, ICD and UNTICD compute the same dependences, i.e. $T1 \rightarrow T2, T3, T4, T5$.

In Figure 4, NTSCD and NTICD are given in terms of nodes but can easily be represented in terms of transitions. Compared to UNTICD, NTSCD considers maximal paths rather than sink-bounded paths and consequently introduces more dependences when there are loops on paths that lead to a control sink. For example, in Figure 3, $wait \xrightarrow{\text{NTSCD}} closing$ because of the loop introduced by the self-transition $T2$. Note that NTSCD and UNTICD have the same dependences inside control sinks—we have formally shown this to be true in Theorem 3.

In Figure 3 there are no NTICD dependences because any control dependency caused by loops on paths to a control sink are ignored and there are

no control dependencies within control sinks because of the fairness condition of sink-bounded paths. Unlike NTICD, UNTICD calculates dependences with control sinks. Also, as formally shown by Theorem 2, the transitive closure of NTICD is contained within the transitive closure of UNTICD, although trivially true in this case.

LG-NTSCD is NTSCD adapted for transitions and with a syntax dependent clause, i.e. that the controlling transition's guard must be non-trivial. This additional clause reduces the number of dependences compared to those of NTSCD. For example, in Figure 3, $T5, T6, T8, T10$ and $T12$ do not control any other transition because they have trivial guards. The transitive closure of LG-NTSCD as for slicing, could produce too few results to be useful.

7 EFSM Slicing with UNTICD

Backward static program slicing was first introduced by Weiser [Wei81] and describes a source code analysis technique that, through dependence relations, identifies all the statements in the program that influence the computation of a chosen variable and point in the program, i.e. the slicing criterion. It is non-termination insensitive. Similarly, EFSM slicing identifies those transitions which affect the slicing criterion, by computing control dependence and data dependence. Data dependence is a definition-clear path between a variable's definition and use. We adopt the data dependence definition of [KSTV03] for an EFSM.

Definition 13 (Data Dependence (DD)). $T_i \xrightarrow{DD}_v T_k$ means that transitions T_i and T_k are data dependent with respect to variable v if:

1. $v \in D(T_i)$, where $D(T_i)$ is a set of variables defined by transition T_i , i.e. variables defined by actions and variables defined by the event of T_i that are not redefined in any action of T_i ;
2. $v \in U(T_k)$, where $U(T_k)$ is a set of variables used in a condition and actions of transition T_k ;
3. there exists a path in an EFSM from the source(T_i) to the target(T_k) whereby v is not modified.

The data dependences for the door controller EFSM in Figure 3 are: $\{T1 \rightarrow T2, T3\}$, $\{T2 \rightarrow T2, T3\}$, $\{T5 \rightarrow T11\}$, $\{T8 \rightarrow T11\}$, and $\{T11 \rightarrow T11\}$.

Definition 14 (Slicing Criterion). A slicing criterion for an EFSM is a pair (t, V) where transition $t \in T$ and variable set $V \subseteq Var$. It designates the point in the evaluation immediately **after** the execution of the action contained in transition t .

Definition 15 (Slice). A slice of an EFSM M , is an EFSM, M' , that contains ε -transitions. The transitions that are not ε -transitions are in the set of transitions that are directly or indirectly (transitive closure) DD and UNTICD on the slicing criterion c .

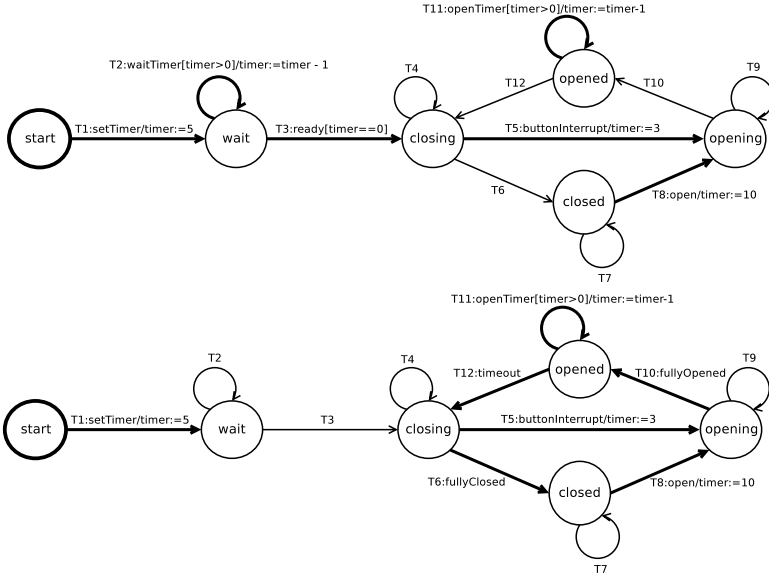


Fig. 5. Static slices computed with LG-NTSCD (top) and UNTICD (bottom). Marked transitions are in bold. LG-NTSCD has less marked transitions than UNTICD because dependences in the control sink are not valid as transitions have trivial guards.

7.1 Computing EFSM Slices

The objective of the slicing algorithm is to automatically compute the slice of an EFSM model M with respect to the given slicing criterion c . First, the algorithm computes the data dependences, using Definition 13, and the control dependences, using Definition 11, for all transitions in M . These are then represented in a dependence graph, which is a directed graph where nodes represent transitions and edges represent data and control dependences between transitions. Then, given the slicing criterion c , the algorithm marks all backwardly reachable transitions from c , i.e. the transitive closure of DD and UNTICD with respect to c . All unmarked transitions are anonymised i.e. become ε -transitions. Note that we can replace UNTICD, with NTICD, LG-NTSCD and NTSCD in order to compare the different slices produced.

If the slicing criterion for the EFSM in Figure 3 is T_{11} , then Figure 5(a) illustrates the slice produced when using UNTICD, and Figure 5(b) illustrates the slice produced when using LG-NTSCD. Unlike LG-NTSCD and NTSCD, UNTICD slicing slices away transitions which are affected by loops (before control sinks) that do not data dependent on T_{11} , i.e. T_3 . Moreover, there are no LG-NTSCD dependences within the control sink because the transitions have trivial guards. Trivial guards in Figure 3 do not affect whether T_{10} and T_9 will be taken non-deterministically, so in the case where event *opening* occurs infinitely, T_{11} is never reached. If the slicing criterion for the EFSM in Figure 3 is T_{12} , then the marked transitions in the UNTICD slice are $\{T_5, T_{10}, T_{12}\}$, while in

the LG-NTSCD slice are $\{T3, T12\}$, in the NTSCD slice are $\{T3, T5, T10, T12\}$ and in the NTICD slice is $\{T12\}$.

8 Conclusions

In this paper, we introduced a non-termination insensitive form of control dependence for EFSM slicing, that built on the recent work of Ranganath et al. [RAB⁺05] and subsumed Korel et al.'s definition [KSTV03]. We demonstrated that by removing the fairness condition of Ranganath et al.'s NTICD no control dependences were removed, but extra control dependences within control sinks were introduced. Unlike NTICD our new definition works with non-terminating systems and, in general, produces smaller slices than those based on NTSCD.

References

- [BH04] Binkley, D., Harman, M.: A survey of empirical results on program slicing. *Advances in Computers* 62, 105–178 (2004)
- [CCD98] Canfora, G., Cimitile, A., De Lucia, A.: Conditioned program slicing. *Information and Software Technology* 40(11), 595–607 (1998)
- [De 01] De Lucia, A.: Program slicing: Methods and applications. In: *International Workshop on Source Code Analysis and Manipulation*, pp. 142–149. IEEE Computer Society Press, Los Alamitos (2001)
- [GGRT06] Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) *TestCom 2006*. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006)
- [Har87] Harel, D.: Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8(3), 231–274 (1987)
- [HBD03] Harman, M., Binkley, D., Danicic, S.: Amorphous program slicing. *Journal of Systems and Software* 68(1), 45–64 (2003)
- [HRB90] Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12(1), 26–61 (1990)
- [HTW98] Heimdahl, M.P.E., Thompson, J.M., Whalen, M.W.: On the effectiveness of slicing hierarchical state machines: A case study. In: *EUROMICRO 1998: Proceedings of the 24th Conference on EUROMICRO*, p. 10435. IEEE Computer Society, Washington (1998)
- [HW97] Heimdahl, M.P.E., Whalen, M.W.: Reduction and slicing of hierarchical state machines. In: Jazayeri, M. (ed.) *ESEC 1997 and ESEC-FSE 1997*. LNCS, vol. 1301, pp. 450–467. Springer, Heidelberg (1997)
- [Kam93] Kamkar, M.: Interprocedural dynamic slicing with applications to debugging and testing. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden (1993)
- [KSTV03] Korel, B., Singh, I., Tahat, L., Vaysburg, B.: Slicing of state-based models. In: *Proceedings of the International Conference on Software Maintenance*, pp. 34–43 (2003)

- [LG08] Labbé, S., Gallois, J.-P.: Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing* (2008)
- [LGP07] Labbe, S., Gallois, J.-P., Pouzet, M.: Slicing communicating automata specifications for efficient model reduction. In: *Proceedings of ASWEC*, pp. 191–200. IEEE Computer Society, Los Alamitos (2007)
- [LH07] Van Langenhove, S., Hoogewijs, A.: SV_tL : System verification through logic tool support for verifying sliced hierarchical statecharts. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) *WADT 2006*. LNCS, vol. 4409, pp. 142–155. Springer, Heidelberg (2007)
- [LHHR94] Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering* 20(9), 684–706 (1994)
- [RAB⁺05] Ranganath, V.P., Amtoft, T., Banerjee, A., Dwyer, M.B., Hatcliff, J.: A new foundation for control-dependence and slicing for modern program structures. In: Sagiv, M. (ed.) *ESOP 2005*. LNCS, vol. 3444, pp. 77–93. Springer, Heidelberg (2005)
- [RAB⁺07] Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.* 29(5), 27 (2007)
- [SW99] Strobl, F., Wisspeintner, A.: Specification of an elevator control system – an autofocus case study. Technical Report TUM-I9906, Technische Universität München (1999)
- [Tip95] Tip, F.: A survey of program slicing techniques. *Journal of Programming Languages* 3(3), 121–189 (1995)
- [WDQ02] Wang, J., Dong, W., Qi, Z.-C.: Slicing hierarchical automata for model checking UML statecharts. In: George, C.W., Miao, H. (eds.) *ICFEM 2002*. LNCS, vol. 2495, pp. 435–446. Springer, Heidelberg (2002)
- [Wei81] Weiser, M.: Program slicing. In: 5th International Conference on Software Engineering, pp. 439–449, San Diego, CA (March 1981)

Proving Consistency of Pure Methods and Model Fields

K. Rustan M. Leino¹ and Ronald Middelkoop²

¹ Microsoft Research, Redmond, USA
leino@microsoft.com

² Technische Universiteit Eindhoven, Holland
r.middelkoop@tue.nl

Abstract. Pure methods and model fields are useful and common specification constructs that can be interpreted by the introduction of axioms in a program verifier’s underlying proof system. Care has to be taken that these axioms do not introduce an inconsistency into the proof system. This paper describes and proves sound an approach that ensures no inconsistencies are introduced. Unlike some previous syntax-based approaches, this approach is based on semantics, which lets it admit some natural but previously problematical specifications. The semantic conditions are discharged by the program verifier using an SMT solver, and the paper describes heuristics that help avoid common problems in finding witnesses with trigger-based SMT solvers. The paper reports on the positive experience with using this approach in Spec# for over a year.

1 Introduction

Pure methods and model fields [1,2] are useful and common specification constructs. By marking a method as pure, the specifier indicates that it can be treated as a function of the state. It can then be called in specifications. Model fields provide a way to abstract from an object’s concrete data. A problem with either technique is that it can introduce an inconsistency into the underlying proof system. In this paper, we discuss how to prove (automatically) that no such inconsistency is introduced while allowing a rich set of specifications.

Starting from a review of the setting, the problem, and previous solutions, this section leads up to an overview of our contributions.

Pure Method Specifications. Figure 1 shows the template for a pure method specification (for simplicity, we show only a single formal parameter, named **p**). As usual, **requires** declares the method’s precondition P , **ensures** declares the method’s postcondition Q , and **result** denotes the method’s return value. The only free variables allowed in P are **this** and **p**. In Q , **result** is allowed as well.

A Deduction System. Marking method m as pure adds an uninterpreted total function $\#m : C \times T' \rightarrow T$ (a *method function* [3]) to the specification language. In predicates in the specification, the expression $E_0.m(E_1)$ is treated as syntactic

<pre> pure T $m(T' \text{ } p)$ requires P; ensures Q; </pre>	<pre> pure int bad() ensures false; { return 4; } </pre>	<pre> pure int n(int i) ensures result = this.p(i); pure int p(int i) ensures result = this.n(i)+1; </pre>
Fig. 1. Template	Fig. 2. Inconsistency	Fig. 3. Harmful indirect recursion

```

class Node {
  Object val;
  rep Node next;
  pure int count(Object obj)
    ensures result = (obj = this.val ? 1 : 0) +
                  (this.next = null ? 0 : this.next.count(obj));
  pure bool has(Object obj)
    ensures result = this.count(obj) > 0;
} //rest of class omitted

```

Fig. 4. Singly linked list (see Sect. 4.1 for **rep**)

sugar for $\#m(E_0, E_1)$. Furthermore, method function $\#m$ is axiomatized in the underlying deduction system for first-order logic by the following axiom:¹

$$\forall \sigma \in \Sigma \bullet \llbracket \forall \text{this}: C, p: T' \bullet P \Rightarrow Q[\#m(\text{this}, p)/\text{result}] \rrbracket \sigma \quad (1)$$

Here, Σ denotes the set of well-formed program states. Partial function $\llbracket E \rrbracket \sigma$ evaluates expression E to its value in state σ . $\llbracket \#m(E_0, E_1) \rrbracket \sigma$ is defined as $\#m(\llbracket E_0 \rrbracket \sigma, \llbracket E_1 \rrbracket \sigma)$. Other details of this evaluation are unimportant here. $P[E/v]$ denotes the predicate like P , but with capture-avoiding substitution of variable v by E . For instance, pure method **has** from Fig. 4 introduces uninterpreted total function $\#has : \text{Node} \times \text{Object} \rightarrow \text{bool}$, and axiom $\forall \sigma \in \Sigma \bullet \llbracket \forall \text{this}: \text{Node}, \text{obj}: \text{bool} \bullet \#has(\text{this}, \text{obj}) = \#count(\text{this}, \text{obj}) > 0 \rrbracket \sigma$.

Consistency of Deduction System. If one is not careful, pure methods can introduce an inconsistency into the deduction system. As an obvious example, consider Fig. 2. This definition introduces *false* as an axiom into the deduction system (more precisely, it introduces $\forall \sigma \in \Sigma \bullet \llbracket \forall \text{this}: C \bullet \text{false} \rrbracket \sigma$). So, it has to be ensured that for all possible values of the arguments of method function $\#m$, there is a value that the function can take. Insuring this by requiring a proof of total correctness of the implementation of m before adding the axiom is highly impractical. If $\#m$ is constrained only by the axiom introduced by m , then it suffices to prove property (2):

$$\forall \sigma \in \Sigma \bullet \llbracket \forall \text{this}: C, p: T' \bullet \exists x: T \bullet P \Rightarrow Q[x/\text{result}] \rrbracket \sigma \quad (2)$$

If other axioms can also constrain $\#m$, as is the case in the presence of mutual recursion, then property (2) needs to simultaneously mention all methods involved. We aim for sound *modular verification*, which means being able

¹ The axiomatization differs slightly in the presence of class invariants. To simplify the presentation, invariants are not considered.

<pre> pure int findInsertionPosition(int N) requires $0 \leq N$; ensures $0 \leq \text{result} \wedge \text{result} \leq N$; </pre>	<pre> pure int max(int x, int y) ensures $(x \leq y \Rightarrow \text{result} = y) \wedge$ $(y \leq x \Rightarrow \text{result} = x)$; </pre>

Fig. 5. Previous syntactic checks forbid these methods; our semantic checks allow them.

<pre> pure bool isEven(int n) requires $0 \leq n$; ensures result = ($n = 0 ? \text{true} : \text{this.isOdd}(n-1)$); measuredBy $2n$; </pre>	<pre> pure bool isOdd(int m) requires $0 \leq m$; ensures result $\neq \text{this.isEven}(m)$; measuredBy $2m+1$; </pre>

Fig. 6. Odd and even (see Sect. 4.3 for **measuredBy**)

to verify a program's modules separately, just like a compiler performs separate compilation of modules. If the mutual recursion can span module boundaries, then there may be no verification scope that has information about all the methods that need to be simultaneously mentioned. Therefore, the consistency of mutual recursion among pure methods is usually stated in a form different from (2).

Previous Solutions. Darvas and Müller [4] prove that inconsistency is prevented if the following two measures are taken: (A) the axiom that is introduced into the deduction system for a method function $\#m$ is not proposition (1), but $(2) \Rightarrow (1)$, and (B) recursion in the pure method axioms is disallowed unless it is direct and well-founded. For example, measure A prevents the pure methods in Fig. 2 from introducing an inconsistency, and measure B forbids the specifications in Fig. 3, whose axioms would otherwise introduce an inconsistency.

Darvas and Leino [3] discuss a problem with measure A, namely that an axiom of the form $(2) \Rightarrow (1)$ is not suitable for automatic reasoning using today's trigger-based SMT solvers like Simplify and Z3 [5,6]. More specifically, these solvers are unable to come up with a witness for the existential quantification in (2) even in simple cases. This means that property (1) is 'silently ignored', which renders the pure method useless (and possibly confuses the user).

To circumvent the practical problem with measure A, Darvas and Leino introduce a simple syntactic check that allows one to conclude that (2) holds once and for all [3]. Thus, (1) can be introduced as an axiom into the deduction system without fear of inconsistencies. However, the syntactic check is restrictive and prevents a number of natural and useful specifications, including the two in Fig. 5. Syntactic checks cannot guarantee the consistency of **findInsertionPosition**, because its result value is constrained by two inequalities, or of **max**, because its result-value constraints are guarded by antecedents.

Measure B is a Draconian way of dealing with mutual recursion. The syntactic check of Darvas and Leino [3] improves on this situation. However, this check is still restrictive; for instance, it does not permit the example in Fig. 6.

```

class Rectangle {
  int x1,y1,x2,y2; //lower left and upper right corner
  model int width satisfies this.width = this.x2-this.x1;
  model int height satisfies this.height = this.y2-this.y1;
  void scaleH(int factor)
    requires 0 ≤ factor;
    ensures this.width = old(this.width) * factor/100;
    { this.x2 := (this.x2 - this.x1) * factor/100 + this.x1; }
} //rest of class omitted

```

Fig. 7. Model fields

A Glimpse of Our Semantic Solution. In our solution, we use heuristics to guess candidate witness expressions for (2). Then we verify that in every program state allowed by the pure method's precondition, one of these candidates establishes the postcondition. For example, for pure method `max` in Fig. 5, we generate three candidate witnesses `1`, `x`, and `y`, and construct a program snippet of the form:

```

r := 1; if ((x ≤ y ⇒ r = y) ∧ (y ≤ x ⇒ r = x)) { return r; }
r := x; if ((x ≤ y ⇒ r = y) ∧ (y ≤ x ⇒ r = x)) { return r; }
r := y; return r;

```

and then attempt to verify, using our program verifier's machinery, that this program snippet establishes the postcondition of the pure method.

Model Fields. Model fields introduce similar problems. A model field gives a way to hide details of an object's concrete state. Figure 7 gives an example (taken from [7]) of the use of model fields: by updating the satisfies clauses, e.g., to `this.width = this.w` and `this.height = this.h`, `Rectangle` can be re-implemented with two `ints` `w` and `h`, without affecting the verification of other classes. For every model field **model** T f **satisfies** Q in a class C , a total function $\#f : C \rightarrow T$ (an *abstraction function*) is added to the specification language. In predicates in the specification, the expression $E.f$ is treated as syntactic sugar for $\#f(E)$. Abstraction function $\#f$ is axiomatized in the deduction system by an axiom $\forall \sigma \in \Sigma \bullet \llbracket \forall \text{this} : C \bullet Q \rrbracket \sigma$.² This axiom is not visible outside of C 's module. The axiomatization problems we have described for method functions apply to abstraction functions as well: for the purpose of this paper, a model field f that satisfies predicate Q can be treated as a parameterless pure method with postcondition Q , with **result** for `this.f`.

Contributions. The contributions of this paper are the following:

1. We formalize and strengthen an implicit claim from [3]: No inconsistency is introduced by axioms of the form (2) $\Rightarrow$ (1) if every method function call in a pure method m 's specification lies below m in a partial order $\prec$ (Sect. 2).

² More axioms might be added depending on the methodology, see Sect. 5.

2. We present a much improved scheme that leverages the power of the theorem prover to prove (2) once-and-for-all (Sect. 3).
3. We introduce a permissive definition for $\prec$ that improves the one in [3] and allows a greater degree of (mutual) recursion than before (Sect. 4).

We report on our experience and discuss related work in Sect. 5.

2 Avoiding Inconsistency

In this section, we identify proof obligations that allow axioms of form (1) to be added to the deduction system without introducing inconsistencies.

Let there be $N+1$ pure methods in the program fragment that is to be verified, labeled $m_0, \dots, m_N$. For simplicity, assume that there are no static pure methods and that every pure method m_i has exactly one formal parameter $\mathbf{p}_i$ of type T'_i (extending to an arbitrary number of parameters is straightforward). Let T_i be the return type of pure method m_i . Let C_i be the class that defines m_i . Let predicates Pre_i and $Post_i$ be the pre- and postconditions of m_i . $PureAx$, defined below, represents the axioms introduced by pure methods (reformulated into a single proposition). We use $\equiv$ to define syntactical shorthands.

Definition 1 ($PureAx$).

$$\begin{aligned} Spec_i &\equiv Pre_i \Rightarrow Post_i \\ MSpec_i &\equiv \forall \mathbf{this}: C_i, \mathbf{p}_i: T'_i \bullet Spec_i[\#m_i(\mathbf{this}, \mathbf{p}_i)/\mathbf{result}] \\ PureAx &\equiv \forall \sigma \in \Sigma \bullet \llbracket MSpec_0 \wedge \dots \wedge MSpec_N \rrbracket \sigma \end{aligned}$$

Let *Prelude* be the conjunction of all axioms in the deduction system that are not introduced by a pure method. The goal is to find proof obligations POs such that if *Prelude* is consistent and POs hold, then adding the axioms for pure methods does not introduce inconsistencies. Theorem 1 formalizes this goal:

Theorem 1. $Prelude \Rightarrow (POs \Rightarrow PureAx)$

The remainder of this section discusses the proof obligations POs that we use to ensure that Thm. 1 holds. The theorem itself is proven to hold in the accompanying technical report [8]. If there is no recursion in pure method specifications, then Thm. 1 can be shown to hold using $POs \equiv PO1$ (see [4]):

Definition 2 ($PO1$).

$$\begin{aligned} PO1_i &\equiv \forall \sigma \in \Sigma \bullet \llbracket \forall \mathbf{this}: C_i, \mathbf{p}_i: T'_i \bullet \exists \mathbf{result}: T_i \bullet Spec_i \rrbracket \sigma \\ PO1 &\equiv PO1_0 \wedge \dots \wedge PO1_N \end{aligned}$$

Note that $PO1_i$ is equivalent to proposition (2) from the introduction.

When there is (mutual) recursion, the crucial property that is in jeopardy is *functional consistency*: if the same function is called twice from the same state and the parameters of the two calls evaluate to the same values, then the two calls evaluate to the same value. For instance, consider the methods in Fig. 3. If pure methods add propositions of the form (1) to the deduction system, then these method definitions allow one to deduce that $\#n(\mathbf{this}, \mathbf{i})$

$= \#n(\mathbf{this}, i) + 1$, which contradicts functional consistency of $\#n$. More formally, since $\llbracket \#m_i(E_0, E_1) \rrbracket \sigma = \#m_i(\llbracket E_0 \rrbracket \sigma, \llbracket E_1 \rrbracket \sigma)$ (see Sect. 1), it follows immediately that $\forall \sigma \in \Sigma, i \in [0, N] \bullet \llbracket \forall c_0, c_1: C_i, p_0, p_1: T'_i \bullet c_0 = c_1 \wedge p_0 = p_1 \Rightarrow \#m_i(c_0, p_0) = \#m_i(c_1, p_1) \rrbracket \sigma$. The proof obligations must ensure that the axioms introduced by pure methods do not contradict functional consistency.

For convenience, we define the equivalence relation $\sim$:

Definition 3 ($\sim$).

$$\llbracket \#m_i(E_0, E_1) \sim \#m_j(E_2, E_3) \rrbracket \sigma \stackrel{\text{def}}{=} i = j \wedge \llbracket E_0 = E_2 \wedge E_1 = E_3 \rrbracket \sigma$$

Then $\forall \sigma \in \Sigma \bullet \llbracket \#m_i(E_0, E_1) \sim \#m_j(E_2, E_3) \Rightarrow \#m_i(E_0, E_1) = \#m_j(E_2, E_3) \rrbracket \sigma$.

To ensure that recursive specifications do not lead to an axiomatization that contradicts functional consistency, we require the verifier to ensure that a function call in the axiomatization of $\#m_i(o, x)$ does not (indirectly) depend on the value of $\#m_i(o, x)$. To this end, we introduce the strict partial order $\prec$ on method function calls (i.e., $\prec$ is an irreflexive and transitive binary relation on expressions of the shape $\#m_i(E_0, E_1)$). The definition of $\prec$ is not relevant to the proof as long as (1) $\prec$ is well-founded, and (2) the following lemma holds:

Lemma 1. $\forall \sigma \in \Sigma, i, j \in [0, N] \bullet \llbracket \forall c_0: C_i, x_0: T'_i, c_1: C_j, x_1: T'_j \bullet \#m_i(c_0, x_0) \prec \#m_j(c_1, x_1) \Rightarrow \#m_i(c_0, x_0) \not\prec \#m_j(c_1, x_1) \rrbracket \sigma$

In Sect. 4, we present a definition of $\prec$ that is suitable for our proof system. Proof obligation *PO2*, defined below, requires every method function call in the specification of m_i to lie below $\#m_i(\mathbf{this}, \mathbf{p}_i)$ in the order $\prec$ in every state in which the result of the call is relevant.

Definition 4 (*PO2*). Let $i, j \in [0, N]$. Let $\text{NrOfCalls}_{i,j}$ be the number of calls to $\#m_j$ in Spec_i . If $l + 1 = \text{NrOfCalls}_{i,j}$, and $k \in [0, l]$, then

$\text{Call}_{i,j,k}$ is the expression that is the k 'th call to $\#m_j$ in Spec_i

$\text{Spec}_{i,j,k}$ is Spec_i , but with a fresh variable substituted for the k 'th call to $\#m_j$

$\text{Smaller}_{i,j,k} \equiv \text{Call}_{i,j,k} \prec \#m_i(\mathbf{this}, \mathbf{p}_i)$

$\text{NotRel}_{i,j,k} \equiv \forall \mathbf{result}: T_j, x: T'_j \bullet \text{Spec}_{i,j,k} = \text{Spec}_i$

$\text{PO2}_{i,j,k} \equiv \forall \sigma \in \Sigma \bullet \llbracket \forall \mathbf{this}: C_i, \mathbf{p}_i: T'_i \bullet \text{Smaller}_{i,j,k} \vee \text{NotRel}_{i,j,k} \rrbracket \sigma$

$\text{PO2}_{i,j} \equiv \text{PO2}_{i,j,0} \wedge \dots \wedge \text{PO2}_{i,j,l}$

$\text{PO2}_i \equiv \text{PO2}_{i,0} \wedge \dots \wedge \text{PO2}_{i,N}$

$\text{PO2} \equiv \text{PO2}_0 \wedge \dots \wedge \text{PO2}_N$

The intuition behind *NotRel* is that $\text{Call}_{i,j,k}$ in Spec_i is not relevant in $\sigma \in \Sigma$ if the result value of $\text{Call}_{i,j,k}$ is not relevant to the value of $\#m_i(\mathbf{this}, \mathbf{p}_i)$ in σ . That is, for any value of **result**, the value of Spec_i is the same for any result of $\text{Call}_{i,j,k}$. As an extreme example, suppose Spec_i is $\mathbf{false} \Rightarrow \mathbf{result} = \mathbf{this}.m_i(\mathbf{p}) + 1$. Then $\text{Smaller}_{i,i,0}$ never holds, but $\text{NotRel}_{i,i,0}$ always holds as $\forall \sigma \in \Sigma \bullet \llbracket \forall \mathbf{this}: C_i, \mathbf{p}_i: T'_i, \mathbf{result}: T_j, x: T'_j \bullet (\mathbf{false} \Rightarrow \mathbf{result} = \mathbf{this}.m_i(\mathbf{p}) + 1) = (\mathbf{false} \Rightarrow \mathbf{result} = x + 1) \rrbracket \sigma$. Then $\text{PO2}_{i,i,0}$ is met, and hence PO2_i is met. We show a more realistic example in Sect. 4.1.

In this section, we formalized the problem sketched in the introduction. Furthermore, we introduced high-level proof obligations that ensure that the extension of the *Prelude* with the axiomatization of pure methods does not introduce

inconsistencies: in [8] we prove that Thm. 1 holds if $POs \equiv PO1 \wedge PO2$. In the next two sections, we address two remaining practical concerns: we provide heuristics to prove $PO1$, and define the partial ordering $\prec$ used in $PO2$.

3 Heuristics for Establishing $PO1$

Proof obligation $PO1$ poses serious difficulties for automatic verification. Even in simple cases, automatic theorem provers are unable to come up with a witness for the existential quantification $\exists \mathbf{result} : T_i \bullet \mathit{Spec}_i$ in $PO1_i$. As a solution, [3] proposes only to allow a pure method m_i when (1) it has a postcondition of the form $\mathbf{result} \text{ op } E$ or $E \text{ op } \mathbf{result}$, where op is a binary operator from the set $\{=, \geq, \leq, \Rightarrow, \Leftrightarrow\}$, and (2) E is an expression that does not contain $\mathbf{result}$. If these conditions are met, then E is a witness for the quantification, i.e., $\forall \sigma \in \Sigma \bullet \llbracket \forall \mathbf{this} : C_i, \mathbf{p}_i : T'_i \bullet \mathit{Spec}_i[E/\mathbf{result}] \rrbracket \sigma$, and therefore $PO1_i$ holds.

This solution has the advantage that it only requires a simple syntactic check. However, it is quite restrictive. Unfortunately, not much more can be done with syntactic checks. For instance, consider method `findInsertPosition` from Fig. 5. Here, 0 is a witness (as $0 \leq N \Rightarrow 0 \leq 0 \wedge 0 \leq N$). However, a syntactic check cannot establish that $0 \leq N$. Our solution is to leverage the power of the theorem prover. Consider the scheme below.

1. Find a witness candidate E .
2. If $\forall \sigma \in \Sigma \bullet \llbracket \forall \mathbf{this} : C_i, \mathbf{p}_i : T'_i \bullet \mathit{Spec}_i[E/\mathbf{result}] \rrbracket \sigma$ can be established by the theorem prover, then $PO1_i$ holds. Otherwise, the program is rejected.

This scheme is more powerful than the syntactic check of [3]. For instance, it allows `findInsertPosition`, assuming that 0 is found as a witness candidate. Before we discuss how to find witness candidates, we improve on the scheme above in one important way. Consider method `max` from Fig. 5. $PO1$ cannot be established for `max` using the scheme above, no matter which witness candidate is found. In particular, neither $\mathit{Spec}_{\max}[\mathbf{x}/\mathbf{result}]$ nor $\mathit{Spec}_{\max}[\mathbf{y}/\mathbf{result}]$ holds. The problem is that the scheme requires that there is a witness that holds in all cases. $PO1$ only requires that in all cases, there is a witness. The latter is true for `max`, but the former is not. If $\mathbf{x} \leq \mathbf{y}$, then $\mathbf{y}$ is a witness. If $\mathbf{y} \leq \mathbf{x}$, then $\mathbf{x}$ is a witness. That is, $\mathit{Spec}_{\max}[\mathbf{x}/\mathbf{result}] \vee \mathit{Spec}_{\max}[\mathbf{y}/\mathbf{result}]$ holds. Therefore, $\exists \mathbf{result} : \mathbf{int} \bullet \mathit{Spec}_{\max}$ holds, and $PO1$ holds. Based on this reasoning, the scheme presented above is replaced by the more liberal scheme below.

1. Find witness candidates $E_0, \dots, E_n$.
2. If $\forall \sigma \in \Sigma \bullet \llbracket \forall \mathbf{this} : C_i, \mathbf{p}_i : T'_i \bullet \mathit{Spec}_i[E_0/\mathbf{result}] \vee \dots \vee \mathit{Spec}_i[E_n/\mathbf{result}] \rrbracket \sigma$ can be established by the theorem prover, then $PO1_i$ holds. Otherwise, the program is rejected.

Next, we present an algorithm to find witness candidates for a pure method. We assume that there is a function $\mathit{kind} : \mathit{Type} \rightarrow \{\mathit{Bool}, \mathit{Enum}, \mathit{Num}, \mathit{Ref}\}$ that distinguishes four kinds of types. The algorithm uses a Haskell-like switch that uses pattern matching and does not fall through. For example, case A of $B \rightarrow C$ $D \rightarrow E$ $\_ \rightarrow F$ should be read as ‘if A matches B, then C, else if A matches D,

then E , else F '. The witness candidates for a pure method m_i with return type T_i and postcondition $Post_i$ are given by $wcs(T_i, Post_i)$. Below, wcs and its helper functions are defined, discussed and illustrated by a number of examples. Note that $ExprSet \equiv Set\ of\ Expression$, and that $|S|$ returns the size of set S .

Definition 5 (wcs).

$wcs : Type \times Predicate \rightarrow ExprSet$

$wcs(T, P) \stackrel{def}{=} \text{case } kind(T) \text{ of}$

$Bool \rightarrow \{true, false\}$

$Enum \rightarrow \text{the enumerator list (i.e. the sequence of enumeration constants) of } T$

$Ref \rightarrow \text{let } euld(P) = \langle S_0, S_1, S_2, S_3 \rangle \text{ in } S_0 \cup \{\mathbf{null}\}$

$Num \rightarrow \text{let } euld(P) = \langle S_0, S_1, S_2, S_3 \rangle \text{ in}$

$S_0 \cup \text{dupl}(S_1, |S_3|, true) \cup \text{dupl}(S_2, |S_3|, false) \cup \text{dupl}(\{1\}, |S_3|, true)$

$euld : Predicate \rightarrow ExprSet \times ExprSet \times ExprSet \times ExprSet$

$euld(P) \stackrel{def}{=} \text{case } P \text{ of}$

result = E or E = **result** $\rightarrow \langle \{E\}, \{\}, \{\}, \{\} \rangle$

result $\geq E$ or $E \leq$ **result** $\rightarrow \langle \{\}, \{E\}, \{\}, \{\} \rangle$

result $\leq E$ or $E \geq$ **result** $\rightarrow \langle \{\}, \{\}, \{E\}, \{\} \rangle$

result $\neq E$ or $E \neq$ **result** $\rightarrow \langle \{\}, \{\}, \{\}, \{E\} \rangle$

result $> E$ or $E <$ **result** $\rightarrow euld(\mathbf{result} \geq E + 1)$

result $< E$ or $E >$ **result** $\rightarrow euld(\mathbf{result} \leq E - 1)$

$P_0 \vee P_1$ or $P_0 \wedge P_1 \rightarrow \text{let } euld(P_0) = \langle S_0, S_1, S_2, S_3 \rangle$

$\text{and } euld(P_1) = \langle S'_0, S'_1, S'_2, S'_3 \rangle \text{ in}$

$\langle S_0 \cup S'_0, S_1 \cup S'_1, S_2 \cup S'_2, S_3 \cup S'_3 \rangle$

$\neg P_0 \rightarrow \text{let } euld(P_0) = \langle S_0, S_1, S_2, S_3 \rangle \text{ in}$

$\langle S_3, \text{addOrSub1}(S_2, true), \text{addOrSub1}(S_1, false), S_0 \rangle$

$P_0 \Rightarrow P_1$ or $P_1 \Leftarrow P_0 \rightarrow euld(\neg P_0 \vee P_1)$

$P_0 \Leftrightarrow P_1 \rightarrow euld((P_0 \wedge P_1) \vee (\neg P_0 \vee \neg P_1))$

$P_0 ? P_1 : P_2 \rightarrow euld((P_0 \Rightarrow P_1) \wedge (\neg P_0 \Rightarrow P_2))$

$- \rightarrow \langle \{\}, \{\}, \{\}, \{\} \rangle$

$\text{addOrSub1} : ExprSet \times Bool \rightarrow ExprSet$

$\text{addOrSub1}(\{E_0, \dots, E_n\}, isAdd) \stackrel{def}{=} \{isAdd ? \{E_0 + 1, \dots, E_n + 1\} : \{E_0 - 1, \dots, E_n - 1\}\}$

$\text{dupl} : ExprSet \times \mathbb{N} \times Bool \rightarrow ExprSet$

$\text{dupl}(\{E_1, \dots, E_n\}, \text{duplCnt}, isAdd) \stackrel{def}{=} \text{duplExpr}(E_1, \text{duplCnt}, isAdd) \cup \dots \cup \text{duplExpr}(E_n, \text{duplCnt}, isAdd)$

$\text{duplExpr} : Expression \times \mathbb{N} \times Bool \rightarrow ExprSet$

$\text{duplExpr}(E, \text{duplCnt}, isAdd) \stackrel{def}{=} \{isAdd ? \{E + 0, \dots, E + \text{duplCnt}\} : \{E - 0, \dots, E - \text{duplCnt}\}\}$

The intuition behind the $wcs(T, P)$ definition is as follows. If $kind(T) \in \{Bool, Enum\}$, then there is no need to scan the postcondition for witness candidates. Instead, we make full use of the possibility to select multiple candidates and let

every value of the type be a witness candidate. If $\text{kind}(T) \in \{\text{Num}, \text{Ref}\}$, then function *euld* is used to scan P for equalities, upper bounds, lower bounds, and disequalities that contain **result**. More precisely, assume $\text{euld}(P) = (S_0, S_1, S_2, S_3)$. Let $\text{cnf}(P)$ yield the conjunctive normal form of P , and let $\text{test}(P, E) \equiv \text{cnf}(P)[E/\text{result}]$. Let E_0, E_1, E_2 and E_3 be elements of S_0, S_1, S_2 and S_3 , respectively. Then for every $n \in \mathbb{N}$, each of $\text{test}(P, E_0)$, $\text{test}(P, E_1 + n)$ and $\text{test}(P, E_2 - n)$ has a satisfied conjunct. Also, at least one conjunct of $\text{test}(P, E_3)$ contains an unsatisfied disjunct. From *euld*'s result, witness candidates are extracted and where needed duplicated using function *dupl*.

We illustrate with several examples. Let $\text{kind}(T_i) = \text{Num}$. If Post_i is **result** = 4, or **result** > 3, or **result** ≤ 4, then $\text{euld}(\text{Post}_i)$ is $\langle \{4\}, \{\}, \{\}, \{\} \rangle$, $\langle \{\}, \{4\}, \{\}, \{\} \rangle$, or $\langle \{\}, \{\}, \{4\}, \{\} \rangle$, respectively. In each case, $\text{wcs}(T_i, \text{Post}_i) = \{4, 1\}$. As $\text{Post}_i[4/\text{result}]$ holds, $\text{Post}_i[4/\text{result}] \vee \text{Post}_i[1/\text{result}]$ holds as well and PO1_i is satisfied. Default witness 1 is included to handle, e.g., the case where Post_i is **result** ≠ 4. Then $\text{euld}(\text{Post}_i) = \langle \{\}, \{\}, \{\}, \{4\} \rangle$. Then $\text{wcs}(T_i, \text{Post}_i) = \{1, 2\}$. As $\text{Post}_i[1/\text{result}]$ holds, PO1_i is satisfied.

We track upper and lower bounds and the number of disequalities N to handle, e.g., the case where Post_i is **result** > 4 ∧ **result** ≠ 5. Then $\text{euld}(\text{Post}_i) = \langle \{\}, \{5\}, \{\}, \{5\} \rangle$, and $\text{wcs}(T_i, \text{Post}_i) = \{5, 6, 1, 2\}$. As $\text{Post}_i[6/\text{result}]$ holds, PO1_i is satisfied. More generally, by trying N different candidates that all satisfy the bound, we are sure to find at least one that satisfies the disequality.

We combine the candidates found in subpredicates of conjunctions and disjunctions to handle, e.g., the case where Post_i is (**result** = 4 ∨ **result** > 8) ∧ **result** > 7. Then $\text{euld}(\text{Post}_i) = \langle \{4\}, \{9, 8\}, \{\}, \{\} \rangle$, and $\text{wcs}(T_i, \text{Post}_i) = \{4, 9, 8, 1\}$. As $\text{Post}_i[9/\text{result}]$ holds, PO1_i is satisfied.

A predicate $\neg P$ is dealt with 'on the fly', which is more efficient than distributing the negation over the subexpressions of P . We interchange S_0 and S_3 as well as S_1 and S_2 , and then add (subtract) 1 to each element of the new S_1 (S_2). The intuition is the following. As was stated above, if $E \in S_1$, then for every n a conjunct in $\text{test}(P, E + n)$ holds. Then for every n , a conjunct in $\text{test}(\neg P, E - 1 - n)$ holds. For example, $\neg(\text{result} \geq E)$ equals **result** ≤ $E - 1$.

As an aside, note that in the cases where P is either $P_0 \Leftrightarrow P_1$ or $P_0 ? P_1 : P_2$, $\text{euld}(P_0)$ and $\text{euld}(P_1)$ are evaluated twice. These cases can be optimized at the expense of a more complicated definition.

4 Defining the Ordering $\prec$

Our definition of $\prec$ builds on work in [3,4]. It uses a function *Order* (defined below) that associates a tuple of numbers with an expression $\#m_i(E_0, E_1)$ in a state σ . Our definition of $\prec$ ensures that $\prec$ is a well-founded strict partial order, and that Lem. 1 holds (as long as *Order* is well-defined):

Definition 6 ($\prec$). $\llbracket \#m_i(E_0, E_1) \prec \#m_j(E_2, E_3) \rrbracket \sigma \stackrel{\text{def}}{=}$

Order($\#m_i, E_0, E_1, \sigma$) is lexicographically ordered below *Order*($\#m_j, E_2, E_3, \sigma$)

As Def. 7 shows, the definition of *Order* uses three functions. *RootDistance* associates a number with an object based on the well-founded strict partial order

on objects provided by ownership, an existing specification technique (Sect. 4.1). *RTVal* associates a number with a method function based on a numbering scheme that can be largely inferred automatically (Sect. 4.2). *MeasuredBy* yields a tuple of numbers that is determined by a pure method's **measuredBy** clause, and that depends only on the values of the numerical parameters (Sect. 4.3). The definition uses $\triangleright$ to denote sequence concatenation.

Definition 7 (*Order*)

Order : *Method Function* $\times$ *Expression* $\times$ *Expression* $\times$ $\Sigma \rightarrow$ *Sequence of* $\mathbb{Z}$

$$\text{Order}(\#m_i, E_0, E_1, \sigma) \stackrel{\text{def}}{=} \langle j, \text{RTVal}(\#m_i) \rangle \triangleright \text{MeasuredBy}(\#m_i, E_1, \sigma),$$

where j is $(\llbracket E_0 \rrbracket \sigma \in \text{Object} ? - \text{RootDistance}(\llbracket E_0 \rrbracket \sigma, \sigma) : 0)$.

Note that if $\#m_j(E_0, E_1)$ occurs in the specification of m_i , and σ does not map E_0 to an object, then the first element of *Order*($\#m_j, E_0, E_1, \sigma$) is 0, thus requiring that the call is not relevant in σ if *PO2* is to hold.

4.1 Root Distance

Ownership, originally developed to enforce state encapsulation [9,10], is a commonly used technique to make whole/part relations explicit in specifications (often applied to the modular verification of invariants [11,12,13,14]). The set of *owners* consists of the set of objects and the special purpose owner **root**. In any given state, every object x is *directly owned* by exactly one owner o , $o \neq x$. The *owned* relation is the transitive closure of the directly owned relation. The intention is that an object x owns the objects that are part of x , i.e., that belong to x 's representation. Objects that are not part of any other object are directly owned by **root**. The owned relation is required to be irreflexive, as a whole is not a part of one of its parts. Therefore, ownership is a well-founded strict partial order, which makes it suitable for use in the definition of $\prec$.

In [4], it is suggested that ‘the height of an object in the ownership hierarchy’ can be used to allow direct recursion. We formalize this notion and apply it to general recursion. The owned relation ensures that every object is owned by **root**. Let function *RootDistance* : *Object* $\times$ $\Sigma \rightarrow \mathbb{N}$ be such that *RootDistance*(x, σ) = n iff x is owned by exactly n objects in σ (we say x has *RootDistance* n in σ). Then *RootDistance* induces a well-founded strict partial order that is an extension of ownership: if object x is owned by object y in state σ , then *RootDistance*(x, σ) > *RootDistance*(y, σ). Additionally, *RootDistance* orders objects that are not ordered by ownership. For instance, if x and y have the same direct owner in state σ , and object z is owned by y , then x and z are not ordered by ownership, but *RootDistance*(x, σ) < *RootDistance*(z, σ).

Note that given Definitions 6 and 7, $\llbracket \#m_i(E_0, E_1) \prec \#m_j(E_2, E_3) \rrbracket \sigma = \text{true}$ when $\llbracket E_0 \rrbracket \sigma = x$, $\llbracket E_2 \rrbracket \sigma = y$, and *RootDistance*(x, σ) > *RootDistance*(y, σ).

It is not necessary, and usually not possible, to determine an object's absolute *RootDistance* during static program verification. Rather, if m_i 's specification contains a call $\#m_j(E_0, E_1)$, one has to establish that the *RootDistance* of the **this**-object is smaller than (or at least equal to) the *RootDistance* of the E_0 -object. I.e., one reasons about the relative *RootDistance*. This involves reasoning


```

class Holding {
  rep Node myComps;
  rep Personnel myPnel;
} //rest of class omitted

```

```

class Company {
  peer Personnel thePnel;
  pure Node myPersonnel()
  ensures  $\forall$  Person p • (
    result.has(p)  $\Leftrightarrow$ 
    this.thePnel.myPers.has(p)  $\wedge$ 
    p.worksFor  $\neq$  null  $\wedge$ 
    p.worksFor.has(this) );
} //rest of class omitted

```

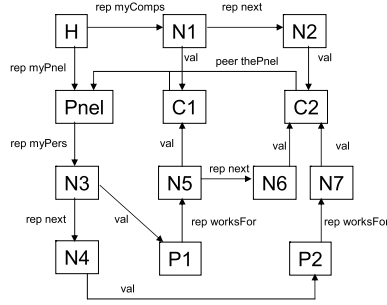


Fig. 8. Administration System. H is a Holding, Pnel a Personnel, N's are Nodes, C's Companies, and P's Persons. Person P2 works only for C2, and P1 works for both C's.

about ownership, which is often made explicit by extending types with *ownership modifiers* [15] like **rep** and **peer**. Consider a state σ in which an object x has a field f that refers to an object y . If the ownership modifier of f is **rep**, then x directly owns y and $\text{RootDistance}(y, \sigma) = \text{RootDistance}(x, \sigma) + 1$. If it is **peer**, then x and y have the same direct owner and $\text{RootDistance}(y, \sigma) = \text{RootDistance}(x, \sigma)$. Alternatively, ownership can be encoded into existing proof system concepts using a specification-only field **owner** [12]. If $x.\text{owner}$ evaluates to y in σ , then $\text{RootDistance}(x, \sigma) = \text{RootDistance}(y, \sigma) + 1$.

The use of *RootDistance* is illustrated by method **Node.count** in Fig. 4. Its specification contains one call, to $\#count(\text{this.next}, \text{obj})$. There are two cases, each of which satisfies *PO2*: if $\llbracket \text{this.next} = \text{null} \rrbracket \sigma = \text{false}$, modifier **rep** on **next** allows the verifier to deduce that $\text{RootDistance}(\llbracket \text{this.next} \rrbracket \sigma, \sigma) = \text{RootDistance}(\llbracket \text{this} \rrbracket \sigma, \sigma) + 1$; if $\llbracket \text{this.next} = \text{null} \rrbracket \sigma = \text{true}$, *NotRel* holds as $\llbracket (\text{this.next} = \text{null} ? 0 : \text{this.next.count}(\text{obj})) \rrbracket \sigma = 0$, which means that the value of $\llbracket \#count(\text{this.next}, \text{obj}) \rrbracket \sigma$ is not relevant.

The extension of ownership provided by *RootDistance* is useful for non-hierarchical scenarios. For instance, Fig. 8 shows two classes and a possible object configuration of an administration system. In this system, a **Holding** consists of multiple **Companies**, and a **Person** that is part of the **Holding** can work for multiple of these **Companies**. A **Personnel** object manages (access to) these **Persons**. Classes **Personnel** and **Person** are omitted. Each has only one relevant field. **Personnel** has a field **rep Node myPers** which refers to a linked list of the **Persons**. **Person** has a field **rep Node worksFor** which refers to a linked list of the **Companies** that **Person** works for. Class **Node** is found in Fig. 4. Pure method **Company.myPersonnel** returns a linked list of all **Persons** that work for that **Company** (e.g., if called on C1, it returns a single node with **val P1**). Assume that it can be deduced that **Company.thePnel** and **Personnel.thePers** are never null (for instance because of an invariant or non-null annotation [16]). Then the **this.thePnel.myPers.has(p)** call in **Company.myPersonnel** is allowed as

`myPers` is a rep field of a peer of `this` and thus has a higher *RootDistance*. More formally, in any state $\sigma \in \Sigma$ in which `this` evaluates to a `Company` object, $\text{RootDistance}(\llbracket \text{this.thePnel.myPers} \rrbracket \sigma, \sigma) = \text{RootDistance}(\llbracket \text{this} \rrbracket \sigma, \sigma) + 1$, and therefore, $\llbracket \# \text{has}(\text{this.thePnel.myPersons}, p) \prec \# \text{myPersonnel}(\text{this}) \rrbracket \sigma$ holds. Likewise, the `p.worksFor.has(p)` call is allowed if one can deduce that the `Persons` in the list maintained by `p.thePnel` are owned by `p.thePnel` or by the `Holding` that owns `p.thePnel`. We discuss the `result.has(p)` call in Sect. 5.

4.2 Recursion Termination Value

For the second ordering, a *Recursion Termination Value* (RTV) is associated with each pure method [3]. A RTV is an element of the interval $[0, \text{maxRTV}]$, where maxRTV is a sufficiently large constant, e.g. maxInt . $\text{RTVal}(\#m_i)$ yields the RTV associated with pure method m_i .

Note that given Definitions 6 and 7, $\llbracket \#m_i(E_0, E_1) \prec \#m_j(E_2, E_3) \rrbracket \sigma = \text{true}$ when $\text{RootDistance}(\llbracket E_0 \rrbracket \sigma, \sigma) = \text{RootDistance}(\llbracket E_2 \rrbracket \sigma, \sigma)$ and $\text{RTVal}(\#m_i) < \text{RTVal}(\#m_j)$.

The RTV can be specified explicitly. For instance, in *Spec#* the RTV is specified by the `RecursionTermination` attribute that takes an integer parameter. The main advantage of the RTV ordering, however, is that it is largely inferred automatically. This inference is complicated by the desire for modular development (see Sect. 1).

Of course, the goal of the inference is to assign a RTV to every $\#m_i$ such that for every i , the inferred RTV is high enough to conclude $PO2_i$. When the specification of m_i is changed, the previously inferred RTV for $\#m_i$ might no longer be high enough (for instance, because the specification of m_i now contains a method call). Therefore, the inference is rerun prior to re-verification. But as a consequence of modular development, it is not possible to re-infer every RTV. In particular, a RTV in a module that is hidden cannot be re-inferred. As a consequence, if an inferred RTV were publicly visible, a change to a specification that is hidden from a module M could indirectly invalidate the verification of M . That is, suppose that m_i and m_j are defined in different modules, and that the proof of $PO2_i$ depends on $\text{RTVal}(\#m_j) = n$. Suppose a part of the specification of m_j that is hidden from m_i is changed in such a way that re-inference of $\text{RTVal}(\#m_j)$ changes it to $n + 1$. Then the proof of $PO2_i$ no longer holds. While this does not go against modular development technically (re-inference of $\text{RTVal}(\#m_j)$ constitutes a change of public part of the specification of m_i), it is not intuitive (as the change is to an *implicit* part of the specification). Therefore, an inferred RTV is private, and an explicitly specified RTV is public. As the specifier has committed to the RTV, it is intuitive that changing it will require re-verification of modules to which it is visible. We discuss an algorithm to infer the RTVs for a module M in [8]. The outline is as follows. Construct a directed graph with a node N for every method visible in M , and with an edge from N to node N' iff N' occurs in the specification of N . For every N with an explicitly specified RTV i , label N with i . For every N with an RTV that is hidden from M , label N with maxRTV . For every remaining N , label N

with the lowest value such that (1) N cannot reach a node with a higher RTV , and (2) if possible, such that N cannot reach a node with the same RTV . (1) is always possible, as $maxRTV$ can be assigned to all nodes. (2) can't be achieved for nodes that are part of a cycle, nor for nodes that can reach a $maxRTV$ node.

4.3 The `measuredBy` Clause

The third ordering allows for directly or mutually recursive method functions. We associate with pure method m_i , a *measuredBy clause* that specifies a tuple of numerical expressions $\langle E_1, \dots, E_n \rangle$. $MeasuredBy(\#m_i, E, \sigma)$ is defined as $\langle \llbracket E_1[E/\mathbf{p}_i] \rrbracket \sigma, \dots, \llbracket E_n[E/\mathbf{p}_i] \rrbracket \sigma \rangle$. For each such expression E_j , there is a proof obligation that $Pre_i \Rightarrow 0 \leq E_j$, which ensures that the ordering is well-founded. We restrict the free variables in these expressions to be the numerical formal parameters of m_i , but one can easily imagine allowing other variables, too, for example so that one can mention the *RootDistance* of a non-`this` object parameter. By default, the `measuredBy` clause is tuple $\langle 0 \rangle$.

The use of the `measuredBy` clause is illustrated by Fig. 6, where it allows the mutually recursive methods `isEven` and `isOdd`. For the call to `this.isOdd(n-1)` in the specification of `isEven`, the reasoning is as follows. Consider an arbitrary $\sigma \in \Sigma$. Assume $r_0, r_1, r_2, t_0, t_1, t_2 \in \mathbb{Z}$ such that $Order(\#isEven, \mathbf{this}, \mathbf{n}, \sigma) = \langle r_0, r_1, r_2 \rangle$, and $Order(\#isOdd, \mathbf{this}, \mathbf{n} - 1, \sigma) = \langle t_0, t_1, t_2 \rangle$. Then $r_0 = t_0$, as both are determined by the *RootDistance* of the `this`-object (see Sect. 4.1). Also, $r_1 = t_1$ as the same RTV is assigned to mutually recursive method functions (see Sect. 4.2). Finally, $r_2 > t_2$ as $r_2 = \llbracket 2n \rrbracket \sigma$, and $t_2 = \llbracket (2m + 1)[n - 1/m] \rrbracket \sigma = \llbracket 2n - 1 \rrbracket \sigma$. Thus, $\langle t_0, t_1, t_2 \rangle$ is ordered lexicographically below $\langle r_0, r_1, r_2 \rangle$. So, if C is the class that declares `isEven` and `isOdd`, then $\forall \sigma \in \Sigma \bullet \llbracket \forall \mathbf{this} : C, \mathbf{n} : \mathbf{int} \bullet \#isOdd(\mathbf{this}, \mathbf{n} - 1) \prec \#isEven(\mathbf{this}, \mathbf{n}) \rrbracket \sigma$. For the call to `isEven(m)` in the specification of `isOdd`, the reasoning is similar (the essential observation being that $2m + 1 > (2n)[m/n]$). Together, these properties establish that *PO2* holds.

5 Related Work and Experience

Frame properties for a model field f declared in a class C (see Sect. 1) are discussed in [7]. Essentially, the idea is to add a specification-only field f to C , and to extend the deduction system with a second axiom $\forall \sigma \in \Sigma \bullet \llbracket \forall \mathbf{this} : C \bullet P \Rightarrow \mathbf{this}.f = \#f(\mathbf{this}) \rrbracket \sigma$, where P (defined by the methodology) describes the conditions under which the relation should hold. The methodology ensures that that $\#f(\mathbf{this})$ is assigned to f whenever P becomes *true*. Breunese and Poll suggest desugaring a model field using its satisfies clause [17]. This simplifies the treatment of model fields considerably, but does not account for recursion or for visibility constraints on satisfies clauses.

Modeling partial functions by underspecified total functions in the underlying logic can lead to unintuitive outcomes for the users of the specification language [18]. Recent work by Rudich *et al.* [19] discusses how to prevent such outcomes. The work also discusses how to allow conditional use of the axioms introduced by

pure methods, as well as class invariants, when establishing $PO1$ (see Sect. 2). Essentially, the idea is that if $Smaller_{i,j,k}$ holds, then the axiom introduced by m_j , instantiated for $Call_{i,j,k}$, can be assumed when proving $PO1_i$ (see Definitions 2 and 4). More formally, let $P_{i,j,k} \equiv (Smaller_{i,j,k} ? Spec_j[Call_{i,j,k}/\mathbf{result}] : \mathbf{true})$. Then $PO1_i$ can be weakened to $\forall \sigma \in \Sigma \bullet \llbracket \forall \mathbf{this}: C_i, \mathbf{p}_i: T'_i \bullet P \Rightarrow \exists \mathbf{result}: T_i \bullet Spec_i \rrbracket \sigma$, where P consists of a conjunct $P_{i,j,k}$ for every $i, j \in [0, N]$, for every $k \in [0, NrOfCalls_{i,j} - 1]$.

In Sect. 4.1, we discussed how our approach allows a number of the calls in the specification of the `myPersonnel` method in Fig. 8. The call to `result.has()` in that specification, however, is problematic. The axiom introduced by the pure method describes a property that holds in every well-formed program state. Therefore, the resulting list of `Nodes` has to exist in each such state (and contain the right `Persons`). This is reflected in $PO1$, which cannot be proven to hold for this example. Possible solutions to this problem are suggested in [4,20,21].

The heuristic guesses of candidate witnesses and the accompanying semantics checks in this paper have been implemented in the Spec# programming system; there is a partial implementation of `RootDistance` and the RTV scheme [3]. Pure methods occur frequently in practice, partly because Spec# by default treats property getters as pure methods. The Spec#/Boogie test suite alone requires 148 consistency checks. From more than a year's use, we find that, with one exception, the heuristics adequately guess candidate witnesses that (for consistently specified pure methods) the semantic checks quickly verify to ensure consistency.

The one exception to this positive experience has been pure methods with a non-null return type. The only non-null candidate witnesses that our heuristics guess are fields or parameters of exactly those types—the heuristics cannot use calls to constructors, as this would require one to first prove the consistency of the specifications of such constructors. Luckily, this case has occurred only for property getters whose body returns a newly allocated object (see [22] for a technique that allows such methods to be considered observationally pure). In the cases we have found, these property getters were not used as pure methods, so we could circumvent the problem by explicitly marking them non-pure.

6 Conclusions

Pure methods and model fields are useful and common specification constructs that can be interpreted by the introduction of axioms in the underlying proof system. Care has to be taken that these axioms do not introduce an inconsistency into the proof system. In this paper, we described and proved sound an approach that ensures no inconsistencies are introduced, and we described heuristics for the part of the approach that is problematic for trigger-based SMT solvers.

References

1. Cok, D.R.: Reasoning with specifications containing method calls and model fields. *Journal of Object Technology* 4(8), 77–103 (2005) (FTJJP 2004 Special Issue)

2. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract. *Softw. Pract. Exper.* 35(6), 583–599 (2005)
3. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Dwyer, M.B., Lopes, A. (eds.) *FASE 2007*. LNCS, vol. 4422, pp. 336–351. Springer, Heidelberg (2007)
4. Darvas, Á., Müller, P.: Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology* 5(5), 59–85 (2006) (FTfJP 2005 Special Issue)
5. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. *J. ACM* 52(3), 365–473 (2005)
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Leino, K.R.M., Müller, P.: A verification methodology for model fields. In: Sestoft, P. (ed.) *ESOP 2006*. LNCS, vol. 3924, pp. 115–130. Springer, Heidelberg (2006)
8. Leino, K.R.M., Middelkoop, R.: Proving consistency of pure methods and model fields. Technical report, Microsoft Research (2009)
9. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales (2001)
10. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
11. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. *Journal of Object Technology* 3(6), 27–56 (2004) (FTfJP 2003 Special Issue)
12. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) *ECOOP 2004*. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
13. Middelkoop, R., Huizing, C., Kuiper, R., Luit, E.J.: Specification and Verification of Invariants by Exploiting Layers in OO Designs. *Fundamenta Informaticae* 85(1–4), 377–398 (2008) (CS&P 2007 Special Issue)
14. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. *Science of Computer Programming* 62(3), 253–286 (2006)
15. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: *OOPSLA 1998*, pp. 48–64. ACM Press, New York (1998)
16. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: *OOPSLA*, pp. 302–312. ACM, New York (2003)
17. Breunese, C.B., Poll, E.: Verifying JML specifications with model fields. In: *FTfJP 2003*, Technical Report 408, ETH Zurich, 51–60 (2003)
18. Chalin, P.: Are the logical foundations of verifying compiler prototypes matching user expectations? *Form. Asp. Comput.* 19(2), 139–158 (2007)
19. Rudich, A., Darvas, Á., Müller, P.: Checking well-formedness of pure-method specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) *FM 2008*. LNCS, vol. 5014, pp. 68–83. Springer, Heidelberg (2008)
20. Naumann, D.A.: Observational purity and encapsulation. *Theor. Comput. Sci.* 376(3), 205–224 (2007)
21. Barnett, M., Naumann, D.A., Schulte, W., Sun, Q.: 99.44% pure: Useful abstractions in specifications. In: *FTfJP 2004*, Technical Report NIII-R0426, University of Nijmegen, 11–18 (2004)
22. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) *ESOP 2008*. LNCS, vol. 4960, pp. 307–321. Springer, Heidelberg (2008)

On the Implementation of @pre

Piotr Kosiuczenko

Institute of Information Systems, WAT, Warsaw

Abstract. The paradigm of design by contract provides a transparent way of specifying object-oriented systems. There exist a number of languages for contractual specification including OCL, JML and Spec#. Nevertheless, there are still a number of research problems concerning this approach. One of them is the implementation of primitive @pre in OCL or equivalently old in JML. Those primitives are used in post-conditions to refer to attribute values in states preceding an operation execution. There are a number of implementations of this operators, but all suffer logical and computational problems. In this paper, existing approaches to the implementation of @pre are discussed and a new solution is proposed.

1 Introduction

The main goal of software engineering is the construction of correct software. There exist various methods for ensuring that software meets requirements ranging from testing to automatic theorem proving. They have been developed in the late 1960s, most prominently by Floyd [6] and Hoare [9]. In the following years several approaches to system specification have been developed. Contracts are the prevailing way of specifying object-oriented systems from the user point of view (see [13]). A contract consists of three basic constraint types: invariants, pre- and post-conditions. The system consistency is ensured by invariants. A pre-condition specifies in which states a method can be called. A post-condition specifies the system state after the method execution.

Eiffel [14] is an object-oriented language including in a pioneering way contractual specification. It turns contracts into run-time checks of program correctness. The implementation of post-conditions is not an easy thing since it is necessary to compare attribute values in method's pre- and post-state. Of course copying the whole pre-state before a method is executed is out of question due to time- and memory-cost. Moreover if a method is called several times in a recursive manner, then it is necessary to copy the whole state again and again. Eiffel avoids this problem by saving before a method execution values of those attributes whose old values are referred to in the post-condition. However as we explain below, this approach has serious drawbacks in case of high level specification languages such as Object Constraint Language (OCL, see [15]).

Old attribute's values are accessed with the help of @pre in case of OCL and old in case of JML and Spec#. These primitives can be used in post-conditions and allow one for the comparison of attribute values in states before and after

operation execution; e.g. `salary = salary@pre + amount`. In case of OCL, one can use @pre with an attribute, an association-end and a query (OCL names the first two ‘property’). Although the idea is simple, those primitives are not easy to implement.

The Eiffel approach requires restrictions of post-conditions’ syntax. It is necessary to restrict post-conditions to formulas of the form: $t_0[t_1@pre/x_1, \dots, t_n@pre/x_n]$ (we use here the OCL notation), where term t_0 does not include @pre and where $t_i@pre$ is obtained from t_i by replacing every attribute, association-end and query a by $a@pre$, for $i = 1, \dots, n$; for example, term $(self.a.b)@pre$ is an abbreviation for OCL term $self.a@pre.b@pre$. $[t_1@pre/x_1, \dots, t_n@pre/x_n]$ denotes simultaneous substitution of terms t_i for variables x_i , for $i = 1, \dots, n$. Values of terms t_i are computed before the underlying method is executed, saved and then used after the method execution to compute the value of the post-condition. It should be noted that term $self.a@pre.b@pre.c.d->size()=self.a->size()$ is of the above mentioned kind, since it can be presented in the form: $(x.c.d->size()=self.a->size())[(self.a.b)@pre/x]$. However, it is not possible to present terms such as $self.a.b@pre$ in this form. The value of $self.a$ is not known in the pre-state and consequently cannot be pre-computed. It is possible to formulate in OCL more natural constraints of the latter kind; e.g. one can specify an operation which creates a new department of a company and makes sure that for all employees assigned to this department their salaries are increased.

It is disputable how severe the above mentioned syntax restriction is. It is true that most post-conditions can be written in that form. However, there are more serious problems with this approach. For example there are subtle problems with quantification (cf. [1]). Even more serious is the problem of cloning and more generally copying of large parts of the state. Cloning disallows dealing directly with object identity. Deep clones are problematic in the presence of circular references. There are also problems in case when more then one object references the same object; in this case one needs to avoid multiple clones of the referenced object. If objects are cloned then reference identity cannot be used for comparison.

Computing all potentially needed values in the pre-state could be very time- and space-consuming. In the case of `if ex then ex1 else ex2 endif`, we have to compute value of an expression e in the case when $e@pre$ occurs in $ex1$ or $ex2$. For example, let us consider expression `if 1 + 1 = 2 then 1 else q@pre endif` where q is a computationally complex, integer-valued query. Obviously, in general there is no need to evaluate q in the pre-state to compute the value of the whole expression. Nevertheless when the Eiffel approach is used, it is necessary to evaluate q in the pre-state. This unnecessarily increases the time and space complexity. Moreover if q does not terminate, then the evaluation of the whole expression does not terminate. Thus evaluation of constraints may not only slow down program execution, but also prohibit termination.

The problems with the Eiffel approach to @pre-values can be classified as follows:

- the support only for the restricted form of constraints
- the need of extensive cloning

- the lack of transparency in respect to object identity
- a potential increase of computational complexity

In this paper we present an algorithm addressing those problems. This algorithm is implemented in AspectJ. AspectJ is the most popular aspect-oriented language. (We refer the interested reader to [11] for a good introduction to aspect-oriented programming). The algorithm can be implemented in other aspect-oriented languages and also using reflective features present in languages such as Java and C#.

This paper is organized as follows. In section 2, we comment on the related work. In section 3, we present an example and use it to describe problems with existing approaches and to explain our algorithm. In section 4, we present the algorithm for computing @pre-values; we show how to deal with collection types and inheritance; we argue that this algorithm does not increase the time-complexity of constrained methods and that checking old values has constant time-complexity. Section 5 concludes this paper.

2 Related Work

There exist various languages for contractual system specification. In the realm of Java, there exist Java Modeling Language (JML, see [3,12]). It allows one for a modular system specification based on a type system grouping objects in master-slave hierarchies, the master being fully in control of its slaves. There is the concept of visible states being the moments when a public or a non-helper method is called or terminates. In the realm of .Net, and in particular C#, there is Spec# [1]. This language is closely related to JML, the main dissimilarity being a different approach to modularity and the relativization of visibility notion to objects' states. An efficient implementation was the primary goal of those specification languages instead of expressivity. Moreover, both specification languages are intimately related to the corresponding programming languages and therefore they are rather low level.

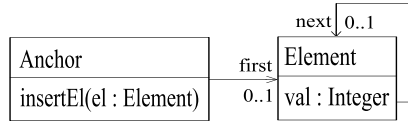
There exist high level, programming independent languages for contractual system specification, the most prominent being OCL [15]. It focuses on expressivity rather than an efficient implementation. There exist numerous tools for monitoring the satisfaction of OCL constraints. We mention here only Dresden OCL Toolkit (DOT) [4], jContractor [10] and OCL2J [5,2] (see [17] for an overview of other tools). Those tools implement OCL partially due to the complexity of the language. Due to high abstraction level of this language, contract monitoring often causes a significant slowdown in program execution.

All above mentioned languages follow the Eiffel approach. Interestingly also the current research focuses on the implementation problems concerned with this approach (cf. e.g. [5,2]). Archiving attributes before method execution is simple in the case of basic OCL types and classes. In the first case a single value is stored in a variable before method execution. In the case of terms of a class type, only an object reference is stored; the object itself is not cloned. If a term τ_i (see section 1) is of a collection type, then the situation is more complicated. A collection

is determined by elements it contains. Thus a clone of a collection has to contain every element of the original collection. As pointed out in [5], using @pre on a collection requires in most cases a duplication of the collection. In some cases, if operations like `size()` are used on the collection, then only the corresponding value has to be stored. If a post-condition relates attribute's values before and after operation execution and those values are of a collection type, then the collection before the operation execution has to be cloned. As pointed out in [5,2], cloning of collections is the major slow down factor in the automatic constraint evaluation and the authors classify this problem as a research challenge.

Another sort of problems emerges when objects are cloned in the pre-state, as it is done in OCL2J. As pointed out in [2], in this approach objects are cloned before being returned by a method, as required by the so called 'strong encapsulation principle'. In this case, one has to deal with user defined objects equality instead of identity (being identity of references or addresses). Some programming styles encourage exclusive use of user defined equality instead of identity; however a specification language should be implementation style independent. There are a number of logical problems when in programs identity comparison, denoted in Java by `==`, have to be replaced by user defined equality relation based on values of object attributes. Redirecting of links is needed when one wants to deal with true object identity and not a kind of equality based on attributes identification (cf. [2]).

As in the case of OCL2J, jContractor [10] allows post-conditions to refer to the state of an object at method entry, however the method is a bit different. A class includes an instance variable named OLD of the same type as the class. Post-conditions access old values of properties by referencing OLD. During compilation, resulting byte-code is instrumented in such a way that the reference to OLD is routed to a clone of the object created at method entry using Java method `clone()`. When a method is executed and its post-condition includes @pre, a clone of the object is created and stored in OLD [10]. As pointed out by the authors, there is a problem with object cloning if the method calls are nested, since in such a case the older clone is overwritten by a newer one. Therefore jContractor adopts a stack based approach: when execution enters a method that needs to save OLD, an object clone is created and pushed onto a stack; when the method terminates, the object is removed from the stack and used to check the post-condition. Pushing an object onto and removing it from the stack when the method is called or terminates does not differ from defining local variables which are then pushed onto and removed from the program stack. In fact, it is equivalent to wrapping a constrained method in another one which checks its pre-condition, saves old values, executes the original method and checks its post-condition. However in the case of post-conditions requiring navigation via several objects, as for example the OCL constraint presented in the introduction, deep cloning is needed. It should be noted that there is also the so called Memo pattern aimed at storing and handling past object states [8]. It should also be noted that there exist various methods for implementing OCL constraints in aspect-oriented languages, in particular AspectJ (cf. e.g. [18,16]).

**Fig. 1.** List

3 Example

In this section we present an example in order to describe the problems and explain the proposed solution. The class diagram in Fig. 1 models a list composed of an anchor object of class `Anchor` and a number of elements instantiating class `Element`. `Anchor` objects have attribute `first` storing the first element of the list. There is also method `insertEl(Element el)` for inserting elements into the list. `Element` objects have attribute `val` storing integers and attribute `next` pointing to the next element. Insertion of an element into a list may be specified by a formula saying that the set of list elements is enlarged by the new element.

```

context Anchor::insertEl(el : Element) post insertPC :
self.elements = self.elements@pre->union(Set{el})
  
```

where

```

context Anchor def :
elements : Set(Element) = if self.first = null then Set{}
else self.first.successorsOf(Set{self.first}) endif
context Element def :
successorsOf(Acc : Set(Element)) : Set(Element) =
if self.next = null or Acc->includes(self.next) then Acc
else self.next.successorsOf(Acc->union(Set{self.next})) endif
  
```

One can use in this post-condition including, but this would not help us to avoid cloning. Eiffel approach requires copying of the list before operation execution and then comparison of the new form of the list with the copied one. Using datatypes such as lists would require deep cloning causing problems listed in the introduction.

We implement `insertEl` in a very inefficient way. An element `el` is inserted into the list if the list is empty or `el.val` is smaller than the value of the first element. If not, then the list elements are removed as long as the first element does not exist or its value is larger than the value of the inserted element, then `el` is inserted and afterwards all removed elements are inserted too. Execution of this method causes a cascade of recursive calls. If we insert into an empty list $n+1$ elements with increasing values, then we need to make n copies of the list; i.e. $1+2+\dots+n = n(n+1)/2$ copies of object references.

We present implementation of classes `Anchor` and `Element`. It should be noted that during execution of `insertEl` several other calls to this method can be made if the inserted element is not smaller than the first element saved in the list. In particular the call stack can contain several calls to this method. On the

other hand, during execution of this method some calls may be terminated. This shows that a proper stamping policy is needed to guarantee that the method calls are numbered uniquely and that one needs a proper logic to figure out which snapshots are valid.

```
public class Anchor {
    public Element first;
    public void insertEl(Element el) {
        if(first == null) {
            first = el; el.next = null;
        }
        else if(el.val <= first.val)
            el.next = first; first = el;
        else {
            Element oldFirst = first;
            first = first.next; oldFirst.next = null;
            insertEl(el); insertEl(oldFirst);
        }
    }
}

public class Element {
    public Integer val = 0; public Element next = null;
    public Element(Element n, Integer v) {
        next = n; val = v;
    }
}
```

4 Implementation of @pre

In this section we present an implementation of the primitive @pre in AspectJ. We explain how to deal with collection types other than lists (i.e. vectors and arrays), queries and inheritance. We present a complexity estimation of the proposed algorithm. This algorithm has the advantage that we neither need to restrict syntax of post-conditions nor redirect references. In our approach we avoid the problems of deep cloning. However we need to reassemble the states of objects. Therefore we have to treat cloned object parts carefully and we need a logic for reassembling objects and for navigating via archived and not archived properties.

4.1 Implementation in AspectJ

Our algorithm is implemented in AspectJ. This language allows us to instrument classes with additional attributes and methods, to add so called pointcuts, for registering relevant events, and advices, for handling those events. As it is common in distributed systems, we use a kind of time-stamp to be able to reassemble objects states which existed at different times, but in our case those stamps refer to method calls.

Instrumenting classes looks as follows. If a property (i.e. attribute or association-end) a of type T occurs in a post-condition in the form $a@pre$, then we instrument the class corresponding to a by superimposing a history attribute $aHIST$ of type $Stack<SnapshotVal<T>>$ to store the history of a , a $pointcut$ which listens to changes of a and an advice which is responsible for storing values of this attribute. For every class possessing such attributes we add a new aspect. The value of $a@pre$ is returned by a superimposed method $aATpre()$. It should be noted that attributes which do not occur in post-conditions in the above mentioned form do not need to be archived. Objects of parametric class $Stack<SnapshotVal<T>>$ store histories of attributes in the form of a stack. Class $SnapshotVal<T>$ defines attribute snapshots, i.e. a temporary value of an attribute plus the corresponding time-stamp, or as we call it later meter-reading. Class $Meter$ stores information about the number of relevant method calls on the program stack and the lists of objects modified during execution of those methods; a method is relevant if it has a post-condition with attributes which require archiving. $MeterAspect$ handles calls to relevant methods. Class $Archive$ includes the core logic for value archiving and implements method $aATpre()$.

We describe our algorithm more precisely using the list example. In this example there are two attributes which must be archived: `first` and `next`. They do not occur in the post-condition directly but are used to define set elements storing list elements. It should be noted that values of attribute `val` are not archived, since `val@pre` does not occur in the post-condition.

Class $Meter$ handles time-stamps. They correspond to the number of calls of `insertEl` which are still on the program stack. This number is stored in attribute `meterValue`. $Meter$ handles also lists of objects modified during those method calls. Those lists are stored as attributes of objects of class $SnapshotModified$. Those objects are stored in stack `meter`. The goal is to minimize the memory use. After a method termination the corresponding list of modified objects is scanned and outdated snapshots are removed.

```
public abstract class Meter {
    static int meterValue = 0;
    static final Stack<SnapshotModified> meter =
        new Stack<SnapshotModified>();
    public static int getReading() {
        if(meter.size() == 0) return 0;
        return meter.top().meterReading;
    }
    public static void increaseMeter() {
        ++meterValue;
        SnapshotModified n = new SnapshotModified(meterValue);
        meter.push(n);
    }
    public static void decreaseMeter() {
        meter.pop();
    }
}
```

Note that part of Meter functionality can also be implemented by an aspect with aspect-object creation per control-flow; in that case meter attribute of class Stack<SnapshotModified> could be replaced by an attribute of snapshot class.

MeterAspect is an aspect implementing the logic concerning calls of methods with a post-condition, in this case insertEl. When the method is called, the meter is increased, a new object of class SnapshotModified storing lists of modified objects is created, stamped with the current meter-reading and pushed onto the meter-stack by executing increaseMeter(). After insertEl terminates, if the stack is not empty, then the top-most object storing lists of modified object is removed from the stack and objects stored in the lists are checked for inclusion of outdated snapshots. If the stack is empty, then all history attributes are emptied and meterValue is set to 0, since there is no active call of a relevant method on the program stack. Then the lists of modified objects are emptied. Note that pointcut ins1 catches only calls to method insertEl; consequently all other method calls do not have influence on the meterValue and are in a sense irrelevant. It should also be noted that aspect MeterAspect has the highest priority. This is necessary since during a relevant method call advices for archiving and retrieving attribute values must be executed after increasing meterValue and adding the new modified snapshot, and before meterValue is decreased and the modified stack is popped.

```
public aspect MeterAspect {
    declare precedence: MeterAspect, *;
    public pointcut ins1() : call(void insertEl(Element));
    before() : ins1() {
        Meter.increaseMeter();
    }
    after() : ins1() {
        SnapshotModified sm = Meter.meter.top();
        Meter.decreaseMeter();
        if(Meter.meter.size() > 0) {
            for(Anchor o : sm.modifiedFirst) o.firstHIST.clean();
            for(Element o : sm.modifiedNext) o.nextHIST.clean();
        }
        else /* Meter.meter.size() == 0 */ {
            for(Anchor an : sm.modifiedFirst) an.firstHIST.empty();
            for(Element e : sm.modifiedNext) e.nextHIST.empty();
            sm.modifiedFirst.removeAllElements();
            sm.modifiedNext.removeAllElements();
            Meter.meterValue = 0;
        }
    }
}
```

Class Archive stores the logic for handling old attribute values. This class contains generic methods for the storing and the retrieval of attribute values. getValueATpre returns the old value of an attribute. It contains two parameters: val corresponding to the current value of an underlying attribute and st

corresponding to the attribute's history-stack. If `st` is empty, then the attribute was not modified and `val` is returned. If not, then outdated snapshots are removed from the attribute's history-stack. If the topmost element in the stack has stamp smaller than the current meter-reading, then it means that during the recent method call the value was not modified and `val` is returned. If not, then the attribute was modified during this or a later, but already terminated, method call and the stored topmost value is returned. Method `getLastUpdateTime` returns last update time of an attribute. This value is taken from the topmost snapshot in the attribute's history-stack, if it is not empty; in the other case 0 is returned. `doArchiving` is meant for storing attribute's snapshots on an underlying attribute's history stack. This method has four parameters: `st` corresponds to the history stack of the underlying attribute, `modTop` corresponds to the topmost list of objects for which the underlying attribute was modified, `modBottom` corresponds to the bottommost list, `target` corresponds to the object for which the relevant attribute is above being modified, `cur` corresponds the current attribute's value. If stack `st` is empty, then a new attribute snapshot is created with the current meter-reading and put onto the stack. At the same time the target object is saved in `modBottom`. The object is saved at the bottom, since it is its first modification and therefore the saved value is the @pre-value for all previous method calls. If `st` is not empty, then it is checked if the attribute's value was already saved. If it was not saved, then a new attribute snapshot with the current meter-reading is created and pushed on stack `st`; at the same time the target object is saved in `modTop` which is the top list of objects for which the underlying attribute was modified. If it was saved, then only the meter-reading in the corresponding snapshot is updated.

```
public class Archive {
    static <T> T getValueATpre(Stack<SnapshotVal<T>> st, T val) {
        if(st.size() == 0) return val;
        st.clean();
        if(st.topReading() < Meter.getReading()) return val;
        else return st.top().value;
    }
    static <T> Integer getLastUpdateTime(Stack<SnapshotVal<T>> st) {
        if(st.size() == 0) return 0;
        st.clean();
        return st.top().meterReading;
    }
    static <T, S> void doArchiving(Stack<SnapshotVal<S>> st,
        Vector<T> modTop, Vector<T> modBottom, T target, S cur) {
        if(st.size() == 0) {
            st.push(new SnapshotVal<S>(cur, Meter.getReading()));
            modBottom.add(target);
        }
        else if(Meter.getReading() > st.top().meterReading) {
            if(st.top().value != cur) {
                st.push(new SnapshotVal<S>(cur, Meter.getReading()));
                modTop.add(target);
            }
            else st.top().meterReading = Meter.getReading();
        }
    }
}
```

```

    }
}
}

```

It should be noted that in the case when the archived value coincides with the attribute value before it is set for the first time during a method execution we need to update the meter-reading of the actual snapshot. This is because the attribute can be changed several times during the method execution. If the meter-reading was not updated before the first modification, then during the next modification the stored value would be treated as out of date and a value different from the stored, and in fact correct one, would be saved as the value in the pre-state.

Abstract class `Snapshot` has attribute `meterReading` for saving the current meter-reading of a snapshot, i.e. the number of relevant method calls on the program stack.

```

public abstract class Snapshot {
    int meterReading;
}

```

`SnapshotModified` extends `Snapshot` and is meant for storing the lists of objects for which attributes requiring archiving were modified during a method execution. In our case, these are attributes `first` and `next`. Inherited attribute `meterReading` stores the current meter-reading.

```

public class SnapshotModified extends Snapshot {
    public Vector<Anchor> modifiedFirst = new Vector<Anchor>();
    public Vector<Element> modifiedNext = new Vector<Element>();
    public SnapshotModified(Integer i) {
        meterReading = i;
    }
}

```

`SnapshotVal` is a parametric class extending `Snapshot`; its objects are used to store attributes' snapshots. Attribute `value` stores the attribute's value.

```

public class SnapshotVal<T> extends Snapshot {
    T value;
    public SnapshotVal(T el, Integer t){
        value = el;
        meterReading = t;
    }
    //... other SnapshotVal constructors
}

```

Parametric class `Stack` implements a stack with methods for popping and pushing snapshots. We implement this class using `Vector` but it can be implemented using class `Stack` from the Java API standard library. Apart of above mentioned methods, it contains method `subTop` for returning the element below

the topmost position. Method `clean()` is used for removing outdated snapshots; i.e. snapshots located at the top of the stack whose meter-reading is larger than the current meter reading and who have a direct follower, returned by `subTop()`, with meter-reading larger than or equal to the current meter reading. It should be noted that the cleaning must be done in a while loop, since in general it is not enough to remove only the topmost outdated snapshot. `smallbottom()` returns the bottommost element and `empty()` removes all elements from the stack.

```
public class Stack<S extends Snapshot> {
    Vector<S> stack = new Vector<S>();
    public void push(S el) {
        stack.add(el);
    }
    public S top() {
        return stack.lastElement();
    }
    public int topReading() {
        if(stack.isEmpty()) return 0;
        return top().meterReading;
    }
    public S pop() {
        S s = top();
        stack.remove(stack.size()-1);
        return s;
    }
    public S subTop() {
        return stack.get(stack.size()-2);
    }
    public int size() {
        return stack.size();
    }
    public void clean() {
        while(size() >= 2 &&
                subTop().meterReading >= Meter.getReading())
            stack.remove(stack.size()-1);
    }
    public S bottom() {
        return stack.firstElement();
    }
    public void empty() {
        stack.removeAllElements();
    }
}
```

For every class `C` with attributes requiring archiving we introduce aspect `ArchiveC` which superimposes corresponding history attributes and methods returning old values of attributes. This aspect also manages setting of attributes. In our example, for classes `Anchor` and `Element` we introduce aspects `ArchiveAnchor` and `ArchiveElement`. `first` is the only attribute of class `Anchor` which has to be

archived. For this attribute method `getFirstATpre()` is superimposed on `Anchor`. This method is implemented with the help of `getValueATpre` and `getLastUpdateTime`. Every manipulation of `first` is detected by pointcut `modFirst`. If the current meter-reading is larger than 0, meaning that there is a relevant method on the stack, then the archiving is performed by `doArchiving`. It should be pointed out, that the archiving is performed only if there is a relevant method on the programm stack, or equivalently the `meterValue` is larger than 0. This is due to the fact that if no method with a post-condition is executed, then there is no need for archiving the pre-state. The archiving is needed first when a relevant method, in this case `insertEl`, starts to execute.

```
public aspect ArchiveAnchor {
    public Stack<SnapshotVal<Element>> Anchor.firstHIST =
                                   new Stack<SnapshotVal<Element>>();
    Element Anchor.getFirstATpre() {
        return Archive.getValueATpre(firstHIST, first);
    }
    Integer Anchor.getFirstLastUpdateTime() {
        return Archive.getLastUpdateTime(firstHIST);
    }
    pointcut modFirst(Anchor target) :
        target(target) && set(Element Anchor.first);
    before(Anchor target) : modFirst(target) {
        if(Meter.getReading() > 0) {
            Archive.doArchiving(target.firstHIST,
                               Meter.meter.top().modifiedFirst,
                               Meter.meter.bottom().modifiedFirst,
                               target, target.first);
        }
    }
}
```

`ArchiveElement` is an aspect analogous to `ArchiveAnchor`. It instruments class `Element` by superimposing history attribute `nextHIST` and method `getNextATpre()` on class `Element`. It defines also pointcut `modNext` which detects changes of attribute `next` and the corresponding advice which does the archiving of old `next`-values.

```
public aspect ArchiveElement{
    public Stack<SnapshotVal<Element>> Element.nextHIST =
                                   new Stack<SnapshotVal<Element>>();
    Element Element.getNextATpre() {
        return Archive.getValueATpre(nextHIST, next);
    }
    Integer Element.getNextLastUpdateTime() {
        return Archive.getLastUpdateTime(nextHIST);
    }
    pointcut modNext(Element target) :
        target(target) && set(Element Element.next);
    before(Element target) : modNext(target) {
        if(Meter.getReading() > 0) {
            Archive.doArchiving(target.nextHIST,
```

```

        Meter.meter.top().modified Next,
        Meter.meter.bottom().modifiedNext,
        target, target.next);
    }
}
}

```

Post condition `insertPC` (see section 3) can be checked with an aspect of the following form. It should be noted that in contrast to aspect-oriented implementations of the Eiffel approach we do not need an around-advice for passing saved values; in such an advice first the values are saved then the underlying method is executed with `proceed` command and then the post-condition is checked (cf. e.g. [18,16]). We do not store any values before method execution. Therefore it suffices to use an after-advice. This makes the constraint implementation more elegant and efficient, since the around-advice causes a significant method slowdown.

```

public aspect ConstraintsAspect {
    public pointcut ins1(Anchor a, Element el) :
        target(a) && args(el) && call(void insertEl(Element));
    after(Anchor a, Element el) : ins1(a, el) {
        //... check the post-condition
    }
}

```

4.2 Collections, Queries and Inheritance

As mentioned in subsection 2 (see also [5]), in the case of collections, the Eiffel approach requires deep cloning. In subsection 4.1, we have shown how to deal with lists. We can deal in a similar way with arrays and vectors. The problem with those collections is that they cannot be directly instrumented by AspectJ, since it is not possible to superimpose attributes and methods on classes from the standard Java API library and other predefined types.

In case of arrays, there is no array class as such. In this case we need to replace arrays with classes in order to instrument them. An array of the form `C[]` can be replaced by class `ArrayC` with an attribute `array` of type `C[]` and history attribute `arrayHist` of type `Stack<SnapshotVal<C>>[]`. The old values of the array can be dealt with as in the case of object valued attributes; the difference is that we have to access certain positions in the array; i.e. we have to define method `getElementATpre(int i)`, which returns the correct value. The method can be defined as `aATpre()` the only difference is that it accesses values at position `i` in `array` and history stacks occurring at position `i` in `arrayHist`.

In case of vectors, we need to extend class `Vector` to be able to instrument it with AspectJ. Thus we can define class `VectorI<C>` which extends `Vector<C>` and has two additional attributes: `vHist` of type `Vector<Stack<SnapshotVal<C>>>` for storing elements' snapshots and `sizeHist` for storing old vector sizes, or more

precisely the history of `size()`. We need also to define new query methods for accessing values corresponding to the pre-state. We have implemented in a natural way methods `sizeATpre()`, `getATpre(int i)`, `lastElementATpre()` corresponding to methods `size()`, `get(int i)`, `lastElement()`, respectively, of class `Vector`. The implementation has been based on method `Archive.getValueATpre`, but takes into consideration the change of length. We skip it here due to the lack of space.

Dealing with queries is very simple in our approach. Every occurrence of `q@pre` can be replaced by query `qATpre`, where the body of `qATpre` is obtained from the body of query `q` by replacing every attribute `b` by `getBATpre()` and every invocation of a query `r` by the corresponding query `rATpre`.

The design by contract approach assumes that constraints are inherited by subclasses and that subclasses may be additionally constrained. If class `Anchor` was extended by class `AnchorB` and the method `insertEl` possessed in `AnchorB` an additional post-condition, then we would have to make sure that also the additional post-condition is checked. This can be achieved by defining another aspect for monitoring constraints which differs from `ConstraintsAspect` in that parameter `a` in pointcut `ins1(Anchor a, Element el)` is restricted to objects of class `AnchorB`, i.e. by defining an additional pointcut with signature `ins2(AnchorB a, Element el)`. In this case only methods executed on objects of class `AnchorB`, and of its subclasses, will be constrained by the additional post-condition corresponding to `ins2`.

Overloading attribute names differs from method inheritance. Attributes are bound at compilation time. Whereas methods are bound at execution time depending on the class of the actual implicit parameter. In our approach we introduce method `aATpre` for every attribute `a` which requires archiving; thus we have to make sure that different attributes correspond to different methods. This can be achieved either by renaming attributes or by defining different methods for defacto different attributes. In the second case, it is necessary to replace every occurrence of `a@pre` by the corresponding method. We can distinguish between defacto different attributes by checking the type of every attribute occurrence in a post-condition.

For example let us assume that class `AnchorB` extends class `Anchor` and that both classes define attribute `first`. We can either rename the attribute in class `AnchorB` to avoid name clash or define two different methods for accessing old values of those two different attributes. In the second case we can superimpose methods `getAnchorFirstATpre` and `getAnchorBFirstATpre` returning the old attributes' values as well as the corresponding history attributes `Anchor.firstHIST` and `AnchorB.firstHIST`. Observe that in the case of history attributes we do not need to use different names since they are superimposed on different classes. Finally for every subterm of the form `t.first` occurring in the post-condition we have to replace `first` by `getAnchorBFirstATpre` if `t` defines objects of class `AnchorB`, or by `getAnchorFirstATpre` in the other case.

4.3 Complexity

In this section we informally show that the algorithm presented in subsection 4.1 does not increase the time-complexity class of the constraint validation nor the time complexity class of constrained methods.

More precisely, let m be a method with post-condition `postCond` including the primitive `@pre`. The execution of book-keeping activities performed by the algorithm to archive old values of attributes does not increase the time-complexity class of m . Let post-condition `postCond'` be obtained from `postCond` by replacing every occurrence of an attribute `a@pre` by query `aATpre`. The evaluation of `postCond'` has the same time complexity as it would have when evaluation of `a@pre` required one time unit. This is due to the fact that the execution of `aATpre` apart of cleaning requires a constantly bound number of steps. Since the `@pre`-values are computed when needed, there is no problem with unnecessary pre-computation of term values, in particular with unnecessary execution of nonterminating queries (see the introduction).

Our method does not increase time complexity class of constrained methods since setting an attribute is accompanied by at most one snapshot archiving and removal and there is only a bound number of steps needed for those two operations. A call of a constrained method results in increasing of `meterValue`, creation of a new `ModifiedSnapshot` object and pushing it onto the stack. When the method terminates, the corresponding lists of modified objects are scanned, irrelevant snapshots are removed from the history stacks and the `ModifiedSnapshot` object is removed from the stack. The creation and removal of a `ModifiedSnapshot` object requires bound time; scanning of modified object lists and removal of outdated snapshots from an attribute's history can be accounted for when counting the steps associated with the corresponding attribute modification.

The space complexity may be increased and in the worst case equal to the time complexity of m . If in constraint $t_0[t_1@pre/x_1, \dots, t_n@pre/x_n]$ (see the introduction) term t_0 does not contain `if then else endif`-statements and if terms t_i are of basic OCL type or return single objects as values, then it is not necessary to archive attributes' values. It is enough to save values of t_i . In this and other cases it may be advantageous to use the Eiffel approach.

In some cases a combination of the Eiffel approach and the algorithm described in subsection 4.1 may be the optimal solution. Of course pre-computing values of some terms and then archiving all values of attributes occurring in those terms is not reasonable. However it makes sense in the case when the set of attributes which require archiving is disjoint from the set of attributes occurring in terms t_i .

5 Conclusion and Future Work

In this paper we discussed currently existing approaches to the implementation of the primitive `@pre` and pointed out that they are all based on the Eiffel approach to `old`-implementation. We listed the corresponding problems and proposed a new algorithm which avoids those problems. This algorithm does not require restriction of the post-condition syntax; no collection cloning is needed and the identity of objects is preserved. Moreover, a post-condition can be implemented

with an after-advice, instead of an around-advice, what makes the constraint implementation more elegant and efficient. We investigated also the complexity of this algorithm and showed that it does not increase time-complexity of method execution and constraint evaluation.

In the future we are going to implement this algorithm using Java reflectivity features. We are going to investigate its defacto time and space overhead, and to figure out when the Eiffel approach and our algorithm can be most efficiently combined. We are also going to provide a formal proof of algorithm's correctness.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Briand, L., Dzidek, W., Labiche, Y.: Using Aspect-Oriented Programming to Instrument OCL Contracts in Java, Tech. Rep. SCE-04-03, Carleton Univ. (2004)
3. Darvas, A., Müller, P.: Reasoning About Method Calls in JML Specifications. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP 2005), Glasgow, Scotland (July 2005)
4. DOT, Dresdener OCL Toolkit, <http://dresden-ocl.sourceforge.net/>
5. Dzidek, W., Briand, L., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 10–19. Springer, Heidelberg (2006)
6. Floyd, R.W.: Assigning meanings to programs, in Mathematical Aspects of Computer Science. In: Proceedings of Symposium in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967)
7. Hussmann, H., Finger, F., Wiebicke, R.: Using Previous Property Values in OCL Postconditions: An Implementation Perspective. In: Int. Workshop UML 2.0 - The Future of the UML Constraint Language OCL, York, UK, October (2000)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison- Wesley, Reading (1995)
9. Hoare, T.: An Axiomatic Basis for Computer Programming. CACM 12(10) (1969)
10. Karaorman, M., Abercrombie, P.: jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation. Formal Methods in System Design 27(3), 275–312 (2005)
11. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming, Manning (2003)
12. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J.: JML Reference Manual, Tech. Rep. 2007/02/07, Iowa State Univ. (2007)
13. Meyer, B.: Applying design by contract. Computer 25(10), 40–51 (1992)
14. Meyer, B.: Eiffel: The Language. Object- Oriented Series. Prentice Hall, New York (1992)
15. OMG, OCL 2.0 Specification, Version 2005-06-06 (June 2005)
16. Richters, M., Gogolla, M.: Aspect-Oriented Monitoring of UML and OCL Constraints. In: Proc. UML 2003 Workshop on Aspect-Oriented Software Development with UML. Illinois Institute of Technology, Department of Computer Science (2003)
17. Toval, A., Requena, V., Fernandez, J.: Emerging OCL Tools. Journal of Software and System Modelling 2(4), 248–261 (2003)
18. Van Der Straeten, R., Casanova, R.: Stirred but not Shaken: Applying Constraints in Object-Oriented Systems. In: Proc. of NetObjectDays, pp. 138–150 (2001)

Formal Specification and Analysis of Timing Properties in Software Systems

Musab AlTurki¹, Dinakar Dhurjati², Dachuan Yu², Ajay Chander²,
and Hiroshi Inamura²

¹ University of Illinois at Urbana-Champaign, Urbana IL 61801, USA
alturki@illinois.edu

² DOCOMO USA Labs, Palo Alto CA 94304, USA
{Dhurjati,Yu,Chander,Inamura}@docomolabs-usa.com

Abstract. Specifying and analyzing timing properties is a critical but error-prone aspect of developing many modern software systems. In this paper, we propose a new specification language and analysis framework for expressing and analyzing timing behaviors of complex software systems. Our framework has the following significant benefits: a) it is expressive, b) it supports trace analysis and simulation of timing behaviors, c) allows for verification of properties of specification, and d) checks for common usage errors of timing constructs. The language constructs for timing were chosen to be very flexible, suitable for expressing different kinds of timing behaviors, and are inspired from timing constructs used in previous languages like SDL. We define the formal semantics of our language using a real-time rewrite theory. Since real-time rewrite theories are executable in Real-Time Maude, our framework supports trace analysis and simulation of timing behavior for specifications. Furthermore, the timed model checker for Real-Time Maude can be readily used for analyzing and verifying various real-time properties of the specifications. Finally, to prevent misuses of timing constructs that can be made possible due to their flexibility, we develop abstract interpretation based static analysis tools that check for common usage errors. We believe that our framework, with the above benefits, provides a significant step forward in facilitating the use of formal tools for specification and analysis of timing behaviors in software development.

1 Introduction

Due to increasing complexity of modern software systems, the likelihood of making errors during the software design phase has increased exponentially. While some of these errors might be detected during the testing phase, it is much more cost effective to detect and correct these errors early during the design phase. For this reason, formal specification and analysis tools are increasingly being deployed to improve the quality of software design.

Many real-world software systems rely on components that have timing requirements to be met. These may represent maximal timing constraints, such as timeouts, minimal timing constraints, such as delays, or durational constraints,

which combine both maximal and minimal constraints. Consequently, correctness of such software systems depends not only on their functional requirements but also on the non-functional timing requirements. Therefore, to be able to formally reason about such requirements, methods and tools for the specification and analysis of real-time requirements need to be developed.

There have been several attempts at developing formal analysis and verification tools for timing properties in software specifications (see [1] and the references there) but there is a gap between the languages used by these tools and what the current specification languages provide, making it hard to integrate them into current design activities in the software development industry. Most of these tools are based on timed formalisms, such as timed automata [2] and timed Petri Nets [3], that typically sacrifice expressiveness for decidability. While they provide efficient formal analysis and verification tools, such timed formalisms are typically difficult to understand and use by the software specification writer, which further limits their applicability in industry. Furthermore, timing constructs in existing high-level specification languages are either restrictive (e.g. Erlang [4]) or flexible but at the cost of allowing many misuses while not providing effective mechanisms to detect them (e.g. SDL [5]).

In this paper, we propose a simple but powerful specification language for expressing timing properties together with an integrated analysis framework that makes available a suite of formal analysis tools for software designers. The language constructs for timing were chosen to be very flexible, suitable for expressing different kinds of timing behaviors, and are inspired from timing constructs used in previous languages like SDL. Due to this expressiveness, timing constructs used in other high level specification languages like SDL and UML can be easily translated into constructs of our specification language. We define the formal semantics of our language with rewrite rules in a real-time rewrite theory [6]. Since real-time rewrite theories are executable in Real-Time Maude [7] under few reasonable assumptions, our framework automatically supports trace analysis and simulation of timing behavior for specifications. Furthermore, the timed model checker for Real-Time Maude can be readily used for analyzing and verifying various real-time properties of the specifications. Thus the integrated analysis framework facilitates the use of formal specification tools by reducing the gap between the specification language and the language used by the verification tools. Finally, since the timing constructs are intended to be very flexible, there is a possibility of misusing the constructs. To prevent such misuse, we develop abstract interpretation based static analysis tools that check for common usage errors.

The main benefits of our framework can be summarized as follows: (1) It is an expressive framework that is capable of formally capturing software specifications given in various specification languages; (2) it supports trace analysis and simulation of timing behaviors; (3) it allows for verification of complex properties of specifications; and (4) it can automatically check for common usage errors of timing constructs. We believe that our framework, with the above

benefits, provides a significant step forward in facilitating the use of formal tools for specification and analysis of timing behaviors in software development.

The rest of the paper is organized as follows. In Section 2 we present our specification language, $\mathcal{L}$ and its semantics. This is followed in Section 3 with a description of a prototype implementation of the language using Real-Time Maude. In Section 4 we describe how the timing abstractions can be misused, and then in Section 5 we describe our abstract interpretation based solution to detect and prevent such misuses. In Section 6 we compare our approach to related work in the area. Finally, we conclude in Section 7 with a summary of our approach.

2 The Specification Language $\mathcal{L}$

In this section, we introduce a high-level specification language $\mathcal{L}$ that is well-suited for describing a spectrum of behaviors of various software systems, including their timed behaviors. $\mathcal{L}$ is a simple, concurrent specification language that is aimed to serve as a formal programming model for various user-level specification languages, such as SDL and UML. The language is intended to provide a unified, well-established specification framework for the analysis and verification of such higher-level specifications. Beside providing a core language with formal semantics for specification creation, management and analysis, the simplicity of $\mathcal{L}$ directly translates into a simple formal model that can easily be analyzed and manipulated.

While the language supports several imperative features for describing sequential computations, concurrency in $\mathcal{L}$ is modeled by asynchronously communicating processes that can be dynamically created or destroyed. A process maintains a thread of sequential computation representing a simple component in software. A process may create another process with a specified computational behavior, or may destroy itself. Processes communicate by exchanging messages asynchronously, and use timers as the basic timing abstraction to account for timing behaviors. The syntax and semantics of $\mathcal{L}$ are described next.

2.1 Syntax and Examples

The syntax of $\mathcal{L}$ is shown in Figure 1. A constant expression in $\mathcal{L}$ can be either an integer value, a boolean value, a literal string, or a variable name. Complex expressions can be constructed using standard arithmetic, relational, and boolean composition operators.

Unlike expressions which evaluate to some constant value, commands do not produce values, but are there for their side effects. A command in $\mathcal{L}$ can be an assignment statement, a scoped declaration of a variable using a **let** statement, a conditional statement, or a while loop statement. The language also has a few process-level commands, which include creating a new process, destroying the current process, sending a message to a process, and receiving a message. The body of the receive statement may consist of a list of exclusive case

$$\begin{aligned}
& x, y \in \text{Variable} \quad n \in \text{Integer} \quad r \in \text{String} \\
& e \in \text{Expression} ::= x \mid r \mid a \mid b \\
& a \in \text{Arithmetic Expression} ::= n \mid a \circ a \\
& b \in \text{Boolean Expression} ::= \text{true} \mid \text{false} \mid a \bullet_{\text{rel}} a \mid b \bullet_{\text{bool}} b \mid \neg b \\
& c \in \text{Command} ::= x := e \mid \text{let } x = e \text{ in } c \mid \text{if } b \text{ then } c \text{ else } c \mid \text{while } b \text{ c} \\
& \quad \mid \text{new } x = y \text{ in } c \mid \text{destroy} \\
& \quad \mid \text{send } e \text{ to } e \mid \text{receive } x \text{ in } \{l; \text{default} : c\} \\
& \quad \mid \text{set } x \text{ to } e \mid \text{release } x \mid \{\} \mid \{c\} \mid c; c \\
& l \in \text{CaseList} ::= \epsilon \mid \text{case } e : c; l \\
& m \in \text{Module} ::= \text{module } x \text{ is } c
\end{aligned}$$

Fig. 1. The abstract syntax of $\mathcal{L}$

statements followed by a default statement. Timers are managed using two constructs: **set**, for starting a timer, and **release**, for dropping a timer. The expiration of a timer in a process triggers a signal that can be checked by a **receive** command. Furthermore, commands can be grouped into command blocks, and sequenced using the semicolon as a sequencing operator. Finally, a specification in $\mathcal{L}$ may use an optional list of module declarations, serving as templates for new processes.

A variable x is bound in c in the commands **let** $x = e$ **in** c and **new** $x = y$ **in** c , and is bound in l and c of the **receive** command. Variables used in **set** and **release**, called *timer variables*, are globally scoped variables and are assumed to be distinct in a given specification for it to be meaningful. A variable is said to be free if it is not bound.

For compactness, we use **if** b **then** c as syntactic sugar for a conditional with an empty *else* branch, and **receive** x **in** c to denote a receive statement with no *case* branches, i.e. **receive** x **in** $\{\epsilon; \text{default} : c\}$. We shall also use **let** $x_1 = e_1, x_2 = e_2$ **in** c as a shorthand for two nested **let** commands.

Example. (CLIENT) The specification shown in Figure 2 defines a client process that interacts with the user and a server, and timeouts responses from the server, for which it maintains two timers t_1 and t_2 . Upon receiving a “resend” request from the user, the process forwards a request to the server and sets a timeout of 60 time units for the first response from the server using the timer variable t_1 . There are three possibilities at this point; (1) a timeout occurs, which is indicated by the incoming signal t_1 , and at which case the process restarts and waits for another request from the user; (2) another request from the user is received, at which case the client resets the timer t_1 , and sends a request to the server; (3) a response from the server is received, at which case t_1 is dropped and a similar process is initiated for subsequent responses from the server using another timer t_2 . For simplicity, the example does not specify how the client actually processes incoming responses from the server.

We will refer to the CLIENT example above in the rest of the paper to illustrate various aspects of the specification framework. Below, we give a more precise description of the semantics of $\mathcal{L}$.

```

module client is
  let  $a = \text{true}, b = \text{true}$  in
  receive  $o$  in                                — server id
  while ( $\text{true}$ )
    receive  $p$  in {                               — resend request from user
       $a := \text{true}; b := \text{true};$ 
      while ( $a$ ) {
        set  $t_1$  to 60;
        send “request” to  $o$ ;
        receive  $m$  in {
          case  $t_1$  :  $a := \text{false};$ 
          case “resend” : release  $t_1$ ;
          default : {
            release  $t_1$ ;
            while ( $b$ ) {
              set  $t_2$  to 120;
              receive  $m$  in {
                case  $t_2$  : {  $b := \text{false}; a := \text{false}$  };
                case “resend” : { release  $t_2$ ;  $b := \text{false}$  };
                default : { release  $t_2$  } } } } } } } } }

```

Fig. 2. A specification in $\mathcal{L}$ for a client process with timeouts

2.2 Formal Semantics

We give the formal semantics of $\mathcal{L}$ as an object-oriented real-time rewrite theory $\mathcal{R}_{\mathcal{L}}$. The semantics is distributed and concurrent in that a state for a specification in $\mathcal{L}$ consists of one or more process objects that are executed concurrently, and which may interact with each other as time elapses.

Real-Time Rewrite Theories. A rewrite theory, a unit of specification in rewriting logic [8], gives a formal description of a concurrent system including its static state structure and dynamic behavior. A rewrite theory is a tuple $\mathcal{R} = (\Sigma, E, R)$, with

- (Σ, E) a membership equational logic [9] theory with signature Σ and a set of universally quantified equations and/or memberships E . The signature Σ declares the sorts and operators to be used in the system specification, while equations and memberships E algebraically specify the properties satisfied by these operators.
- R a set of universally quantified, possibly conditional, rewrite rules specifying the computational behavior of the system. A rewrite rule has the form:

$$r : t_1 \longrightarrow t_2 \text{ if } C \quad (1)$$

where r is a label, t_1 and t_2 are terms over Σ , and C is a conjunction of equational or rewrite conditions. A rewrite rule gives a general pattern for a possible change in the state of a concurrent system (See [10] for a detailed account of generalized rewrite theories).

A real-time rewrite theory [6] extends a regular rewrite theory with support for modeling temporal behaviors of systems. In particular, in a real-time rewrite theory $\mathcal{R}^\tau = (\Sigma^\tau, E^\tau, R^\tau)$: (i) the equational theory (Σ^τ, E^τ) contains a sort for *Time* representing the time domain, which can be either dense or discrete, and declares a system-wide operator that encapsulates the whole system being modeled into a special sort *GlobalSystem* for managing time elapse, and (ii) the set of rewrite rules R^τ is the disjoint union of two sets R_I and R_T , where R_I consists of *instantaneous* rewrite rules having the form (1) above and representing instantaneous transitions in the system, and R_T consists of *tick* rewrite rules modeling system transitions that take non-zero amount of time to complete. A tick rewrite rule has the following form

$$r : \{t_1\} \xrightarrow{\tau} \{t_2\} \text{ if } C$$

where τ is a term of sort *Time* representing the duration of time required to complete the transition specified by the rule. The global operator $\{\_ \}$ encapsulates the whole system into the sort *GlobalSystem* to ensure the correct propagation of the effects of time elapse to every part of the system.

Semantic Infrastructure. We fix a sort V of values in $\mathcal{L}$. Lists of values can be constructed as fully associative lists of comma-separated values. An environment σ is a mapping from variable names to values, specified in $\mathcal{R}_{\mathcal{L}}$ as an associative list of entries of the form $[x, v]$ with identity *nil*.

A state in the system is represented by a *configuration* consisting of a multi-set of objects. The fundamental class of objects within a configuration is the *Process* class. In addition to the process object identifier, a process object contains the following fields: a name, an environment, a command, a field for the timer set of the process, and a queue of incoming messages:

$$\langle id : Process \mid name : x, env : \sigma, cmd : c, tmr : T, msg : M \rangle$$

The queue of messages M is simply a list of values, and T is a set of timer records of the form $\{x, v_t\}$, with v_t a time value. A timer record in T represents an *active* timer, which is a timer that has been started but is not yet expired or handled.

Instantaneous Transition Rules. In $\mathcal{R}_{\mathcal{L}}$, instantaneous transitions of $\mathcal{L}$ are modeled by regular rewrite rules, which specify the behavior of a process within a configuration based on the next command to be executed by that process. The command field *cmd* of a process serves as a continuation that defines what action to be taken next. For example, the rule labeled **set** below specifies the semantics for setting a timer:

$$\begin{aligned} [\mathbf{set}] : \langle id : Process \mid env : \sigma, cmd : \mathbf{set} \ x \ \mathbf{to} \ a ; c', tmr : T \rangle \\ \longrightarrow \langle id : Process \mid env : \sigma, cmd : c', tmr : \{x, a \downarrow_\sigma\}, T \rangle \end{aligned}$$

where $e \downarrow_\sigma$ denotes the evaluation of e using the environment σ , while expiration of a timer is captured by the **timeout** rule below:

$$\begin{aligned} [\text{timeout}] : \langle id : \text{Process} \mid env : \sigma, cmd : \text{receive } x \text{ in } C ; c', tmr : \{y, 0\}, T \rangle \\ \longrightarrow \langle id : \text{Process} \mid env : \sigma[x, y], cmd : \text{cases}(x, C); pop; c', tmr : T \rangle \end{aligned}$$

with $\text{cases}(x, C)$ and pop as auxiliary continuation items for processing the body of a receive statement. We note that the rules are given in the object-oriented specification style, in which attributes within a process object that do not play a role in the rule need not be mentioned. We assume that message exchanges are instantaneous (take no time to complete) and are therefore modeled by instantaneous rewrite rules.

2.3 Timed Semantics

Assuming R is a time value and C is a configuration, the *tick* rule in $\mathcal{L}$ that models time elapse and its effects is as follows:

$$[\text{tick}] : \{C\} \xrightarrow{R} \{\delta(C, R)\} \quad \text{if } R \leq mte(C) \wedge inactive(C)$$

There are several important observations to be made here:

- The function δ equationally propagates the effect of a time tick to all objects within the configuration C , which comprises decreasing all timer values within all process objects by the amount R of the tick.
- The function mte equationally defines the *maximum time elapse* until the next event of interest. This is a standard technique in RTM to specify upper bounds on how much a clock is allowed to advance before the next event in the configuration. In this case, the mte of a configuration of processes is determined by the timer with the minimum time value among all sets of timers in all processes:

$$mte(T, \{x, v_t\}) = \min(mte(T), v_t), \quad mte(\phi) = \infty$$

- The predicate *inactive* distinguishes states in which instantaneous (untimed) transitions are enabled (also called *active states*) from those in which the only possible transition is a tick transition advancing time (*inactive states*). The predicate is used to restrict applications of the tick rule to inactive states so that instantaneous transitions have precedence over time tick transitions. This is to maintain the expected semantics of timers and to prune uninteresting behaviors in which a configuration might appear to be progressing while it is not (for example, advancing time without doing anything else). This semantics enforces the fact that when a timer in a process expires, its signal cannot be ignored and must be handled, either by releasing the timer or by consuming its signal. For this semantics to be fully meaningful, however, configurations may only assume *non-Zeno* behaviors (which are behaviors in which time will always eventually have a chance to advance), which is a common assumption for real-time specifications with logical time.

3 Analysis of $\mathcal{L}$ Specifications in Real-Time Maude

Real-Time Maude (RTM) [7], which is based on Maude [11], provides a highly efficient implementation of real-time rewrite theories. We have developed a prototype in RTM for $\mathcal{L}$ that corresponds to the specification $\mathcal{R}_{\mathcal{L}}$ described above. As an immediate consequence of specifying the formal semantics of $\mathcal{L}$ in RTM, we obtain a simulator and several formal analysis tools essentially for free. Among the analysis tools provided are: (1) the timed fair rewrite `tfrew`, which simulates one possible behavior (a sequence of rewrite steps) of a specification up to a given time bound; (2) the `tsearch` command, which performs timed breadth-first search on the reachable state space from given an initial state, while looking for a state matching a given term and satisfying a given semantic condition; and (3) the timed model-checking command `mc T |=t F in time <= R`, which checks for satisfiability of the linear temporal logic (LTL) formula F along paths starting from the initial state T within the time bound R .

The prototype is specified as a *real-time object-oriented module* $\mathcal{M}_{\mathcal{L}}$ in RTM, which is declared using the syntax `tomod Name ... tomend`. To simplify analysis, we assume a discrete time domain, implemented using the domain of natural numbers extended with infinity, which can be specified by letting the module $\mathcal{M}_{\mathcal{L}}$ extend RTM's predefined module `NAT-TIME-DOMAIN-WITH-INF`.

We consider the `CLIENT` specifications given in Figure 2 above to illustrate the use of such formal tools. We use `client` to denote its specification in $\mathcal{M}_{\mathcal{L}}$. Since `client` is a template for a reactive process that communicates with a user and a backend server, we assume an initial configuration `system` in which a user object and a server object are defined in order to be able to perform analysis on `client`. The initial configuration contains a server process object that upon receiving a request sends out five responses, five time units apart, and a user process object that sends two “resend” requests, the first at time 1, and the other at time 20. To simplify the presentation of the analysis, another object, called the *Observer* object, is used to record traces of events of interest along with their time stamps.

3.1 Simulation and Prototyping

A sample run of `system` for a duration of 200 time units can be obtained by issuing the following command (where some of the output is omitted for brevity):

```
Maude> (tfrew
system in time <= 200 .) Result ClockedSystem : {...
  < oo : Obser | out : ([6 : "t1: first response received"
[11 : "t2: response received"] [16 : "t2: response received"]
[21 : "t2: resend before it expires"] [21 : "t1: first response received"]
[26 : "t2: response received"] [31 : "t2: response received"]
[36 : "t2: response received"] [41 : "t2: response received"]
[161 : "t2: expired"]) >
  < p(1): Process | name: 'client, cmd: receive p in ... , tmr: empty >
  < p(2): Process | name: 'server, cmd: receive m in ... , tmr: empty >
  < p(3): Process | name: 'user, cmd: {}, ... , tmr: empty > in time 200
```

The result above shows that after 200 clock ticks, the system reaches a quiescent state where no more message exchanges exist or are scheduled, and no timers are yet to be set or processed. As can be seen from the recorded trace, a “resend” request from the user was received at time 21 while the client was processing the third response from the server, immediately after which the client resent the request and restarted processing. Since the server sends only five responses to a given request, we see the timeout at time 161 after the fifth response had been received at time 41.

Furthermore, using timed search, one can verify, starting from `system`, the property that the system will in fact never be in a quiescent state before that.

```
Maude> (tsearch system =>+ { CF:Configuration } such that
      inactive({CF:Configuration}) and noAliveTimer(CF:Configuration)
      in time <= 160 .)
rewrites: 217595 in 720ms cpu (720ms real) (301842 rewrites/second)
No solution
```

The arrow `=>+` means states reachable by one or more rewrites from the given state. The semantic condition `inactive(CF)` and `noAliveTimer(CF)` captures exactly what it means for a state to be quiescent.

3.2 Model Checking Analysis

RTM also provides powerful time-bounded model-checking tools for verifying general LTL formulas, representing both liveness and safety properties, which can be immediately applied to specifications in $\mathcal{L}$. The LTL formulas are based on a set of atomic propositions that capture state properties of interest and a labeling function that assigns to each state in the system a subset of atomic propositions that are true in that state. Given a module `M` for some specification in $\mathcal{L}$, this is done in RTM by defining a module `M'` that imports the module `M` and the internal module `TIMED-MODEL-CHECKER` and specifies equationally the meanings of these propositions and the labeling function. For our running example, `client`, we would perform model checking against a module extension of the form:

```
(tomod MODEL-CHECK-CLIENT is
  including TIMED-MODEL-CHECKER .
  protecting CLIENT .
  ...
endtom)
```

where `including` and `protecting` represent module extension modes (see [11]). The internal module `TIMED-MODEL-CHECKER` declares sorts for states `State`, atomic propositions `Prop`, logical formulas `Formula` to which the various LTL operators belong, and the logical time-bounded satisfaction operator `|=t`, among several other things. Thus, within the module above, one can declare the following two propositions (the keywords `ops`, `var`, and `eq` introduce, respectively, operator declarations, variable declarations, and equations in Maude):

```

ops first-response timeout : -> Prop .
var CF : Configuration . var O1 O2 : Output . var R : Time .
eq {CF < oo : Obser | out : (O1 [R : "t1: first response received"]
    O2) > } |= first-response = true .
eq {CF < oo : Obser | out : (O1 [R : "t2: expired"] O2) > } |=
    timeout = true .

```

The first proposition **first-response** is true in a state in which the client has already received its first response from the server, while the other proposition **timeout** is true in a state where the second timer has expired. States in which a proposition does not hold need not be specified.

Using these proposition, we can verify a fairly complex property about the system modeled by **client**: it is always the case that within the first 200 time units and after receiving the first response from the server, the second timer will eventually expire. This property holds since the server will cease to send out responses after the first response, causing the client to eventually timeout. This can be checked automatically using the model-checking command:

```

Maude> (mc system |=t [] (first-response -> <> timeout) in time <= 200 .)
rewrites: 164943 in 689ms cpu (693ms real) (239084 rewrites/second)
Result Bool : true

```

where `[]` denotes “always”, `->` “implication”, and `<>` the “eventually” operator. However, the property does not hold if we restrict traces to 100 time units. The corresponding model-checking command presents a counter example trace to that effect:

```

Maude> (mc system |=t [] (first-response -> <> timeout) in time <= 100 .)
rewrites: 35567 in 4332ms cpu (4345ms real) (8209 rewrites/second)
Result ModelCheckResult :
  counterexample({{< od : Decls | decl :(( module 'client is let a = true
  ...
  [self,vpid(3)],msg : nil,name : 'user,tmr : empty >} in time 41,'tick})

```

4 Proper Use of Timing Abstractions

In order to be able to model a wide range of software systems with real-time components, the timing abstractions of $\mathcal{L}$ are designed so that they are expressive and flexible. However, such flexibility might enable unintended or undesirable usage patterns of these abstractions. We overview in this section possible usage problems with timers and discuss automatic means to detect them.

We consider again our working example specification **CLIENT** shown in Figure 2. There are several possible misuses of timer-related constructs in **CLIENT** which would render the specification erroneous or unnecessarily complex. For instance, by dropping any one of the release commands in this specification or by dropping any one of the receive case branch statements, we introduce possible execution paths along which a timer is set but never released or processed. Moreover, by adding any new release command or case statement to this specification,

we essentially introduce dead code that is either superfluous or even unreachable along any execution path in the specification. We note that such problems become more pronounced as specifications get larger and more complex.

The fundamental reason behind such potential problems is flexibility. Indeed, timers of a process are globally scoped within that process. Furthermore, the *set* and *release* statements are not tightly coupled together, which implies that complex timer patterns are possible. Finally, the unified treatment of timer signals and incoming messages in **receive** statements might also add to the conceptual complexity of properly using timers. It is worth mentioning that most of these characteristics are shared with timer-based specification languages such as SDL, making these languages, too, vulnerable to mishandled timers.

The problem, which we call *Mishandled Timers*, identifies usage patterns of timers that could potentially cause semantic or structural problems with specifications in $\mathcal{L}$. It consists of three sub-problems:

1. *Unhandled timers*: a timer is not properly handled in a specification if there exists a possible execution path along which a timer is set but then neither dropped nor its signal is ever consumed.
2. *Extra release commands*: a **release** command is extra if it attempts to drop a timer that is always properly handled along all execution paths to it.
3. *Unreachable case branches*: a **receive** case branch is unreachable if the timer whose signal is being checked is always properly handled along all execution paths to that case branch.

The significance of such analysis revolves around both specification correctness and optimization. Unhandled timers immediately indicate a problem in the specification, since the meaning of an unhandled timer is not clear. Both extra release commands and unreachable case branches might also be the result of an accidentally missed **set** command and can therefore change the intended semantics. In the case that no **set** command was missed, such superfluous statements can as well be eliminated to optimize the specification.

Fortunately, the mishandled timers problem can be formulated as a data-flow analysis problem, and can therefore be checked automatically using standard static analysis means. Instead of defining a static checker that is specific to the mishandled timers problem, we develop a general static analysis framework to be integrated with the specification language $\mathcal{L}$ so that different other static analyses can be easily specified and used. We describe below the static analysis framework and its instantiation to the mishandled timers problem.

5 Static Analysis of Specifications in $\mathcal{L}$

The formal analysis tools and techniques provided by RTM and described above are very useful for analyzing specifications in $\mathcal{L}$ and verifying properties about them. However, due to the dynamic nature of the analysis, such properties are necessarily specific to the specification in hand, and an initial state must be constructed for them to be carried out. For example, for *CLIENT*, the property

that the system will never be in a quiescent state before 160 time units have passed applies only to this specification and was verified against one possible initial state defined by **system**.

Another class of formal verification techniques with which generic properties can be automatically verified can be obtained through static analysis. Static analysis is an automated formal analysis technique that is based on the static structure of specifications rather than their dynamic behavior. The analysis allows for the verification of a different class of properties dealing with the proper use of constructs in $\mathcal{L}$. These properties are generic in the sense that they are not tied to any particular specification and do not depend on any given initial state. As a result, a library of static analysis properties can be built and reused to check specifications in $\mathcal{L}$ for common bugs or to perform common optimizations, which considerably increases the usefulness and effectiveness of $\mathcal{L}$ and its associated tools as a software specification framework.

5.1 A Generic Abstract Interpretation Framework

The approach to static analysis we use is based on the well-studied framework of Abstract Interpretation [12], which enables building safe approximations of a given concrete semantics, so that if a property holds in the abstract semantics, it also holds in the concrete semantics. Specifically, we use control flow graphs (CFGs) to build such abstract interpretations. A CFG for a specification S consists of a set of nodes, representing commands (or basic blocks) in S , and a set of directed edges, representing possible immediate flows between commands.

We have specified our abstraction framework for $\mathcal{L}$ as an equational theory and implemented it in Maude as a functional module. The module defines an operator **cfg**, which, given a specification in $\mathcal{L}$, builds a flattened graph as a set of nodes and directed edges grouped together using the associative and commutative empty juxtaposition operator. A node in a CFG is a pair $\langle I : B \rangle$, consisting of an identifier I and a statement B corresponding to the command represented by that node, while a directed edge is a triple $[I1 : S : I2]$, consisting of identifiers $I1$ and $I2$ for the source and target nodes, respectively, and an abstract state S on that edge, which is used for analysis. The CFG construction process is defined inductively over the structure of commands in $\mathcal{L}$, and computation of fixed points is specified by straight-forward equations that are mostly facilitated by Maude's efficient associative-commutative matching algorithms on the flattened graph. For instance, the following equation specifies the effect of the assignment command (**ceq** introduces a conditional equation):

$$\begin{aligned} \text{ceq } [I1 : S : I] < I : x := e > [I : S' : I2] \\ = [I1 : S : I] < I : x := e > [I : S'' : I2] \\ \text{if } S'' := \text{assign}(S, x, e) \wedge S' < S'' . \end{aligned}$$

where **assign**(S, x, e) is the transfer function for assignment and $<$ is the strict partial ordering relation on abstract states. The particular definitions of transfer functions, abstract states, and the ordering relation are dependent on the specific property to be analyzed and are therefore left unspecified in the

abstraction framework. Below, we give an instantiation of it for the analysis of the mishandled timers problem.

5.2 Mishandled Timers

We formulate the mishandled timers problem as a data-flow analysis problem, and then use the abstract interpretation framework described above to automatically check for it. The analysis computes, at each point in a specification, the set of timers that may have not been properly handled on some path to that point in the specification.

By computing such intermediate states, we can build decision procedures to detect misuses of timers as follows. We first define the abstract domain to be a simple lattice $\{\top, \perp\}$ with the usual ordering. A timer variable is mapped to $\top$ in an abstract state if it references a timer that may not have been handled in that state. Otherwise, it is mapped to $\perp$. The abstract state is a valuation from timer variable names to values in the lattice. Both the lattice ordering and the join operation are extended in the usual way to abstract states.

Then, we define, for each command in $\mathcal{L}$, the transfer function that specifies the effect of that command on the abstract state. Most of these functions are fairly trivial to define for this problem since most functions are the identity function on states, except for the commands **set** and **release**, for which the transfer functions respectively map variable names to $\top$ and to $\perp$. Furthermore, the transfer function for the conditional command is defined to reflect the possible change in state in the true and false branches of several other commands, such as receive case statements. Finally, we define the following operators that will automatically check the three problems : (1) **utimers** for unhandled timers, (2) **ers** for extra release commands, and (3) **ecs** for unreachable case branches.

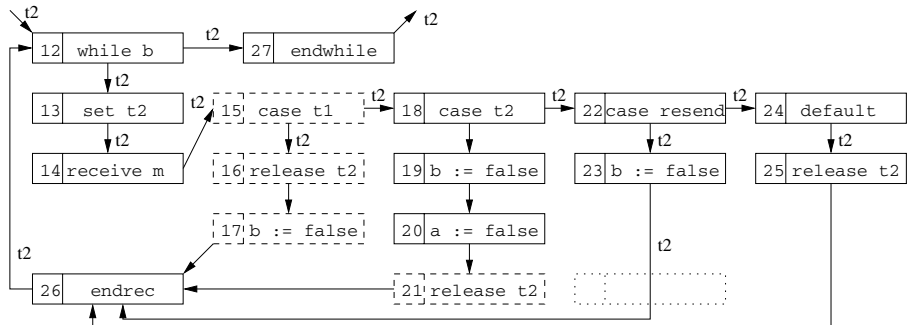


Fig. 3. A partial CFG computed by the mishandled timers analysis for the innermost while loop of a modified (buggy) version of the **client** example; dashed nodes are the ones added, while the dotted node is a **release t2** node that was removed . Only timer variables mapped to $\top$ in abstract states are shown on edges.

To illustrate the use of these operators, we apply them to an instrumented version of the `client` specification, named `BuggyClient`, to which we introduced some instances of the mishandled timers problem. The CFG for the innermost **while** (*b*) in the modified specification along with internal results of the analysis algorithm are shown in Figure 3. As is clear from the figure, timer `t2` is not properly handled, which can be automatically realized by the command¹

```
Maude> red utimers(cfg(BuggyClient)) .
rewrites: 47660 in 58ms cpu (59ms real) (807919 rewrites/second)
result State: [t2,top]
```

and which is resulting from a missing `release` statement within the case branch labeled 22. Moreover, the release command labeled 21 is extraneous, which can be checked by issuing the command:

```
Maude> red ers(cfg(BuggyClient)) .
rewrites: 47675 in 59ms cpu (59ms real) (794702 rewrites/second)
result Node: < 21 : release t2 >
```

Similarly, by executing the command `red ecs(cfg(BuggyClient))`, we can verify unreachability of the case branch labeled 15 in the figure.

6 Related Work

Real-time languages, for which a large body of research exists, differ widely in terms of the timing abstractions they support and their semantics depending mainly on their targeted application domains. The closest languages to our design of timing abstractions are SDL [5,13], a high-level specification language, and Erlang [4], which is a programming language based on the Actors model [14] for distributed, soft real-time systems. Both languages are based on a concurrent process model, and they both use timers and check for timer signals as incoming messages. However, our design has a stricter timers semantics than that of SDL and is much more expressive than Erlang's (some nested timing patterns, which can be expressed in $\mathcal{L}$, are not expressible in Erlang). There has also been some attempts at improving the timing abstractions in SDL for specification writers, such as the work in [15] on extending timers with annotations and supporting transitions with urgencies. Many other timed high-level languages exist [16,17,18].

Real-time rewrite theories and their implementations in Real-Time Maude have been used in the specification and analysis of various protocols and algorithms [19,20,21,22]. Our application is fundamentally different though as it applies these methods to a specification language rather than a protocol or an algorithm, which has subtle consequences in terms of design and analysis.

Finally, the technique of abstract interpretation [12] has been successfully applied over the years to static analysis (see [23] for a recent survey, and [24] on its use for data-flow analysis). In particular, the technique has been applied to validation of timing requirements [25] and for more efficient model checking [26].

¹ The Maude command `red` or `reduce` evaluates the given expression according to the equations and memberships of the current module.

7 Conclusion

In this paper, we presented a new simple specification language with formal semantics that can be used to specify and analyze timing behaviors of software systems. Our specification language is flexible and supports, through translation, the timing models of various other high level specification languages like SDL and UML. Our formal semantics is defined as a real-time rewrite theory. This automatically gives us the ability to perform simulation and trace analysis using the RTM tool. Furthermore, we take advantage of the timed model checker provided with RTM, to provide an integrated analysis framework for software designers. Finally we show how to use traditional abstract interpretation based approaches to detect common misuses of timing constructs, thus automatically preventing some of the common errors that a software designer can make when using the flexible timing constructs. Together, we believe that our approach provides a significant step forward in facilitating the use of formal tools for specification and analysis of timing behaviors in software development.

Acknowledgements. Many thanks to José Meseguer for his comments.

References

1. Wang, F.: Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE* 92(8), 1283–1305 (2004)
2. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* 126(2), 183–235 (1994)
3. Merlin, P., Farber, D.: Recoverability of communication protocols - implications of a theoretical study. *IEEE Tran. on Comm.* 24(9), 1036–1043 (1976)
4. Barklund, J., Virding, R.: Specification of the standard Erlang programming language, Draft version 0.7 (June 1999)
5. ITU-T: Recommendation Z.100(08/02), languages and general software aspects for telecom. systems - specification and description language (SDL) (August 2002)
6. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science* 285, 359–405 (2002)
7. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation* 20(1-2), 161–196 (2007)
8. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.* 96(1), 73–155 (1992)
9. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) *WADT 1997*. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
10. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.* 360(1-3), 386–414 (2006)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: *All About Maude - A High-Performance Logical Framework*. LNCS, vol. 4350. Springer, Heidelberg (2007)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *POPL 1977*, pp. 238–252. ACM, New York (1977)

13. ITU-T: Recommendation Annex F1(11/00), languages and general software aspects for telecom. systems - SDL formal semantics definition (November 2000)
14. Agha, G.: *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge (1986)
15. Bozga, M., Graf, S., Mounier, L., Ober, I., Roux, J.L., Vincent, D.: Timed extensions for SDL. In: Reed, R., Reed, J. (eds.) *SDL 2001*. LNCS, vol. 2078, pp. 223–240. Springer, Heidelberg (2001)
16. Tardieu, O.: A deterministic logical semantics for Pure Esterel. *ACM Trans. Program.* 29(2), 8 (2007)
17. Taft, S.T., Duff, R.A., Brukardt, R.L., Ploedereder, E., Leroy, P.: *Ada 2005 Reference Manual*. LNCS, vol. 4348. Springer, Heidelberg (2006)
18. Bollella, G., Gosling, J.: The real-time specification for Java. *Computer* 33(6), 47–54 (2000)
19. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design* 29(3), 253–293 (2006)
20. Ölveczky, P.C., Prabhakar, P., Liu, X.: Formal modeling and analysis of real-time resource-sharing protocols in Real-Time Maude. In: *22nd Int'l Parallel and Distributed Processing Symp. (IPDPS 2008)*. IEEE Computer Society Press, Los Alamitos (2008)
21. Ölveczky, P.C., Caccamo, M.: Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In: Baresi, L., Heckel, R. (eds.) *FASE 2006*. LNCS, vol. 3922, pp. 357–372. Springer, Heidelberg (2006)
22. Ölveczky, P.C., Grimeland, M.: Formal analysis of time-dependent cryptographic protocols in Real-Time Maude. In: *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*. IEEE Computer Society Press, Los Alamitos (2007)
23. Cousot, P.: Abstract interpretation and application to static analysis (invited tutorial). In: *First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE 2007*, Shanghai, China (June 2007)
24. Nielson, F., Nielson, H.R., Hankin, C.: *Principles of Program Analysis*, 2nd printing edn. Springer, Heidelberg (2005)
25. Wilhelm, R., Wachter, B.: Abstract interpretation with applications to timing validation: Invited tutorial. In: Gupta, A., Malik, S. (eds.) *CAV 2008*. LNCS, vol. 5123, pp. 22–36. Springer, Heidelberg (2008)
26. Ioustinova, N., Sidorova, N.: A transformation of SDL specifications - a step towards the verification. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) *PSI 2001*. LNCS, vol. 2244, pp. 64–78. Springer, Heidelberg (2001)

Formal Foundation for Pattern-Based Modelling

Paolo Bottoni¹, Esther Guerra², and Juan de Lara³

¹ Università di Roma “Sapienza”, Italy
`bottoni@di.uniroma1.it`

² Universidad Carlos III de Madrid, Spain
`eguerra@inf.uc3m.es`

³ Universidad Autónoma de Madrid, Spain
`jdelara@uam.es`

Abstract. We present a new visual and formal approach to the specification of patterns, supporting pattern analysis and pattern-based model completion. The approach is based on graphs, morphisms and operations from category theory and exploits triple graphs to annotate model elements with pattern roles. Novel in our proposal is the possibility of describing (nested) variable submodels, as well as inter-pattern synchronization across several diagrams (e.g. class and sequence diagrams for UML design patterns). We illustrate the approach on UML design patterns, and discuss its generality and applicability on different types of patterns, e.g. workflow patterns using Coloured Petri nets.

1 Introduction

Patterns are increasingly used in the definition of software frameworks, as well as in Model Driven Development, to indicate parts of required architectures, drive code refactorings, or build model-to-model transformations. The full realisation of their power is however hindered by the lack of a standard formalization of the notion of pattern. Presentations of patterns are typically given through natural language, to explain their motivation, context and consequences; programming code, to show usages of the pattern; and diagrams, to communicate their structure and behavior. However, the use of domain modelling languages, such as UML for design patterns, or Coloured Petri Nets for workflows, forces pattern proposers to provide only examples of their realisation, appealing to intuition to extend them to the complete semantics of the pattern. For instance, the fact that in the *Visitor* pattern there must be a distinct operation in the *Visitor* interface for each *ConcreteElement* is only understood through generalisation of the examples or reading the associated text [4].

The search for a general definition of patterns, independent from the specific modelling language, has led to several proposals based on the association of constraints to diagrammatic definitions, or to extensions of the UML meta-model, in order to distinguish roles from concrete modelling types. However, these proposals incur some problems to define the relations between different components of a patterns specification, as will be illustrated in Section 2.

We propose a formal notion of pattern, grounded in category theory [9], which sees them as formed of: i) a vocabulary of roles; ii) a collection of diagrams, possibly in different modelling languages, defining cardinalities and associations between roles, and supplemented with indications of variable regions; iii) a collection of interfaces between these diagrams, to identify roles across different diagrams. Two main results are thus obtained, representing a distinguishing novel feature of the approach. First, dependencies among different components of a pattern specification can be formally defined. Second, a clear specification of the parts of the pattern which can be replicated (and of the admissible number of replicas) is given, without recurring to concrete examples. As our notion of pattern generalises patterns from different fields, it opens the way to the extension of pattern-based techniques to new areas, to the formalisation of existing ones, and in general, to the development of pattern-based languages.

The paper is organized as follows. Section 2 presents related approaches. Section 3 introduces our new definition of pattern. Section 4 shows the procedure for pattern application. Finally, Section 5 ends with the conclusions.

2 Related Work

The shortcomings of presenting patterns in the GoF (Gang of Four [4]) style have been addressed by several researchers, who advocate a more formal approach. For example, [3] extends the UML meta-model for class diagrams with specific roles and constraints. Conformance of a model to a pattern is checked as the usual model/meta-model relationship. The technique works for UML class and sequence diagrams, and the emphasis is on specification of patterns, but not on their use for model completion. A non-uniform interpretation of interaction diagrams is provided, where reference to interaction fragments is translated to their unfolding with respect to the instantiation of roles.

The limitations posed by reference to UML diagrams, in particular as regards premature commitment to hierarchical structures, are overcome in [10] by extending the UML meta-model with stereotypes accounting for possible realizations of a given pattern. A distinction between roles, types and classes is used in [8] to decouple representations of roles, at which level the semantics of the pattern is abstractly described by incorporating constraint diagrams into the UML notation, from their refinement as types, and their implementation into classes. This way, the GoF presentation of patterns is shown to be a realization of the abstract level. However, commitment to names and multiplicities is already needed at the type level, with the class level providing concrete implementations. Our proposal, on the contrary, provides the separation through triple graphs which establish a correspondence between roles in a pattern model and types in a pattern specification. Variability is thus maintained also at the type level, leaving matching morphisms to provide the relation with any given realization of the pattern. Independence from UML is achieved in [7] by using Object-Z, but it is limited to structural aspects. Constraints on the use of patterns are exploited in [14] to maintain the consistency of a pattern-based

software framework through the use of high-level transformations specific to each pattern.

In [1] a method is proposed for visualizing the roles of the elements in UML class and communication diagrams. The technique extends the UML Profile meta-model with stereotypes and tagged values for pattern annotation. This also allows pattern composition through single instances. In [6] the intent of design patterns is described with an ontology, which can be queried to obtain suggestions about the most appropriate pattern solving a certain problem.

In [13], the authors propose a logic-based approach, using subsets of First Order Logic – for structural aspects – and Temporal Logic of Actions – for behavioral ones – and supporting pattern combinations. As the language does not include implications, support for complex constraints appears limited.

Graph transformation [2] has also been used to formalise patterns. In [11], patterns are represented with rules, applied to abstract syntax trees to annotate the pattern instances found. In [12], models are transformed to conform to patterns, after having exploited graph queries that detect needs for transformations. In [16] Spatial Graph Grammars provide a graph representation of GoF patterns, to transform object structure graphs so that they conform to patterns. Although declarative, the rules are based on a concrete presentation of patterns, and not on a meta-model characterisation.

In general, we observe the lack of an integrated, domain-independent formalism able to give account of mutual synchronization constraints within a pattern and across different ones, and to support pattern checking, identification and application. In the rest of the paper we present such a formalism.

3 Pattern Specification

3.1 Variable Patterns

We define a variable pattern as made of a fixed part *root* and a number of variable parts V_i , which can be replicated according to a given interval (*low*, *high*). Variable patterns support nesting – variable parts inside variable parts – and can be used with any graph model, from unattributed graphs $G = (V_G, E_G, src, tgt : E_G \rightarrow V_G)$, to typed node and edge attributed graphs (e.g. E-graphs [2]). Here we use typed attributed graphs and injective morphisms.

Definition 1 (Variable Pattern). *A variable pattern is defined as $VP = (P = \{V_i\}_{i \in I}, root \in P, int : P \rightarrow \mathbb{N} \times (\mathbb{N} \cup \{*\}), Emb = \{v_{i,j} : V_i \rightarrow V_j | \exists V_i, V_j \in P\})$, where V_i are non-empty graphs, $v_{i,j}$ are injective morphisms, Emb is a tree with graphs $V_i \in P$ as nodes and morphisms $v_{i,j}$ as edges, rooted in $root \in P$, and int is a function returning the variability interval for each graph $V_i \in P$.*

A finite set S satisfies interval (l, h) , written $S \triangleleft (l, h)$, iff $|S| \geq l$ and $h = * \vee |S| \leq h$. Note that if (l, h) is such that $l > h$, then it cannot be satisfied.

Example. The GoF *Observer* pattern captures one-to-many dependencies between objects so that when the *subject* object changes its state, all the dependent *observer* objects are notified and updated automatically. Fig. 1 presents a

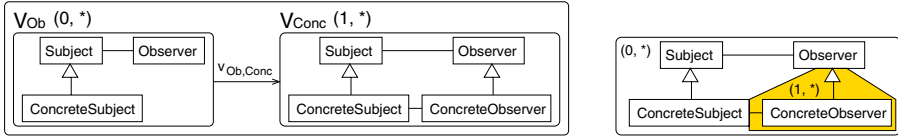


Fig. 1. The Observer Pattern in Theoretical (left) and Compact (right) Forms

simplified version of this pattern using Def. 1. The left part directly encodes the theoretical form, with full definition $VP = (P = \{V_{Ob}, V_{Conc}\}, root = V_{Ob}, int = \{(V_{Ob}, (0, *)), (V_{Conc}, (1, *))\}, Emb = \{v_{Ob, Conc}\})$. The variability interval is related to the satisfiability of the pattern by a model. V_{Ob} has $(0, *)$ as variability interval, thus the pattern is not mandatory and we may have any number of instances. As $int(V_{Conc}) = (1, *)$, we require at least one **ConcreteObserver** in each pattern instance. The right part of the figure shows a compact form, where fixed and variable parts are presented together. In the figure, we use the concrete syntax of UML class diagrams, but the abstract syntax can also be used. ■

The next definition states when a given graph satisfies a variable pattern.

Definition 2 (Pattern Satisfaction). *Given a graph G and a variable pattern VP as in Def. 1, G satisfies VP , written $G \models VP$, iff:*

- $M_{root} = \{p_{root}^k: root \rightarrow G\} \triangleleft int(root)$.
- $\forall v_{i,j}: V_i \rightarrow V_j \in Emb$:
 - $\forall p_i^k \in M_i, M_j^k = \{p_j^l: V_j \rightarrow G | p_i^k = p_j^l \circ v_{i,j}\} \triangleleft int(V_j)$.
 - Define $M_j = \bigcup M_j^k$, with $k = 1..|M_i|$.

Remark. The procedure in Def. 2 induces a tree traversal for Emb , because when $V_i \rightarrow V_j$ is traversed, M_i must have been previously calculated. ■

The left of Fig. 2 describes the morphisms for the satisfaction checking for a variable pattern with one level of nesting, i.e. $Emb = \{root \rightarrow V_i, V_i \rightarrow V_j\}$. We assume one morphism $p_{root}^1 \in M_{root}$. The procedure checks that the set M_i^1 of all

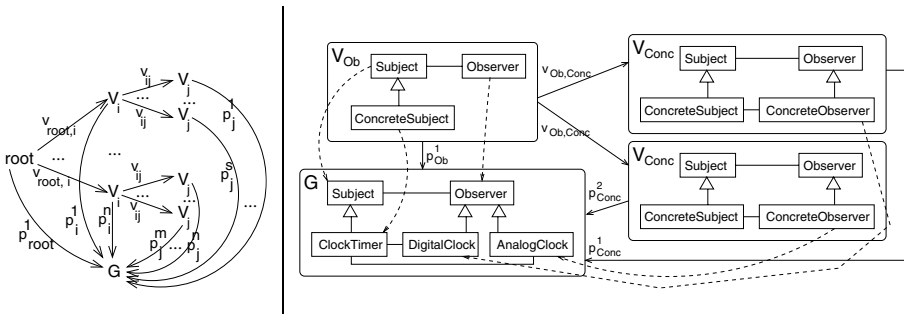


Fig. 2. Pattern Satisfaction (left). Pattern Satisfaction Example (right).

morphisms $p_i^1, \dots, p_i^n: V_i \rightarrow G$ commuting with $G \xleftarrow{p_{root}^1} root \xrightarrow{v_{root,i}} V_i$ satisfies the interval $int(V_i) = (low_i, high_i)$. Similarly, given each morphism $p_i^k \in M_i = M_i^1$, each set M_j^k of morphisms $p_j^1, \dots, p_j^s: V_j \rightarrow G$ commuting with $G \xleftarrow{p_i^k} V_i \xrightarrow{v_{i,j}} V_j$ satisfies the interval $int(V_j)$. In the figure, we have replicated V_j and V_i for each different morphism p_j^k and p_i^k , to show more intuitively the tree traversal.

Example. The right of Fig. 2 shows a model G that satisfies the specification of the observer pattern of Fig. 1. The fixed part V_{Ob} occurs once, and the variable region V_{Conc} twice. This is checked by building the set $M_{Conc} = \{p_{Conc}^1: V_{Conc} \rightarrow G, p_{Conc}^2: V_{Conc} \rightarrow G\} \triangleleft (1, *)$. In the figure, the dotted lines indicate some of the mappings induced by p_{Ob}^1 , p_{Conc}^1 and p_{Conc}^2 . ■

3.2 Annotating Structure with Roles: Triple Patterns

In order to specify a pattern, we need its structure (as given by a variable pattern), a vocabulary of pattern roles, and a mapping from the elements in the pattern to the vocabulary. We call this structure *annotated pattern* and ground it in the notion of triple graph [5], composed of two graphs, *source* and *target*, related through a *correspondence* graph. Nodes in the correspondence graph (also called *mappings*) have morphisms to nodes or edges in the other two graphs. Thus, mappings may relate two nodes, two edges, an edge and a node, or just point to a single element in the source or target graphs [5].

Definition 3 (Triple Graph). A triple graph $TrG = (G_s, G_c, G_t, c_s, c_t)$ has three graphs G_i ($i \in \{s, c, t\}$), and two functions $c_j: V_{G_c} \rightarrow V_{G_j} \cup E_{G_j} \cup \{\cdot\}$ ($j = s, t$).

The element “ $\cdot$ ” in the codomain of c_j denotes that the correspondence function c_j can be undefined. A triple graph is also written as $(G_s \xleftarrow{c_s} G_c \xrightarrow{c_t} G_t)$. We say that a node or edge x of G_s is related to a node or edge y of G_t iff $\exists n \in V_{G_c}$ s.t. $x \xleftarrow{c_s} n \xrightarrow{c_t} y$ and we write $x \text{ rel}_{TrG} y$. As [5] showed, triple graphs and morphisms form the category¹ **TrGraph** [9].

Example. Fig. 3 shows a triple graph. Its source graph G_s at the bottom conforms to the UML meta-model, its target G_t at the top is a pattern vocabulary model, and the correspondence graph G_c maps UML elements to vocabulary elements. The morphisms from the nodes in G_c are shown as dotted arrows. The correspondence function c_s is undefined for node `:PatternInstance` (i.e. $c_s(\text{:PatternInstance}) = \cdot$) and this is represented by omitting the edge. ■

Triple graphs are typed by meta-model triples [5] made of two meta-models, source and target, related through a correspondence meta-model. In our case, a meta-model triple relates the meta-model of a specific language (e.g. UML) with that of a generic pattern vocabulary, which can be refined by subclassification in

¹ A category is made of objects (e.g. triple graphs) and arrows (e.g., triple morphisms) satisfying some conditions [9].

order to define patterns for the language. The top of Fig. 4 shows the meta-model for the vocabulary. All classes except the subclasses of **PatternRole** allow defining patterns in any language. A **Pattern** has a name and a type, contains participating roles, and can be documented with its motivation, applicability, intent and consequences. Relations between patterns are given through the **Relation** association class. We have specialized the meta-model for UML, so that roles are applicable to operations, classifiers, classes, structural features and associations. The bottom of Fig. 4 partially shows the UML meta-model, and the correspondence meta-model maps roles to UML elements. The **PatternInstance** class is used to group the mappings of each pattern instantiation.

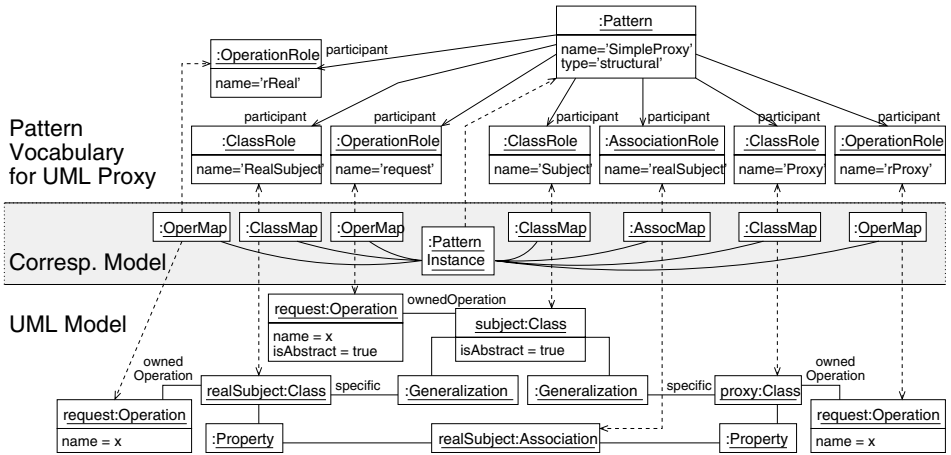


Fig. 3. Annotated Pattern with the Structural Part of the Proxy (in Abstract Syntax)

A *pattern-annotated model* is a triple graph whose source is a model in some language (e.g. UML), and the target contains a subgraph isomorphic to a pattern vocabulary, for each pattern used in the source model. The correspondence graph is called *annotation graph* and has a **PatternInstance** node for each pattern instance and one **RoleMap** for each element playing a role in the instance. For example, the annotated pattern in Fig. 3, specifying the structure of the *Proxy* pattern [4], conforms to the meta-model triple in Fig. 4. The intent of this design pattern is to provide a surrogate or placeholder *proxy* for an object with role *realSubject* in order to control access to it. The bottom part of Fig. 3 uses the abstract syntax for UML class diagrams. For simplicity, we only show the pattern and the roles in the vocabulary model. The pattern has only one fixed graph, no variable part, and requires that the operations in **Subject**, **RealSubject** and **Proxy** have the same name, which is modelled with a variable *x*. More complex attribute conditions such as in [2] are possible. We have omitted the name of classes and associations, so they can be mapped to any name.

An *annotated pattern* is a variable pattern where all the graphs in Def. 1 are triple graphs, and morphisms are triple morphisms. The notion of pattern

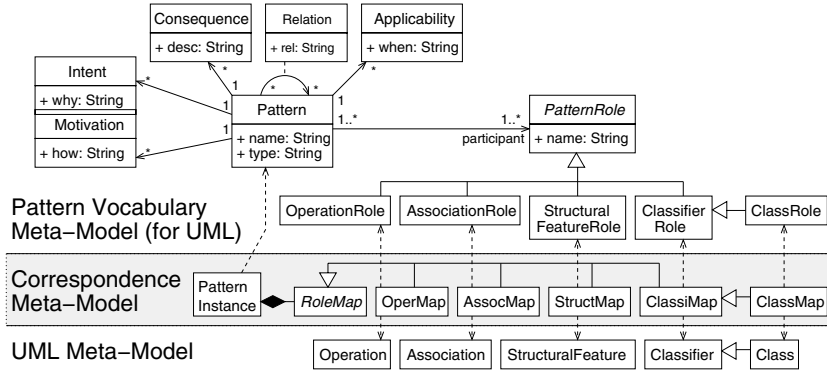


Fig. 4. Meta-model Triple for UML Patterns

satisfaction remains as in Def. 2, but using triple graphs. Thus, annotated patterns are satisfied by triple graphs, called pattern-annotated models.

Example. Workflow patterns [15] collect recurring constructs from existing workflow systems and provide descriptions of their usage. Their presentation is textual in the style of GoF patterns. For control flow patterns, dealing with synchronization policies, an intuitive semantics is given through Coloured Petri Nets showing example realizations of the pattern, to be inferred by the reader. Fig. 5 shows three annotated patterns expressed via a syntax identifying the roles of places and transitions, as *input*, *split*, *output*, *and*, and *merging*. ■

As seen before, a tool need not show the triple graph to the user, but the annotation can be done by marking the source graph [1]. However for the theory, an explicit triple graph has some advantages: (i) we do not modify or extend the source meta-model (e.g. the UML one) with additional classes or attributes for tagging; (ii) triple graphs help in enforcing the patterns, as shown in Section 4; (iii) it is easier to distinguish the instances of a pattern, as these are identified by `PatternInstance` nodes.

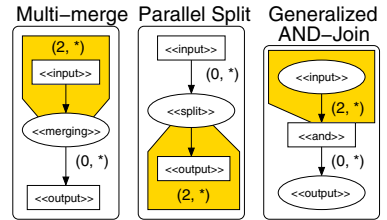


Fig. 5. Workflow Patterns

3.3 Synchronizing Different Variable Patterns

A pattern specification can be composed of more than one diagram. For instance, the GoF patterns are described using class and sequence diagrams. Thus, relationships have to be established between the elements in the different diagrams, which we do through their roles in the pattern.

Example. The *Visitor* pattern explicitly represents as objects the operations to be performed on the elements of an object structure, so that new operations can

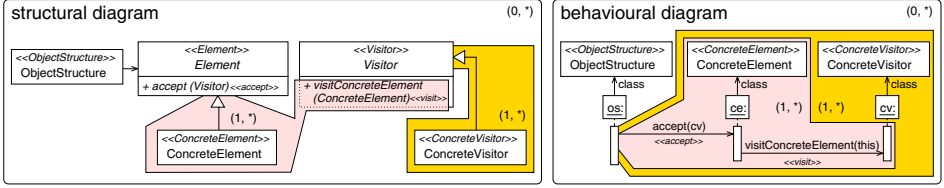


Fig. 6. Visitor Pattern

be defined without changing the classes of the elements on which they operate. It is one of the most complex GoF patterns as it requires two levels of variation for the two hierarchies of **Visitor** (i.e. the operations) and **Element** (i.e. the object structure), with the set of operations for **Visitor** varying together with the **ConcreteElement** set. This covariance cannot be expressed through a constraint on cardinalities, as it requires an exact match on types of parameters. The pattern is shown in Fig. 6, where the presence in the same variance region of the signatures for the **visit** operations constrains the types of their parameters to satisfy the constraint of equality with the types of **ConcreteElement**.

Both the structural and the behavioral part of *Visitor* require two variable parts, relative to the two hierarchies. However, while in the structural part the two regions are independent, in the behavioral part they are nested. This reflects the double constraint that each concrete visitor can be accepted by each concrete element and that each concrete visitor has an operation to visit each concrete element. The synchronization between the different regions involved is represented by the equality of the colours used in the different pattern graphs, and formally as a *synchronization graph*.

Fig. 7 shows the scheme of the two patterns for *Visitor*: $SP : V_1^{SP} \leftarrow V_0^{SP} \rightarrow V_2^{SP}$ and $IP : V_0^{IP} \rightarrow V_1^{IP} \rightarrow V_2^{IP}$. SP has V_0^{SP} as root and two variable parts, and describes the structural part (a class diagram).

IP has root V_0^{IP} and a variable part V_1^{IP} with nested V_2^{IP} , modelling a sequence diagram. The *synchronization graph* $I_{11} \leftarrow I \rightarrow I_{22}$ declares the intersections between the roots and pairs of variable parts of each pattern. Morphism $I \rightarrow I_{11}$ is derived as we have $V_0^{SP} \rightarrow V_1^{SP}$ and $V_0^{IP} \rightarrow V_1^{IP}$, and similar for $I \rightarrow I_{22}$. All squares in the diagram must commute to ensure coherence, so $I \rightarrow V_0^{SP} \rightarrow V_1^{SP} = I \rightarrow I_{11} \rightarrow V_1^{SP}$, and similar for V_0^{IP} with I_{11} and I_{22} . ■

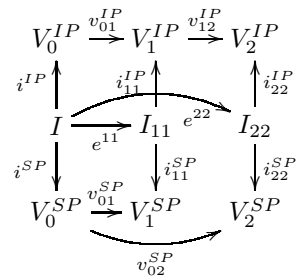


Fig. 7. Synchronization

Example. Fig. 8 shows a variation of the *Proxy* pattern, in concrete syntax, allowing several proxies for a given subject ($V_0^{SP} \rightarrow V_1^{SP}$). The upper part ($V_0^{IP} \rightarrow V_1^{IP}$) shows the annotated sequence diagram. For clarity, we show the classifiers of the **p** and **rs** objects, as both classifiers play a role in the pattern. ■

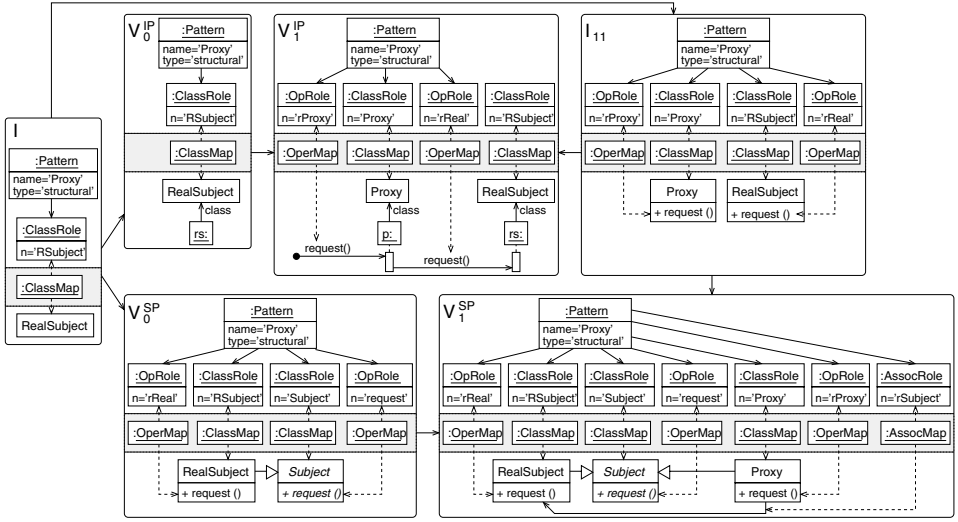
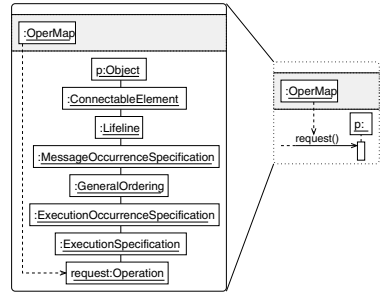


Fig. 8. Synchronization Example for the Proxy Pattern

In Fig. 8 and others, we use a shortcut notation, shown below, for sequence diagrams. The **OperMap** node in the annotation graph points to the message arrow for the invocation, while in the abstract syntax the morphism reaches the operation to be executed through a **MessageOccurrenceSpecification** object. The two variable patterns to be synchronized share the same vocabulary model, as they describe different diagrams of the *same* pattern. Given the parts to be synchronized, the synchronization graph is automatically calculated by the intersections w.r.t. the roles. Thus, two elements in two patterns SP and IP , if mapped to the same role, will be present in I .

Once the intersection graphs I_{ij} (and the morphisms $V_i^{SP} \leftarrow I_{ij} \rightarrow V_j^{IP}$) are derived, the morphisms of the synchronized graph are derived as well. Morphism $I_{ik} \rightarrow I_{jl}$ is added, iff there is a path $V_i^{SP} \rightarrow V_j^{SP}$ in the *Emb* tree of SP and a path $V_k^{IP} \rightarrow V_l^{IP}$ in the *Emb* tree of IP .



Definition 4 (Synchronization Graph). Let $AP^k = (P^k = \{V_i^k\}_{i \in I^k}, \text{root}^k \in P^k, \text{int}^k, \text{Emb}^k = \{v_{i,j}^k: V_i^k \rightarrow V_j^k\})$, $k = \{1, 2\}$, be two annotated patterns on the same vocabulary VP ; their synchronization graph $SG = (V = \{\text{root}^1 \xleftarrow{I^1} I \xrightarrow{I^2} \text{root}^2, \dots, V_i^1 \xleftarrow{I_{ij}^1} I_{ij} \xrightarrow{I_{ij}^2} V_j^2, \dots\}, E = \{e_{ij}^{kl}: I_{ij} \rightarrow I_{kl}\})$ is calculated as follows:

- For each pair of triple graphs $V^k = (V_s^k \leftarrow V_c^k \rightarrow V_t^k)$ to be synchronized:

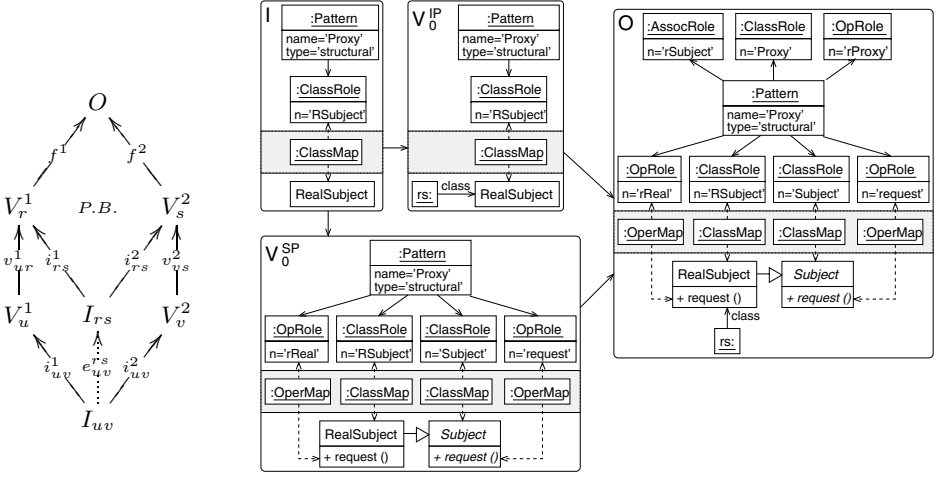


Fig. 9. Calculating Intersection Edges (left). Example (right).

- Build the triple graph $O = (O_s \leftarrow O_c \rightarrow VP)$, and triple morphisms $V^1 \xrightarrow{f^1} O \xleftarrow{f^2} V^2$, jointly surjective on O_s and O_c as follows: (i) VP is the common vocabulary model, so there are inclusions $V_t^k \hookrightarrow VP$ (ii) if two elements $a \in V_s^1, b \in V_s^2$ are mapped to the same role in VP , (i.e. $a \text{ rel}_{V^1} x, b \text{ rel}_{V^2} x$) only one element ab is added to O_s , and the functions f^1 and f^2 identify a and b to such element, $f^1(a) = ab = f^2(b)$ (if ab does not exist, the graphs cannot be synchronized).
- Intersection I is given by the pullback $^2 V^1 \xleftarrow{i^1} I \xrightarrow{i^2} V^2$ of $V^1 \xrightarrow{f^1} O \xleftarrow{f^2} V^2$.
- Given two intersection nodes $V_r^1 \leftarrow I_{rs} \rightarrow V_s^2$ and $V_u^1 \leftarrow I_{uv} \rightarrow V_v^2$ s.t. there are paths $V_u^1 \rightarrow V_r^1$ in Emb^1 and $V_v^2 \rightarrow V_s^2$ in Emb^2 , add edge $e_{uv}^{rs}: I_{uv} \rightarrow I_{rs}$ to SG , as morphism e_{uv}^{rs} uniquely exists due to the pullback universal property [2], see the left of Fig. 9.

Remark. Two graphs V^1 and V^2 can be synchronized only if their variability intervals overlap (i.e. $\exists M | M \triangleleft \text{int}^1(V^1)$ and $M \triangleleft \text{int}^2(V^2)$).

Example. The right of Fig. 9 shows an example of the calculation of an intersection graph for the synchronization scheme shown in Fig. 8. ■

3.4 Full Pattern Specification

We now put all elements together to define a full pattern, made of a primary structuring pattern SP , and a number of secondary patterns IP_i synchronized with SP through synchronization graphs SG_i .

² Roughly, a pullback [9] is the biggest intersection of two objects A_i through a common one B to which both are mapped. The pullback $A^1 \leftarrow C \rightarrow A^2$ identifies the elements of A^1 and A^2 that are mapped to a common element in B .

Definition 5 (Full Pattern Specification). A Full Pattern Specification $PS = (VP, SP, Sec = \{(IP_s, SG_s)\}_{s \in S})$ is composed of:

1. A Pattern Vocabulary VP , mentioning the relevant roles for the patterns.
2. A Structuring Pattern Diagram $SP = (P^{SP} = \{V_i^{SP}\}_{i \in I}, root^{SP}, int^{SP}, Emb^{SP} = \{v_{i,j}^{SP}: V_i^{SP} \rightarrow V_j^{SP}\})$, an annotated pattern where the V_i^{SP} are triple graphs whose target is a subgraph of the pattern vocabulary VP .
3. A set Sec of pairs made of a Secondary Pattern Diagram $IP_s = (P_s^{IP} = \{V_i^{IP}\}_{i \in I_s}, root_s^{IP}, int_s^{IP}, Emb_s^{IP} = \{v_{i,j}^{IP}: V_i^{IP} \rightarrow V_j^{IP}\})$ and a Synchronization Graph $SG_s = (V_s = \{V_i^{SP} \xleftarrow{i_j^{sp}} I_{ij} \xrightarrow{i_j^{ip}} V_j^{IP}\}, E_s = \{e_{ij}^{kl}: I_{ij} \rightarrow I_{kl}\})$, which synchronizes the secondary patterns with the primary one.

For UML patterns, the primary pattern SP is a class diagram, and Sec generally contains a sequence diagram synchronized with the class diagram.

The satisfaction of full patterns is similar to Def. 2, but using annotated patterns and taking into account the synchronization graphs.

Definition 6 (Full Pattern Satisfaction). Given a pattern-annotated model $TrG = (G_s \xleftarrow{c_s^G} G_c \xrightarrow{c_t^G} G_t)$, and a full pattern specification $PS = (VP, SP, \{(IP_s, SG_s)\}_{s \in S})$ as in Def. 5, TrG satisfies PS , written $TrG \models PS$, iff:

- $\forall (IP, SG) \in Sec$ (i.e. for each secondary pattern and synchronization graph):
 - Let $M_{root, root} = \{root^{SP} \xrightarrow{ps^k} TrG \xleftarrow{pi^k} root^{IP} | root^{SP} \xleftarrow{i^{sp}} I \xrightarrow{i^{ip}} root^{IP} \text{ is pullback}\}$, then $M_{root, root} \triangleleft int^{SP}(root^{SP})$ and $M_{root, root} \triangleleft int^{IP}(root^{IP})$.
 - $\forall e_{jk}^{lm}: I_{jk} \rightarrow I_{lm} \in E$ (i.e. for each edge in SG):
 - ◊ $\forall V_j^{SP} \xrightarrow{ps^r} TrG \xleftarrow{pi^r} V_k^{IP} \in M_{j,k}$, let $M_{l,m}^r = \{V_l^{SP} \xrightarrow{ps^u} TrG \xleftarrow{pi^u} V_m^{IP} | ps^u \circ (V_j^{SP} \rightarrow V_l^{SP}) = ps^r \text{ and } pi^u \circ (V_k^{IP} \rightarrow V_m^{IP}) = pi^r \text{ and } V_l^{SP} \xleftarrow{i^{sp}} I_{lm} \xrightarrow{i^{ip}} V_m^{IP} \text{ is pullback}\}$. Then $M_{l,m}^r \triangleleft int^{SP}(V_l^{SP})$ and $M_{l,m}^r \triangleleft int^{IP}(V_m^{IP})$. See Fig. 10(a).
 - ◊ Define $M_{l,m} = \bigcup M_{l,m}^r$, with $r = 1..|M_{j,k}|$.

Example. Fig. 10(b) shows in compact notation the satisfaction of the *Proxy* pattern of Fig. 8. The model contains a class diagram and two sequence diagrams, enclosed in different regions, which a tool would present in three views. The model has one instance of the *Proxy* pattern, and the variability region that affects the *Proxy* role has been instantiated twice (classes *ImageProxy* and *RemoteProxy*). Fig. 10(c) shows the first step in the satisfaction checking, where the pullback of the fixed parts is depicted, corresponding to the intersection graph shown to the right of Fig. 9. The satisfaction check follows by computing two additional pullbacks for the two instantiations of the variable parts. ■

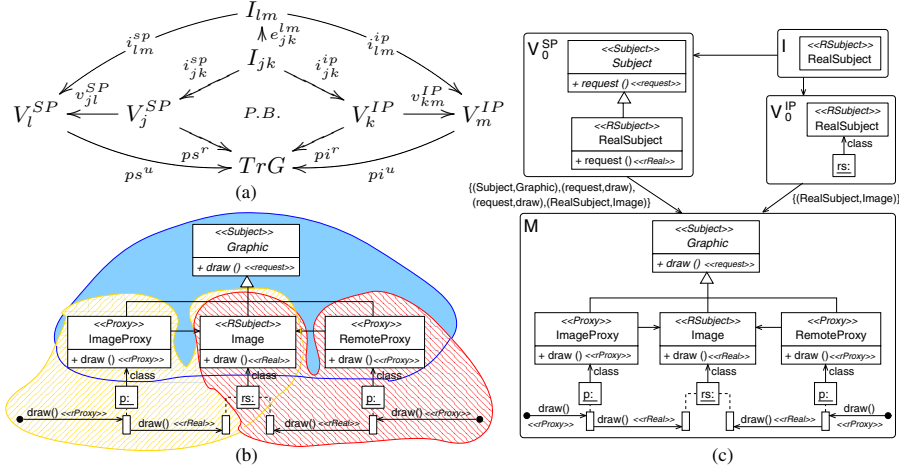


Fig. 10. (a) Satisfaction of Full Pattern, where Outer Square is Pullback. (b) Annotated Model Satisfying the Proxy Pattern. (c) First Step in Satisfaction Checking.

4 Pattern-Based Model Completion

Patterns as defined above can be used in different scenarios: (i) to query how many instances of each pattern a model contains, or to analyse pattern instance interactions; (ii) for pattern extraction; (iii) to check whether a part of the model (maybe created by hand) conforms to a pattern; and (iv) to automatically complete a model according to a pattern. In this section we concentrate on the latter, giving the algorithm for pattern application for model completion and proving its correctness.

We start by showing how to apply the primary pattern, and then present how synchronization with the secondary patterns is achieved. Given a pattern $(VP, SP, Sec = \{(IP_s, SG_s)\}_{s \in S})$, the application of the primary pattern SP to an annotated model $M = (M_s \leftarrow M_c \rightarrow M_t)$ is as follows:

1. **Vocabulary Extension.** Add the vocabulary of the pattern to M_t if it was not added before (i.e. this is the first instance of the pattern).
2. **Role Annotation.** The user selects in M_s the elements playing some role in the pattern. Thus, a **RoleMap** node is created in M_c for each of these elements, associated with a node pm of type **PatternInstance**. pm is a new node for a new instance of the pattern, or an existing one, if extending a previous instance. Construct the morphisms from the modified annotation graph M_c to M_t and M_s . A new instance cannot be created if the number of existing instances would exceed the interval for the pattern root.
3. **Instance Extraction.** Construct a pattern graph PG by navigating from pm to the elements in M_c belonging to the defined instance of the pattern, and from these to elements in M_t and M_s along the c_s and c_t morphisms.

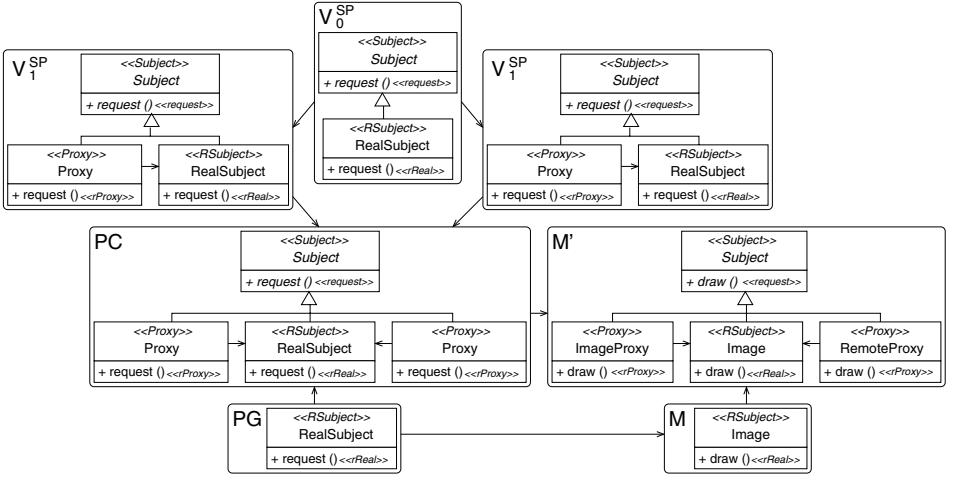


Fig. 12. Applying a Structural Pattern to an Annotated Model

- (a) **Let** V_i^{SP} be the current node; calculate the pullback object X of $V_i^{SP} \rightarrow PC \leftarrow PG$.
- (b) **If** $X \not\cong V_i^{SP}$ **then**
 - i. **If** $V_i^{SP} = \text{root}^{SP}$ **then** update the model M' according to the pushout shown in the center of Fig. 13.
 - ii. **Else let** V_j^{SP} be the predecessor of V_i^{SP} in Emb^{SP} and update the model M' according to the diagram to the right of Fig. 13.
 - iii. **Repeat**
 - **Let** V_m^{IP} be the child of V_l^{IP} in Emb^{IP} . Update the model (like in steps i and ii) for each instance V_n^{SP} s.t. $V_n^{SP} \leftarrow I_{nm} \rightarrow V_m^{IP}$, and whose pullback object X in $V_n^{SP} \rightarrow PC \leftarrow PG$ is not isomorphic to V_n^{SP} .
- Until** all descendants of $V_l^{IP} \in Emb^{IP}$ have been visited.

The procedure is incremental – one can update an existing instance – and supports heterogeneous synchronization, e.g. Fig. 7, where the structural pattern has two independent variable parts and nesting in the interaction pattern.

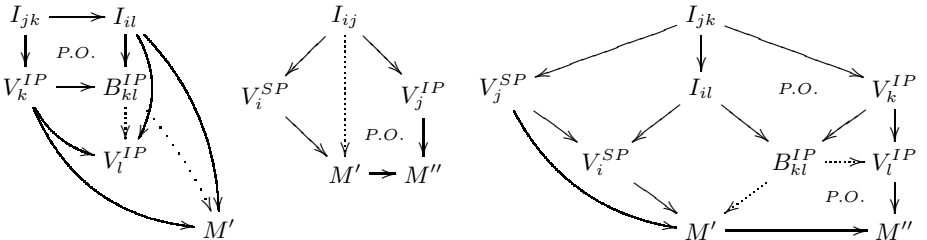


Fig. 13. Building B_{kl}^{IP} (left). First (center) and Second (right) Cases for Model Update.

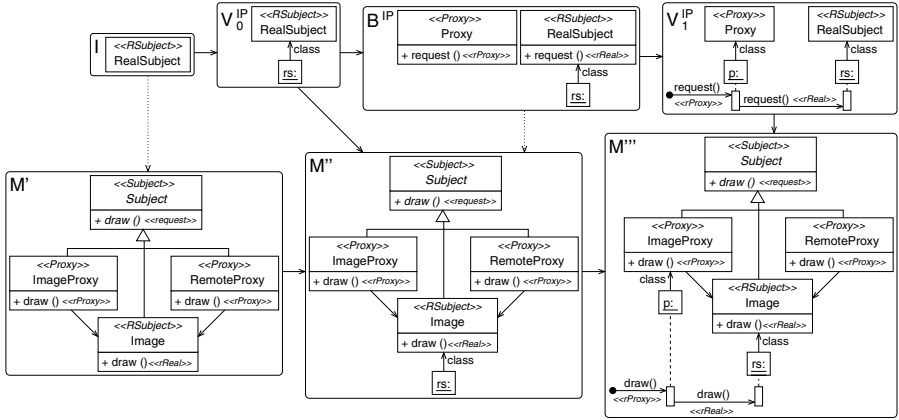


Fig. 14. Synchronization: Building the First Sequence Diagram

Example. For *Proxy*, Fig. 14 shows how the first sequence diagram is created starting from M' of Fig. 12. Note that, as there were two instantiations of V^{SP} , the procedure would follow by adding an additional sequence diagram. ■

Finally, it can be shown that, after applying a pattern according to the previous procedure, the resulting model satisfies such pattern according to Def. 6.

Proposition 1 (Application Correctness). *If model $M^{k'}$ is obtained from M by applying pattern SP according to the previous procedure, then $M^{k'} \models SP$.*

Proof Sketch. The nodes I_{ij} of SG are the pullbacks of $V_i^{SP} \rightarrow O \leftarrow V_j^{IP}$. When applying the primary pattern, the vocabulary model is added to M , and after the pushouts for the secondary pattern are made, yielding $M^{k'}$, we have $O \rightarrow M^{k'}$. This implies that all I_{ij} are the pullback objects of $V_i^{SP} \rightarrow M^{k'} \leftarrow V_j^{IP}$. The satisfaction $M^{k'} \models SP$ then follows as the application procedure takes care of the correctness of the replications w.r.t. the allowed variability intervals (step 4 of the application of the primary pattern). ■

5 Conclusions and Future Work

We have presented a formal approach to the specification of patterns, as well as procedures for checking whether a model satisfies a pattern and applying a pattern to a model for model completion. In the proposed formalization, patterns are annotated by roles in a vocabulary, support the synchronization of different types of diagrams, and allow the definition of variable parts, possibly nested. The proposal relies on a general meta-model for patterns, not necessarily based on UML.

Our approach presents several benefits w.r.t. existing ones. First, variability regions – with the possibility of nesting – are more flexible than current proposals, which annotate single elements with cardinalities [3] and for which it is more difficult to express that several elements have to vary together.

Second, our mechanism for pattern application considers synchronization of several diagrams. Third, we separate the roles from the pattern structure by a triple graph, without extending existing meta-models. This clean and non-intrusive solution facilitates the manipulation and querying of the vocabulary models, as well as the identification of pattern instances. Our formal treatment facilitates the derivation of rules for refactoring towards patterns.

We plan to investigate pattern conflicts and reason about pattern interaction effects, e.g. using graph constraints or critical pairs [2] in pattern application.

Acknowledgments. Work supported by the Spanish Ministry of Science and Innovation, projects METEORIC (TIN 2008-02081) and MODUWEB (TIN 2006-09678). We thank the referees for their insightful and detailed comments.

References

1. Dong, J., Yang, S., Zhang, K.: Visualizing design patterns in their applications and compositions. *IEEE Trans. Software Eng.* 33(7), 433–453 (2007)
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*. Springer, Heidelberg (2006)
3. France, R.B., Kim, D.-K., Ghosh, S., Song, E.: A UML-based pattern specification technique. *IEEE Trans. Software Eng.* 30(3), 193–206 (2004)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading (1994)
5. Guerra, E., de Lara, J.: Event-driven grammars: Relating abstract and concrete levels of visual languages. *Software and System Modeling* 6(3), 317–347 (2007)
6. Kampffmeyer, H., Zschaler, S.: Finding the pattern you need: The design pattern intent ontology. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS 2007. LNCS*, vol. 4735, pp. 211–225. Springer, Heidelberg (2007)
7. Kim, S.K., Carrington, D.: Using integrated metamodeling to define OO design patterns with Object-Z and UML. In: *APSEC*, pp. 257–264. *IEEE Computer Society*, Los Alamitos (2004)
8. Lauder, A., Kent, S.: Precise visual specification of design patterns. In: Jul, E. (ed.) *ECOOP 1998. LNCS*, vol. 1445, pp. 114–134. Springer, Heidelberg (1998)
9. Mac Lane, S.: *Categories for the Working Mathematician*, 2nd edn. *Graduate Texts in Mathematics*, vol. 5. Springer, Heidelberg (1998)
10. Mak, J.K.-H., Choy, C.S.-T., Lun, D.P.-K.: Precise modeling of design patterns in UML. In: *ICSE*, pp. 252–261. *IEEE Computer Society*, Los Alamitos (2004)
11. Niere, J., Schäfer, W., Wadsack, J.P., Wendehals, L., Welsh, J.: Towards pattern-based design recovery. In: *ICSE*, pp. 338–348. *ACM*, New York (2002)
12. Radermacher, A.: Support for design patterns through graph transformation tools. In: Münch, M., Nagl, M. (eds.) *AGTIVE 1999. LNCS*, vol. 1779, pp. 111–126. Springer, Heidelberg (2000)
13. Taibi, T., Ngo, D.C.L.: Formal specification of design pattern combination using BPSL. *Information and Software Technology* 45, 157–170 (2003)
14. Tourwé, T., Mens, T.: High-level transformations to support framework-based software development. In: *SET. ENTCS*, vol. 72-4 (2003)
15. van der Aalst, W., ter Hoefstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. *Distributed and Parallel Data Bases* 14(3), 5–51 (2003)
16. Zhao, C., Kong, J., Dong, J., Zhang, K.: Pattern-based design evolution using graph transformation. *J. Vis. Lang. Comput.* 18(4), 378–398 (2007)

Problem-Oriented Documentation of Design Patterns

Alexander Fülleborn¹, Klaus Meffert², and Maritta Heisel¹

¹ University Duisburg-Essen, Germany

`alexanderfuelleborn@hotmail.de`, `maritta.heisel@uni-duisburg-essen.de`

² Technical University Ilmenau, Germany

`pattern@klaus-meffert.com`

Abstract. In order to retrieve, select and apply design patterns in a tool-supported way, we suggest to construct and document a *problem-context pattern* that reflects the essence of the problems that the design pattern is meant to solve. In our approach, software engineers can choose examples of source code or UML models from the special domains that they are experts in. We present a method that enables software engineers to describe the transformation from a problem-bearing source model to an appropriate solution model. Afterwards, the *inverse* of that transformation is applied to the UML solution model of the existing design pattern, resulting in an abstract problem-context pattern. This pattern can then be stored together with the solution pattern in a pattern library. The method is illustrated by deriving a problem-context pattern for the Observer design pattern.

Keywords: Design patterns, problem derivation, model abstraction, cross-domain documentation, problem-context patterns, UML models, source code.

1 Introduction

Design patterns support software engineers in creating maintainable and extendable software. In order to select and apply design patterns, practitioners typically learn a pattern by reading a design pattern book or paper or by studying UML diagrams or source codes, respectively. From our point of view, this kind of documenting design patterns is lacking a machine-processable representation of the *problems* in their contexts to be solved by the design patterns. The pattern itself as the solution for the related problems, however, is represented by UML models, besides explanations in natural language and program code examples. This is why it can be instantiated as a solution model to the concrete problems. As there exists no corresponding problem model, it is difficult for software engineers to judge whether their concrete source code or design models, respectively, match the problems that the design pattern is meant to solve. They are forced to make a comparison that is based on two different formats in order to select an appropriate pattern. In addition, the contexts of the problems are expressed on

different abstraction levels. Due to the lack of cross-domain knowledge, there is a risk that software engineers assume their problem to be of a domain-specific type that does not match the essence of the problem addressed by a design pattern. Therefore they do not choose the solution provided by such a design pattern, even if it solved their problem.

We are interested in developing methods that help software engineers to retrieve, select and apply design patterns in a tool-supported way. It should be possible to apply these methods in the forward as well as in the re-engineering phase of the software product lifecycle. In this paper, we put special emphasis on *problem orientation* in documenting design patterns. We illustrate our ideas by a re-engineering example. Key of our approach is to enable software engineers in their role of pattern documentalists to use their daily, domain-specific work together with their knowledge about design patterns, in order to complete the documentation of these design patterns. We introduce the possibility for software engineers to start this completion process either with UML models or with source code, in order to obtain appropriate UML models that reflect the essence of the related problems in their contexts. The resulting artefacts that we call *problem-context patterns* are the basis for our overall methodology of semi-automated retrieval, selection and application of design patterns.

Our approach consists of a sophisticated way of documenting the situation before and after a design pattern is being applied. For the first part, this documenting is done by adding non-functional requirements as annotations to concrete, domain-specific source code or UML models that have design deficiencies, in order to document the problems in their contexts that the chosen design pattern solves. For the second part, we formally document the solved problems in a way that they can be compared to the situation before the chosen design pattern was applied. By way of that comparison, the *transformation* between the situation before and after applying the design pattern is made explicit. This transformation is then reused on the design pattern abstraction level in order to derive the reusable cross-domain representation of the situation before the chosen design pattern is being applied. To obtain the problem-context pattern, the *inverse* of the transformation is applied to the already existing UML model of the chosen design pattern that we call *solution pattern*.

The rest of the paper is organized as follows: in Section 2, we introduce our method for deriving problem-context patterns. We illustrate our method in Section 3 by using an example from the business domain of *human resources* and the Observer design pattern. Section 4 discusses other work in this area. Section 5 consists of conclusions of our findings and an outlook on future work.

2 A Method for Deriving Problem-Context Patterns

An overview of our method is given in Table 1.

Steps 1 to 4 are performed on the domain-specific level. In Step 1, software engineers choose a concrete, problem-bearing source code or UML model example that exists in their specific expert domain. In case software engineers choose a

Table 1. Method for deriving problem-context patterns

Step	Description
1	Choose a problem-bearing, domain-specific source code or UML model example
2	Annotate the chosen problem-bearing source code and UML models with <i>problem motives</i>
3	Perform transformations by applying design pattern under consideration to the chosen source code and UML models
4	Annotate the resulting source code and UML models with <i>solution motives</i>
5	Annotate the UML solution model of the cross-domain design pattern with the same solution motives as in Step 4
6	Perform <i>inverse</i> design pattern transformations to the existing design pattern UML solution models that are annotated according to Step 5

problem-bearing source code for which no corresponding UML model exists yet (the typical re-engineering case), the latter must be created, as it is needed in the later steps of the method. In case software engineers choose a problem-bearing UML model as a starting point for the example (the typical forward engineering case), they do not need to have corresponding source code, as Steps 5 and 6 are only based on the UML models. The chosen example must fit to the abstract, natural-language problem description of the design pattern for which they want to complete the documentation. We assume that software engineers are familiar with the design pattern, for which they intend to complete the documentation. By using specific examples from expert domains, the procedure of completing a design pattern documentation is facilitated. The advantage of this approach is that it is integrated into the usual work of software engineers. The effort needed to complete the documentation of design patterns is minimized, because software engineers can derive them by doing their daily work of modeling, coding and improving designs.

In Step 2, annotations to the problem-bearing source code and to the UML models are added manually. We call these annotations *problem motives*. A method for deriving problem motives on the source code level can be found in [5]. On the modeling level, we assign these problem motives to UML model elements. They represent the non-functional requirements that need to be fulfilled by the source code and UML models after the design pattern was applied.

In Step 3, the problem-bearing source code and UML models are transformed step by step to re-engineered new source code and UML models, according to the knowledge embodied in the design pattern. In case new elements are added or existing elements are changed, annotations are manually added to these elements in Step 4. We call these annotations *solution motives*. They directly correspond to the problem motives in the problem-bearing source code and UML models. It is also possible that there does not exist a problem motive that corresponds to a solution motive. In this case, the solution provides an additional advantage. It is also possible that a problem motive without any corresponding solution motive exists. This indicates a non-optimal solution of the problem.

The preceding steps all take place on the domain-specific level with its special vocabulary and semantics, and with limited reuse potential. In the final two steps of our method, software engineers operate on the generic, cross-domain design pattern level. Here, the main purpose is to *complete the documentation of the generic design pattern* that can be reused across domains. The goal is to find an appropriate cross-domain UML model for the situation before transformations are being performed.

In Step 5, software engineers annotate the UML solution model of the chosen design pattern that has been applied on the domain-specific level before. For this purpose, they reuse the knowledge they already gained in the domain-specific scenario: they take the same solution motives they also used for the domain-specific UML solution model and add them as annotations to the cross-domain UML solution models. Then, they also reuse the transformations, but in this case, they apply these transformations *inversely* to the generic, cross-domain UML solution models of this design pattern. Thus, they *derive the cross-domain problems in their cross-domain contexts* that fit to the design pattern. Finally, the obtained problem-context pattern can be stored together with the solution pattern in a design pattern library. It then can be retrieved and selected according to the method described in [2].

3 Case Study *Salary Statement Application*

To illustrate our method, we present a case study from the *human resources* business domain. It is about an existing software application for creating *salary statements* of monthly employee salaries. This application needs to be re-engineered, as it has non-functional deficiencies that can be removed by applying the Observer design pattern. The software engineers who perform this re-engineering task know the pattern well and are able to apply it to the existing code. No UML models exist, only source code is available to them. Hence, this source code represents the problem-bearing, domain-specific source code example according to Step 1 of our method. Besides this domain-specific, pure re-engineering task, the software engineers are also asked to complete the Observer design pattern documentation. Hence, they are asked to perform the remaining steps of our method. In Sections 3.1 and 3.2, we present the results of Steps 1 to 4 of our method on the source code level. We chose the object-oriented Java language for demonstrating purposes as Java is widely spread and state-of-the-art. In Sections 3.3 and 3.4, we illustrate the results of Steps 1 to 4 on the UML model level. In Section 3.5, the results of Steps 5 and 6 are presented.

3.1 Salary Statement Application: Annotated Problem-Bearing, Domain-Specific Source Code

According to Steps 1 and 2 of our method, the following source code from the salary statement application has been chosen and annotated:

```

010    public class SalaryStatementApplication {
020        public static void main(String[] args) {
030            EmployeeDetail employeeDet = new EmployeeDetail();
040    acd    EmployeeSelector employeeSel = new EmployeeSelector(
        employeeDet);
050        // User Input in a GUI field. Sent by an event handler
060        employeeSel.updateEmployeeID("D026143");
070    }
080    }

100    public class EmployeeSelector {
110    a    private EmployeeDetail employeeDet;
120        private PersNo    selectedEmployee;

130    acd    public EmployeeSelector(EmployeeDetail employeeDet) \{
140    ac        this.employeeDet = employeeDet;
150        \}

180        // Method will be called after user entered employee ID
190        public void updateEmployeeID(PersNo ID) {
200            selectedEmployee = ID;
210    a        employeeDet.changeEmployeeID(ID);
220        }
230    }

300    public class EmployeeDetail {
340        public void changeEmployeeID(PersNo ID) {
350            // Read employee with given ID from database
360            PersName name = ...
370            // Read salary data for employee from database
380            Salarybracket salarybracket = readSalarybracket(ID);
390            // Update the display of the graphical user interface
400            ...
410        }

420        public Salarybracket readSalarybracket(PersNo ID) {
430            // Read salary data with given ID from database
440            Salarybracket result = ...
450            return result;
470        }
480    }

```

Listing 1. Salary statement application: annotated problem-bearing, domain-specific source code

The source code given in Listing 1 represents an application with three classes. The first class, *SalaryStatementApplication*, is responsible for starting the application. It creates two graphical elements. The first element is represented by the class *EmployeeSelector* and implements a selection list with basic master data related to employees. When users have chosen an entry from this list, the detail information that is needed in this context can be read and displayed. In our example, the detail information is represented by the second element, the class *EmployeeDetail*, and is displayed as a screen area with detail data such as *Name* and *Salarybracket* (which is the technical term for a salary group) of the employee, that has been selected by the employee selector. As already mentioned above, the Observer design pattern should be applied to this source code. According to the terminology used in Observer, the class *EmployeeSelector* corresponds to the *Subject* class, and *EmployeeDetail* is the observer class to the subject. The basic non-functional deficiency of this source code is the fact that

Table 2. Salary statement application: problem motives

Problem motive identifier	Description
a	Observer class does not implement any interface
c	Observer class can only be registered at one given point in time
d	No unified registering mechanism existing

the classes *EmployeeSelector* and *EmployeeDetail* are too tightly coupled, which is not desired. We can further differentiate this fundamental problem for the given context. Firstly, the subject class *EmployeeSelector* can only be reused in a limited way, as it has an attribute of type *EmployeeDetail*. A common interface provided by the subject class to share data or functionality with observers is missing. This is a structural coupling. Secondly, objects of the class *EmployeeDetail* can only be registered at one point in time, namely when objects of class *EmployeeSelector* are created. Thus, the objects of the classes *EmployeeDetail* and *EmployeeSelector* are also coupled in time. Thirdly, only objects of the class *EmployeeDetail* can be registered with the class *EmployeeSelector*. A unified registering mechanism for objects of arbitrary classes is missing, despite of the fact that they are not present at the moment. According to Step 2 of our method, the described detail problems are added to the source code as *problem motives* by adding a character as an identifier to each detail problem. In Table 2, the used problem motives are listed.

Table 2 contains several aspects of the *too tightly coupled* problem, which are addressed by the Observer design pattern. Note that identifier *b* does not appear in the problem-bearing source code. The reason is that it has been reserved for a solution motive in the post-transformation source code, that we will discuss later. Performing Step 2 of our method, we place problem motive identifiers after the line numbers in the problem-bearing source code. As already mentioned before, a method exists for annotating source code which can be found in [5]. For example, problem motive *a* has been declared for all lines of code that are affected by the fact that class *EmployeeDetail* does not implement any interface. These are all lines of code where this class type is used. Applying the Observer design pattern removes the deficiencies that are mirrored by the problem motives. In the following, the Observer design pattern is applied to the given source code according to Steps 3 and 4 of our method.

3.2 Salary Statement Application: Performing Transformations by Applying the Observer Design Pattern

According to Step 3 of our method, every change of the problem-bearing source code that is caused by applying the Observer design pattern is reflected by a transformation. Performing method Step 4, each of these transformations has an explanation, given by an annotated solution motive. This solution motive is the complement to the problem motive and is directly related to the latter. A problem motive that has been replaced by a solution motive indicates that the

Table 3. Salary statement application: solution motives

Solution motive identifier	Description
a	Treat observer classes equally
b	Possibility to register any number of observers
c	Possibility to register at any time
d	Unified registering mechanism
e	Notify all observers

given problem is solved. A solution motive without any corresponding problem motive reflects a positive property that has been added without any problem relationship. This is true for solution motive *b*, that we already mentioned, and it is true for solution motive *e*. An identifier of a solution motive that equals an identifier of the problem motive indicates that it solves the corresponding problem. An overview of the solution motives used in this example is given in Table 3.

The transformations used to apply the Observer pattern are all related to these solution motives. For each transformation, we give the resulting source code including the solution motives. If the transformation is a deletion, only the deleted source code is shown. For better traceability, the line numbers from Listing 1 are given, too. Equal line numbers in the post-transformation source code indicate a change, new line numbers indicate an insertion.

Transformation T1: declaring the subject without relation to any observer

In this transformation, the observer and subject class are decoupled from each other by deleting the static attribute for observer from the subject class.

```

109  /**@motive a(1): Treat observer classes equally*/
110  a ——— private EmployeeDetail employeeDet;

```

Transformation T2: constructing the subject without relation to any observer

This transformation causes the change of two dependent parts within the source code, namely of a constructor declaration and the creation of objects of class *EmployeeSelector* by this constructor. The changed constructor looks as follows:

```

127  /**@motive a(2): Treat observer classes equally*/
128  /**@motive c(1): Possibility to register at any time*/
129  /**@motive d(1): Unified registering mechanism*/
130  public EmployeeSelector(){
137  /**@motiv a(3): Treat observer classes equally*/
138  /**@motive c(2): Possibility to register at any time*/
139  /**@motive d(2): Unified registering mechanism*/
140  ——— this.employeeDet = employeeDet;

```

As a result, the constructor call must also be adjusted:

```
037    /**@motive a(4): Treat observer classes equally*/
038    /**@motive c(3): Possibility to register at any time*/
039    /**@motive d(3): Unified registering mechanism*/
040    EmployeeSelector employeeSel = new EmployeeSelector();
```

As every solution motive can appear more than once, we use a serial number per motive, which we put into brackets behind each solution motive.

Transformation T3: introducing the base class for observer

Part of the core concept of the Observer design pattern is the usage of an abstract base class for observer classes. In this context, we call this class *Observer*. Firstly, we introduce this base class:

```
899 /**@motive a(5): Treat observer classes equally*/
900 public abstract class Observer {
909    /**@motive e(1): Notify all observers*/
910    public abstract void update(Object state);
920 }
```

Next, the specialized observer class *EmployeeDetail* can inherit from this class:

```
299 /**@motive a(6): Treat observer classes equally*/
300 public class EmployeeDetail extends Observer {
```

Furthermore, the abstract method needs to be implemented:

```
309    /**@motive e(2): Notify all observers*/
310    public void update(Object state){
320        changeEmployeeID((PersNo)state);
330    }
```

Transformation T4: introducing a universal registration mechanism within the subject

Now, we can introduce a registration method with a variable for storing observer references:

```
109    /**@motive b(1): Possibility to register any number of observers*/
110    private List<Observer> observers = new Vector();

147    /**@motive a(7): Treat observer classes equally*/
148    /**@motive b(2): Possibility to register any number of observers*/
149    /**@motive c(4): Possibility to register at any time*/
150    public void register(Observer a_observer){
160        observers.add(a_observer);
170    }
```

The newly introduced solution motive with the identifier *b* has no corresponding problem motive in the problem-bearing source code. This means that it is an additional advantage of the solution that was not seen as a problem before.

Transformation T5: unified notification of all observers by the subject

The way the subject notifies its observers can be adjusted, too. Firstly, we introduce a new method for notifications:

```

222  /**@motive e(3): Notify all observers*/
223  public void notify(){
224      /**@motive a(8): Treat observer classes equally*/
225      /**@motive e(4): Notify all observers*/
226      for(Observer observer:observers){
227          observer.update(selectedEmployee);
228      }
229  }

```

Now the method call must be placed at a suitable location:

```

209  /**@motive e(5): Notify all observers*/
210  notify();

```

The remaining reference to *EmployeeDetail* in line 030 of the problem-bearing source code can be replaced by a reference to the class *Observer*. However, this is not needed here. To give readers of this paper a complete overview about the resulting post-transformation source code, we provide it as a listing in an appendix at the end of the long version of this paper.¹

3.3 Salary Statement Application: Annotated Problem-Bearing, Domain-Specific UML Model

Until now we performed Steps 1 to 4 of our method on the source code level. Next, we repeat these steps on the UML model level, because Steps 5 and 6 are based on the UML models. We start with Steps 1 and 2. First, we create domain-specific UML models of the annotated problem-bearing source code. Extracting the model can be automated to a certain extent. Next, the problem motives of the source code are added to those model elements that cause the problem, according to Step 2 of our method. While taking over the problem motives, some information gets lost, as not all of the problem motives can be taken over to the UML model level. The reason is that problem motives in the source code can also relate to single program statements that do not appear on the UML model level. However, this loss of information can be reduced by using UML comments, which contain these program statements with assigned problem motives in order to give software engineers some guidance in implementing the details. The resulting UML problem-context model, which is needed later in Step 6 to derive an appropriate problem-context pattern, is shown in Figure 1.

In this problem-context model, the problem motive identifiers of the annotated problem-bearing source code are assigned to the relevant elements using the tilde symbol. For example, annotation *motives: a* is assigned to the problem-causing attribute *employeeDet* in the class *EmployeeSelector*. As stated in Table 2, this identifier means *Observer class does not implement any interface*, which refines the basic problem that class *EmployeeSelector* and class *EmployeeDetail* are too tightly coupled. This problem is expressed by a static attribute that limits the reuse of class *EmployeeSelector*. For the sake of readability, we only look at

¹ Available at <http://swe.uni-duisburg-essen.de/techreports/Fase09Longversion.pdf>.

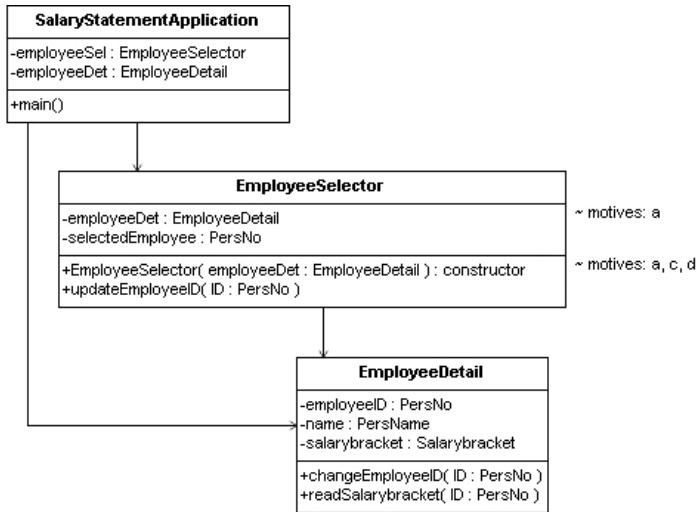


Fig. 1. Salary statement application: annotated UML model (problem-context model)

the structural aspects in our example. In our research work, we also applied the described steps to sequence diagrams, which works well, but does not provide additional information and does not necessitate any extension of the method.

3.4 Salary Statement Application: Annotated Resulting Domain-Specific UML Model

In this section, we repeat Step 2 of our method on the modeling level. The resulting domain-specific UML solution model is directly generated from the re-engineered source code. Then, software engineers analyze the differences between the pre- and post-transformation UML models by considering the differences in the pre- and post-transformation source codes, and annotate the models with comments about the solved problem. The resulting annotated class diagram is shown in Figure 2.

Note that there are comments assigned to several model elements. These model elements are exactly the classes or relationships that have been changed from the problem-context model. The comments log the type and the reason for the change. The type of changes, namely *adding* or *deleting* is described by an appropriate keyword. The reason for the change is described by adding solution motives, which also establish relationships to the underlying problem motives in the problem-context model. For example, a comment is added to the class *EmployeeSelector* that states that the attribute *employeeDet* is deleted due to solution motive *a*. In the problem-context model, this attribute still exists and is annotated with the corresponding problem motive. By using a numbering within the used motives, software engineers can better reflect the order of the single transformation steps and the involved model elements.

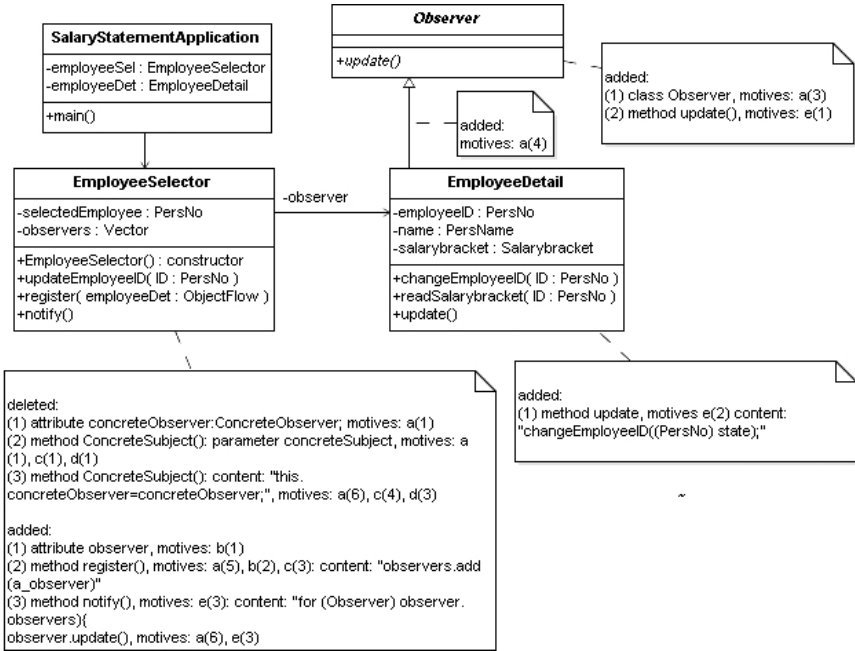


Fig. 2. Salary statement application: annotated resulting post-transformation UML solution model

3.5 Deriving a Fitting Problem-Context Pattern (Cross-Domain)

Up to this point in the procedure, all activities take place on the domain-specific level. From now on, the cross-domain level is considered by following Steps 5 and 6 of our method. Here, the main purpose is to *complete* the generic, cross-domain Observer design pattern by finding an appropriate cross-domain UML model, a problem-context pattern for the pre-transformation situation. First, the existing UML solution model of the Observer design pattern is annotated according to Step 5 of our method. For this purpose, the solution motives that have been used to annotate the domain-specific *salary statement* UML solution model are taken as a starting point. Thus, software engineers extend the Observer UML solution models with information about the problems they solve and with information about the way how they solve them. As in the domain-specific example, for the sake of readability and simplicity, we use a variant of the Observer design pattern. In this variant, no explicit subject superclass exists, and observers do not ask separately for state changes. Instead, the subject provides the observers proactively with information about state changes. While adding information about the performed transformations, the transformed model elements are also abstracted. The result is illustrated in Figure 3.

Note that annotations have also been changed. They are adapted to the pattern in the sense of an abstraction. For example, the model element *attribute*

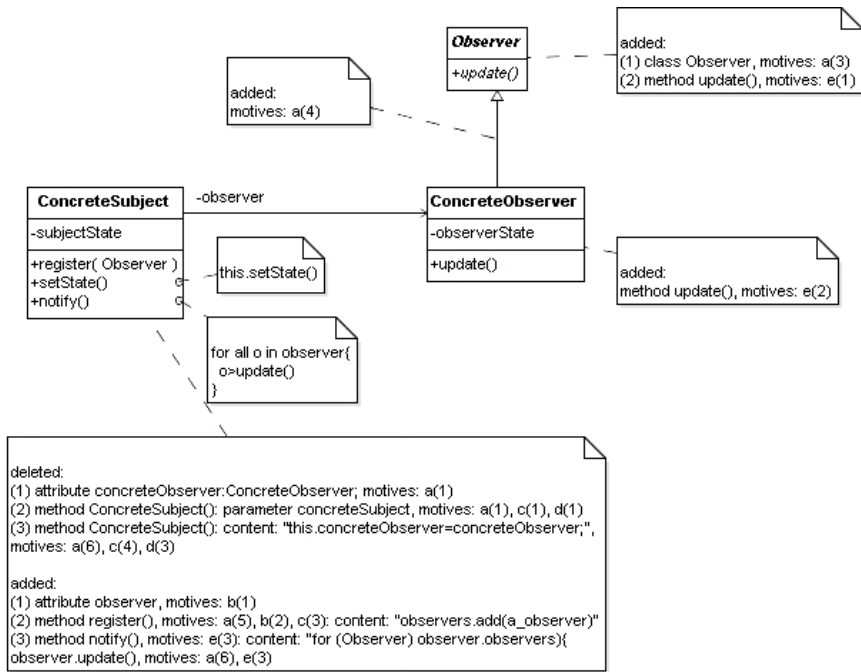


Fig. 3. Annotated UML solution model of the Observer design pattern (variant) according to method Step 5

employeeDet of class *ConcreteSubject* that can be found under the *deleted* annotation has been renamed to *attribute concreteObserver*. This annotated solution pattern is the starting point for the next step of *inverse transformations*. In Step 6 of our method, a suitable Observer problem-context pattern is derived by applying inverse transformations. All transformation steps, that have been performed before on the domain-specific level, started from problem-bearing models (problem-context models) and resulted in solution models. On the cross-domain level, solution models exist already, as they are described in the literature [3]. Thus, in order to obtain appropriate problem models for the Observer design pattern, the knowledge about the way to transform the problem-bearing models into solution models on the domain-specific level is reused, but in the opposite direction. For example, in Figure 3 the class *Observer* is annotated with:

added:
 (1) class *Observer*, motives: a(3)
 (2) method *update()*, motives: e(1)

This annotation is based on the transformations *add class Observer* and *add method update()* to class *Observer*. Thus, the appropriate inverse transformations are:

delete method update() from class Observer
delete class Observer

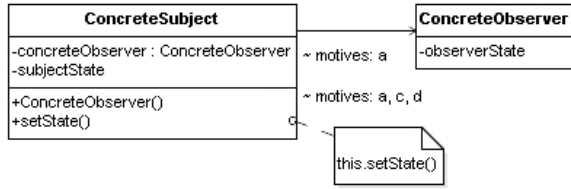


Fig. 4. Derived problem-context pattern of the Observer design pattern according to method Step 6

Besides performing inverse transformations, also problem motives are needed in the derived problem-context pattern, which establish the link to the solution motives in the solution pattern. These problem motives are derived from the domain-specific problem-context model. The derived *problem-context pattern* is illustrated as a result of the described actions in Figure 4.

The illustrated problem-context pattern abstractly describes a possible starting point for applying the Observer design pattern, using the same means of expression as the solution, namely UML. Together with the UML solution models, the *solution pattern*, it forms the Observer design pattern and can be stored in a pattern library. The problem-context pattern is the access key for a semi-automated pattern retrieval and selection method. Such a method is described in [2].

4 Related Work

There has only been little work on documenting the problem essence of design patterns in an appropriate way. Different from our approach of expressing the problem essence as UML models, other contributors take *UML meta models* as the basis for their methods. Mili and El-Boussaidi [6] use transformation meta models, besides problem and solution meta models, to describe appropriate design pattern problem models and the way they are transformed to design pattern solution models. Differently from our method, their problem meta models do not contain any non-functional requirement or problem description that would be comparable to our problem motives.

Kim and El Khawand [4] propose to rigorously specify the problem domain of design patterns. In contrast to Mili and El-Boussaidi [6] and our approach, they do not describe the problem-bearing model *before* a design pattern is applied to it. Moreover, they focus on the functional aspects of design patterns, not on non-functional aspects. The authors are mainly interested in developing tool support for checking whether existing UML models conform to known design patterns.

The work of Fanjiang and Kuo [1] introduces the concept of design-pattern-specific *transformation rule schemata* to be used as an additional design pattern documentation. The transformation steps, which are described in natural language, are similar to ours and are helpful in applying design patterns. However, as the authors do not intend to support software engineers in their role

of documentalists, they do not give guidance on deriving transformation rule schemata.

The work of O'Cinnéide and Nixon [7] aims at applying design patterns to existing legacy code in a highly automated way. They target code refactorings. Their approach is based on a semi-formal description of the transformations themselves, needed in order to make the changes in the code happen. In contrast to our method, they describe precisely the transformation itself and under which pre- and postconditions it can successfully be applied. In our work, we illustrate the situation before and after the transformation. To a certain extent, the described preconditions of the transformations can be compared with our problem context as it outlines the situation before the design pattern is applied. The advantage of our approach, however, is that we *explicitly* describe the non-functional deficiencies by using annotations in the source code of the sub-optimal situation.

5 Summary and Future Work

In this paper, we have presented a method for the problem-oriented documentation of design patterns that consists of 6 steps. The results of Steps 1 to 2 are an annotated problem-bearing source code and the corresponding UML models, stemming from the practical work of software engineers. By applying the chosen design pattern and by adding *solution motives* to the resulting source code and corresponding UML models, the outcome of Steps 3 and 4 are the transformed and annotated solution source code and UML models. In Step 5, the same solution motives are used to obtain annotated UML solution models of the cross-domain design pattern. Finally, inverses of transformations according to Step 3 are applied to these UML solution models, which result in a *problem-context pattern* that fits to the chosen design pattern.

The use of expert domain, real world examples on the source code or on the modeling level in order to derive the problem-context pattern on the cross-domain level is novel and makes this approach especially useful and efficient in re-engineering as well as in forward engineering projects. In addition to that reuse of domain-specific knowledge on the abstract level, reusing the already documented UML models of the design pattern solution part to derive the abstract problem-context pattern is efficient.

To demonstrate how our method works, we used Observer, a behavioural design pattern. In our research work, we also applied it to creational and structural patterns, which works well, too. Besides the scenario of completing *existing* design patterns that are known from the literature, it also seems promising to support the creation of *new* patterns in this way. This aspect is part of our future work. Furthermore, we are working on the development of a *problem statement language* that helps to reuse generic, standardized problems by making use of the ideas provided by Willms et al. [8]. Another subject area we want to address is the cross-domain reuse of functional requirements as opposed to the typical non-functional requirements that are addressed by design patterns.

References

1. Fanjiang, Y.-Y., Kuo, J.-Y.: A pattern-based model transformation approach to enhance design quality. In: Cheng, H.D., Chen, S.D., Lin, R.Y. (eds.) JCIS 2006, Proceedings of the 2006 Joint Conference on Information Sciences. Atlantis Press (2006)
2. Fülleborn, A., Heisel, M.: Methods to create and use cross-domain analysis patterns. In: Zdun, U., Hvatum, L. (eds.) EuroPLoP 2006, Proceedings of the 11th European Conference on Pattern Languages of Programs, pp. 427–442. Universitätsverlag Konstanz (2007)
3. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design patterns: Abstraction and reuse of object-oriented design. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 406–431. Springer, Heidelberg (1993)
4. Kim, D.-K., Khawand, C.E.: An approach to precisely specifying the problem domain of design patterns. *Journal of Visual Languages and Computing* 18(6), 560–591 (2007)
5. Meffert, K., Philippow, I.: Supporting program comprehension for refactoring-operations with annotations. In: Fujita, H., Mejri, M. (eds.) Proceedings of the fifth SoMeT 2006: New Trends in Software Methodologies, Tools and Techniques, vol. 147, pp. 48–67. IOS Press, Amsterdam (2006)
6. Mili, H., El-Boussaidi, G.: Representing and applying design patterns: What is the problem? In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 186–200. Springer, Heidelberg (2005)
7. O’Cinnéide, M., Nixon, P.: A methodology for the automated introduction of design patterns. In: ICSM 1999: Proceedings of the IEEE International Conference on Software Maintenance, p. 463. IEEE Computer Society Press, Washington (1999)
8. Willms, J., Wentzlaff, I., Specker, M.: Kreativität in der informatik: Anwendungsbeispiele der innovativen prinzipien aus triz. In: Informatik 2000, Neue Horizonte im neuen Jahrhundert, 30. Jahrestagung der Gesellschaft für Informatik. Springer, Heidelberg (2000)

Certification of Smart-Card Applications in Common Criteria

Proving Representation Correspondences

Iman Narasamdya¹ and Michaël Périn^{2,*}

¹ FBK-Irst, Italy

narasamdya@fbk.eu

² Verimag - UJF, France

Michael.Perin@imag.fr

Abstract. We present a method for proving representation correspondences in the Common Criteria (CC) certification of smart-card applications. For security policy enforcement, the CC defines a chain of requirements: a security policy model (SPM), a functional specification (FSP), and a target-of-evaluation design (TDS). In our approach to the CC certification, these requirements are models of applications that can have different representations. A representation correspondence (RCR) describes a correlation between the representations of two adjacent requirements. One task in the CC certification is to demonstrate formal proofs of RCRs. We first develop a modelling framework by which the representations of SPM, FSP and TDS can be described uniformly as models of an application. We then define RCRs as mutual simulations between two application models over sets of observable events and variables. We describe a proof technique for proving RCRs and providing certificates about them based on assertions relating two models at specific locations. We show how RCRs can help us prove property preservation from the SPM to the FSP and the TDS.

1 Introduction

We describe in this paper our work on developing a method for formal certification of smart-card applications in the framework of Common Criteria (CC) [1]. This work is part of an industrial project called EDEN2.¹ The CC is an international standard for the evaluation of security related systems. It guarantees that a target of evaluation (TOE), or a system, enforces security policies by means of an assurance architecture. For assurances in the development process, this architecture consists of a chain of requirements starting from the model of the policies at the start of the chain, to the low-level design and the implementation of the system at the end of the chain.

At the highest level of the CC certification, which is called evaluation assurance level 7 (or EAL7), the following chain of requirements are needed in the assurance architecture: (1) a formal security model (SPM), (2) a formal functional specification

* This research has been supported by funding from RNTL EDEN project.

¹ Research and industrial partners include Verimag, CEA, Gemalto, and Trusted-Logic; see <http://www.eden-rntl.org>

of security functions (FSP), and (3) a TOE design (TDS). The SPM models the policy independently of the implementation, the FSP describes input-output relationships of security functions, and the TDS is a low-level design that is close to the implementation. A representation correspondence (RCR) demonstrates the correlation between each two adjacent requirements in the chain. The CC EAL7 certification consists of proving that the SPM, the TDS, and the FSP satisfy the security policies, and providing certificates about this satisfaction. In addition, the CC EAL7 also requires formal proofs of RCRs between the SPM and the FSP, and between the FSP and the TDS.

In this paper we are concerned with proving RCRs and providing certificates about them. We present a method for proving RCRs in the context of smart-card applications. First, we develop a framework for modelling smart-card applications such that the formal models capture the operations of the applications, in particular our model allows one to reason about card tears (or power loss) and transaction mechanism that are present in smart-card applications. In this framework, a model of an application consists of a set of command procedures (or simply command). Each command is presented by two transition graphs (or control-flow graphs), one describes the normal behavior of the command and the other describes what the command has to perform when a card tear occurs. The FSP and the TDS are essentially models of an application. In EDEN2, the SPM consists of two entities: one entity is a model of the application and the other is a set of assertions (or formulas) in some logic such that the assertions describe security properties. In the sequel, we refer to the former entity when we speak about SPM. Card readers communicate with a smart-card application by sending a sequence of commands. We model this interaction with a main procedure that takes as the only input a sequence of commands, and for each command, the procedure calls the corresponding command procedure in the application. The semantics of an application is then characterized by the set of the main procedure's runs.

We define RCRs between two application models as bisimulation equivalence consisting of mutual simulations between the models over observable events and variables. To this end, given two models S and T of an application, we associate with S and T the same set of observable events and for each event we associate a mapping between observable variables. Intuitively, we say that there is an RCR between S and T if for every run of S , there is a run of T on the same input, and vice versa, such that (1) both runs exhibit the same sequence of observable events, and (2) for each two equal events, the values of corresponding observable variables coincide. Having a unified model for smart-card applications allows us to have only a single definition of RCRs such that the definition is applicable for RCRs between the SPM and the FSP, and between the FSP and the TDS. Furthermore, we will show that our definition of RCR helps us prove *property preservation* from one model to the other. That is, as required by the CC EAL7 certification, the RCRs must guarantee that all security properties satisfied by the SPM are satisfied by the FSP and the TDS.

We develop a proof technique for proving RCRs. We prove RCRs between S and T by proving the RCR between each corresponding commands in S and T . We apply a theory of inter-program properties described in [16] to proving RCRs. Inter-program properties are properties relating two programs. RCRs are essentially inter-program properties. We prove RCRs by using assertions that describe data abstraction

and control mapping between the transition graphs of the corresponding commands. The theory also provides a notion of certificate about inter-program properties. Such a certificate is essential to the CC EAL7 certification.

Proving RCRs are challenging due to nontrivial data abstractions between application models and due to language features in which the models are written. Consider a command `checkPIN` used to authenticate users by checking an input PIN against the PIN stored on the card. The security policy does not require the PIN to be in some specific format. Thus, in the SPM the PIN can simply be a natural number. For security, the command uses variables `trial` as a trial-remaining counter. If the input PIN does not match the stored PIN, then `trial` is decremented, and if it gets 0, then the PIN is blocked. In the FSP, developers usually take defensive measures. The PIN in the FSP is now an array of natural numbers, and prior to checking the input PIN, the variable `trial` must be decremented. We then have the following excerpts of over-simplified `checkPIN`, the SPM and the FSP are on the lefthand and righthand, respectively:

<pre> if (pin $\neq$ input) { trial := trial - 1; return fail; } </pre>	<pre> trial := trial - 1; while (i < length) { if (pin[i] $\neq$ input[i]) return fail; } </pre>

If we associate an event with every update of `trial`, then in the SPM this event occurs at the end of command execution, but in the FSP it occurs at the beginning. Thus, we may end up with different sequences of observable events. This poses some difficulties in determining observable events in RCRs. Note that in the SPM and the FSP above, the data abstraction introduces a loop in the FSP. To prove that for every run of the SPM there is a “corresponding” run of the FSP, one has to prove that the loop will not yield non-terminating run. We will show later that in the presence of transaction mechanism, we sometimes have to relax the definition of RCR. That is, we only require that for every run of the TDS, there is a corresponding run of the FSP.

In summary the contributions of this paper is a method for proving representation correspondences as a part of the CC EAL7 certification of smart-card applications.

The outline of this paper is the following. We first discuss our framework for formally modelling smart-card applications. We then develop a notion of representation correspondence based on this framework. Afterward we describe briefly the theory of inter-program properties. Then, we discuss our proof technique for proving RCRs based on the theory. We then show how RCRs allow us to preserve property in the chain of the CC requirements. Finally, we discuss some related work and conclude this paper.

2 Formal Models and Representation Correspondences

2.1 Transition Graphs and Computation Sequences

A smart-card application is a program consisting of $m+1$ procedures: $main, c_1, \dots, c_m$, where $main$ is the main procedure and $c_1, \dots, c_m$ are command procedures. In the sequel, command procedures are often called *commands*. Each procedure P consists of a finite set of *program points* and is presented as two disjoint *transition graphs* (or

program-point flow graphs) $\mathbf{G}_P^n$ and $\mathbf{G}_P^a$. A transition graph is a finite directed graph whose nodes are program points. Each edge of a transition graph is labelled with a guard, an assignment instruction, a goto instruction (or a skip instruction), or a procedure call. The transition graph $\mathbf{G}_P^n$ describes the normal behavior of P , while the transition graph $\mathbf{G}_P^a$ describes what the application has to do when a card tear occurs during the execution of P .

We assume that every transition graph $\mathbf{G}_P$ has a unique entry point, denoted by $\text{entry}(\mathbf{G}_P)$ and a unique exit point, denoted by $\text{exit}(\mathbf{G}_P)$. As such, every procedure P has a unique entry point $\text{entry}(P) = \text{entry}(\mathbf{G}_P^n)$, and two exit points, *normal exit point* $\text{exit}_n(P) = \text{exit}(\mathbf{G}_P^n)$ and *abrupt exit point* $\text{exit}_a(P) = \text{exit}(\mathbf{G}_P^a)$.

The main procedure takes as input a sequence of input commands. In turn, the procedure reads each input command of the form $(C, \bar{v})$, where C is the command name and $\bar{v}$ are the input values for C . For each input command $(C, \bar{v})$, the main procedure calls the corresponding command C on input $\bar{v}$, or call $C(\bar{v})$.

We introduce a restriction on command procedures, that is, for every command procedure P , the graphs $\mathbf{G}_P^n$ and $\mathbf{G}_P^a$ do not contain edges labelled with procedure calls. Similarly, the graph $\mathbf{G}_{main}^a$ does not contain such edges. This restriction does not limit the applications that can be modelled in our framework. Procedures called by command procedures in smart-card applications are usually not recursive and thus can be inlined. For technical reason, we assume that, for every command procedure, $\mathbf{G}_{main}^n$ contains an edge labelled with a call to the procedure.

We describe the run-time behavior of an application as sequences of configurations. A *configuration* of a run is a pair (p, σ) where p is a program point and σ is a *state* mapping variables to values. Given a procedure P , a configuration (p, σ) is called an *entry configuration* for P if p is an entry point of P , a *normal exit configuration* for P if p is a normal exit point of P , and an *abrupt exit configuration* for P if p is an abrupt exit point of P .

The semantics of an application is defined as a transition relation with transitions of the form $(p_1, \sigma_1) \xrightarrow{l} (p_2, \sigma_2)$, where (p_1, σ_1) and (p_2, σ_2) are configurations and l is a transition label. Transitions are of the following kinds:

- Intra-graph transition, where the pair (p_1, p_2) is an edge of a transition graph, l is the label of the edge such that l is not a procedure call.
- Call and return transitions, where l is a procedure call and a special label *ret*, respectively.
- Abrupt transition, where p_1 is in $\mathbf{G}_P^n$, p_2 is $\text{entry}(\mathbf{G}_P^a)$, l is a special label *ab*, and $\sigma_1 = \sigma_2$.

We allow labels of transitions (or edges of transition graphs) to be associated with events, which means that the transitions emit the events. We will use a special event variable ε to store emitted events. That is, if a transition emits an event E , then it is the same as an assignment of E to ε . Details of transition relations are in [12].

We use the following assumptions for transition relations. First, for every procedure P , every point p in $\mathbf{G}_P^n$, and every state σ , there is a transition $(p, \sigma) \xrightarrow{l} (\text{entry}(\mathbf{G}_P^a), \sigma)$. That is, a card tear can occur non-deterministically. Second, there is no transition from an exit configuration (p, σ) , where $p = \text{exit}_a(P)$ for every procedure P , or $p = \text{exit}_n(\text{main})$. Third, intra-graph transitions are deterministic. Forth, transitions are atomic.

A *computation sequence* of an application A is either a finite or an infinite sequence of

$$(p_0, \sigma_0) \xrightarrow{l_1} (p_1, \sigma_1) \xrightarrow{l_2} (p_2, \sigma_2) \dots$$

where, for all i , the transition $(p_i, \sigma_i) \xrightarrow{l_{i+1}} (p_{i+1}, \sigma_{i+1})$ is justified by a transition in the transition relation of A . When a computation sequence is finite, then it ends with a configuration. A *run of a procedure P in A from a state σ_0* is a computation sequence of A such that $p_0 = \text{entry}(P)$. For every run of a command procedure P , the run terminates when it reaches an exit configuration for P , and can only terminate in such a configuration. We say that the run *terminates normally* (*terminates abruptly*) if the final configuration is a normal (abrupt) exit configuration for P . A *run of an application A from a state σ* is a run of the procedure *main* from σ . Especially for *main*, a run of *main* *terminates normally* if the final configuration is a normal exit configuration for *main*, and *terminates abruptly* if the final configuration is an abrupt exit configuration for any procedure. A *run of a transition graph G in an application A* is a computation sequence of A such that $p_0 = \text{entry}(G)$ and for all i , the pairs (p_i, p_{i+1}) is an edge of the graph.

2.2 Representation Correspondences

For our discussion on representation correspondences (RCRs), we assume that we are given two models S and T of an application, where T is an implementation of S . That is, S and T can be, respectively, an SPM and an FSP, or they can be, respectively an FSP and a TDS. For simplicity, we assume that each command in S has a corresponding command, with the same name, in T , and vice versa. We assume further that S and T have disjoint sets of transition graphs and disjoint sets of variables.

To define RCRs, we associate with both S and T the same set of observable events, and for each observable event we associate a one-to-one correspondence between observable variables of S and T at the start or final configurations of the transitions that emit the event. Intuitively, there is an RCR between S and T if for every run of T , there is a run of S on the same input, such that (1) both runs terminate or generate infinite computation sequences, (2) these runs exhibit the same sequence of observable events, (3) the values of corresponding observable variables in the configurations of each corresponding events coincide, and (4) vice versa for every run of S .

We first discuss the set of observable events. For every procedure P , we associate every incoming edge into $\text{exit}_n(P)$ with either a Pass_P or a Fail_P events. The first event denotes a successful completion of a run of P , while the latter denotes a logic failure. We associate every incoming edge into $\text{exit}_a(P)$ with an Abrupt_P event and every call transition to a procedure P with a Call_P event.

Next, we associate one-to-one correspondences between observable variables for events. For each command procedure P and for every configuration γ such that there is a configuration γ' and $\gamma' \xrightarrow{l} \gamma$ where l is associated with Pass_P , we associate with γ a set O_S of observable variables if γ belongs to an S 's run, and a set O_T if γ belongs to a T 's run, such that there is a one-to-one correspondence Obs between O_S and O_T . Similarly for l associated with Fail_P and Abrupt_P . When l is associated with

Call_P, then, instead of γ , we associate O_S and O_T with γ' such that if the parameters of P in S and in T are, respectively, $\bar{x} = x_1, \dots, x_m$ and $\bar{y} = y_1, \dots, y_n$, then $m = n$, $\{x_1, \dots, x_m\} \subseteq O_S$ and $\{y_1, \dots, y_n\} \subseteq O_T$, and Obs maps x_i to y_i for all $i = 1, \dots, m$. We also associate entry configurations of *main* with the sets O_S and O_T such that the input variables of S and T are mapped to each other.

We associate *observation function* $\mathcal{O}$ with each S and T to identify observable configurations and transition labels. That is, for a configuration γ , the function $\mathcal{O}(\gamma) = \gamma$ if γ is associated with a set of observable variables, otherwise $\mathcal{O}(\gamma) = \perp$. Similarly, for a label l of a transition, $\mathcal{O}(l) = e$ if l emits an observable event e , otherwise $\mathcal{O}(l) = \perp$. An *observation sequence* of a computation sequence R , denoted by $o(R)$, is obtained by turning R into an alternating sequence of configurations and transition labels, and applying the observation function $\mathcal{O}$ to each configuration and transition label of R . That is, for a computation sequence $R = \gamma_0 \xrightarrow{l_1} \gamma_1 \xrightarrow{l_2} \gamma_2 \xrightarrow{l_3} \dots$, we have $o(R) = \mathcal{O}(\gamma_0), \mathcal{O}(l_1), \mathcal{O}(\gamma_1), \mathcal{O}(l_2), \mathcal{O}(\gamma_2), \mathcal{O}(l_3), \dots$. A $\perp$ -free *observation sequence* of a computation sequence R , denoted by $o_\perp(R)$ is obtained from $o(R)$ by suppressing $\perp$ in $o(R)$.

We say that two states σ_1 and σ_2 are *compatible* with respect to a one-to-one correspondence Obs between the sets O_1 and O_2 of observable variables in the domain of, respectively, σ_1 and σ_2 if for every $x \in O_1$, we have $\sigma_1(x) = \sigma_2(Obs(x))$. Two configurations $\gamma_1 = (p_1, \sigma_1)$ and $\gamma_2 = (p_2, \sigma_2)$ are *compatible* if there are sets O_1 and O_2 of observable variables associated with γ_1 and γ_2 such that (1) there is a one-to-one correspondence Obs between O_1 and O_2 , and (2) σ_1 and σ_2 are compatible with respect to Obs .

DEFINITION 2.1. We say that two computation sequences R_1 and R_2 are *observationally equivalent* (or *stuttering equivalent*) if, let

$$o_\perp(R_1) = \theta_1, \theta_2, \dots \quad o_\perp(R_2) = \theta_1, \theta'_2, \dots,$$

$o_\perp(R_1)$ and $o_\perp(R_2)$ are of the same length, and for all i , we have either (1) $\theta_i = \gamma$ and $\theta'_i = \gamma'$, for configurations γ and γ' , such that γ and γ' are compatible, or (2) $\theta_i = \theta'_i$. $\square$

DEFINITION 2.2. There is a *representation correspondence* between a procedure P of S and a procedure P' of T if for every run R of P from a configuration γ , there is a run R' of P' from a configuration γ' , where γ and γ' are compatible, and vice versa, such that R and R' are observationally equivalent.

There is a *representation correspondence* between S and T if there is a representation correspondence between *main* of S and *main* of T . $\square$

In the above definition, due to call transitions and our assumption that $\mathbf{G}_{main}^n$ contains at least a call edge for every command procedure, the configurations γ and γ' have sets of observable variables associated with them. Note that to have γ and γ' compatible, then the procedures P and P' must refer to the same command. The notion of RCR for procedures is useful for proving RCR between S and T . Since *main* can be thought of as a loop that read input command and call the command, then proving RCR between S and T can be reduced to proving RCR between each corresponding commands.

3 Theory of Inter-Program Properties

In this section we describe an abstract theory for describing and proving properties that relate two programs, or *inter-program properties*. A detailed description of the theory can be found in [16]. The theory deals with programs that are represented as transition graphs described in the previous section.

For describing and proving inter-program properties, the theory considers two programs P_1 and P_2 as a pair (P_1, P_2) , such that they have disjoint flow graphs and disjoint sets of variables. A state σ for the pair (P_1, P_2) can be considered as a pair $(\sigma_1, \sigma_2) = \sigma$, such that σ_1 is for P_1 and σ_2 is for P_2 . A configuration is a tuple $(p_1, p_2, \sigma_1, \sigma_2)$ such that (p_1, σ_1) is a configuration for P_1 and (p_2, σ_2) is a configuration for P_2 . The semantics of (P_1, P_2) is a transition relations containing two kinds of transitions:

1. $(p_1, p_2, \sigma_1, \sigma_2) \mapsto (p'_1, p_2, \sigma'_1, \sigma_2)$, such that $(p_1, \sigma_1) \mapsto (p'_1, \sigma'_1)$ is in P_1 ;
2. $(p_1, p_2, \sigma_1, \sigma_2) \mapsto (p_1, p'_2, \sigma_1, \sigma'_2)$, such that $(p_2, \sigma_2) \mapsto (p'_2, \sigma'_2)$ is in P_2 .

In the description of the theory in this section, we omit the transition labels for simplicity. Thus, a computation sequence is simply a sequence of configurations.

The theory assumes an *assertion language* and uses relation $\sigma \models \alpha$ to mean that the state σ satisfies the assertion α . For a configuration $\gamma = (p, \sigma)$, we write $\gamma \models \alpha$ for $\sigma \models \alpha$. An assertion is *valid* if it is satisfied by any state.

The formalization of the theory is based on the notion of assertion function. An *assertion function* of (P_1, P_2) is a partial function

$$I : \text{Point}_{P_1} \times \text{Point}_{P_2} \rightarrow \text{Assertion}$$

mapping pairs of program points of (P_1, P_2) to assertions, such that I is defined on $(\text{entry}(P_1), \text{entry}(P_2))$ and $(\text{exit}(P_1), \text{exit}(P_2))$. This requirement is technical as one can always define I on these pairs as $\top$. Assertions defined on such an I are called *inter-program assertions*. Given a pair of points $\hat{p}$ and a pair of states $\hat{\sigma}$ of (P_1, P_2) , we say that $\hat{p}$ is I -observable if $I(\hat{p})$ is defined. For a configuration $\gamma = (\hat{p}, \hat{\sigma})$, we write $\gamma \models I$ if $I(\hat{p})$ is defined and $\hat{\sigma} \models I(\hat{p})$.

The theory introduces the notion of weakly-extendible assertion function as a well-suited notion for describing inter-program properties.

DEFINITION 3.1. Let I be an assertion function of a pair (P_1, P_2) of programs. The function I is *weakly extendible* if every run

$$\gamma_0, \dots, \gamma_i$$

of (P_1, P_2) , such that $i \geq 0$, $\gamma_0 \models I$, $\gamma_i \models I$, and γ_i is not an exit configuration, can be extended to a run

$$\gamma_0, \dots, \gamma_i, \dots, \gamma_{i+n}$$

such that (1) $n > 0$, and (2) $\gamma_{i+n} \models I$.

In [16] we show that, without appealing to the standard proof technique that uses well-founded set, and using only inter-program assertions and the notion of weak extendibility, we can prove program equivalence and mutual simulations of two programs where one program has a loop that does not correspond to any loop in the other program, or even the loop is eliminated in the other program. For proving RCRs, we often encounter

such a situation. For example, PIN is a scalar variable in the SPM, but is an array variable in the FSP. So, for checking and updating the PIN, the FSP contains loops that do not exist in the SPM.

We now develop verification conditions that guarantee weak extendibility. To this end, we need a notion of path of pairs of programs. A path π of (P_1, P_2) can be viewed as a *trajectory* in a two dimensional space: $\pi = (\pi_1, \pi_2)$, where π_1 is a path in the flow graph of P_1 and π_2 is a path in the flow graph of P_2 . A path is *trivial* if it consists of a single pair of points. Given a *path* π and an assertion ψ , we denote by $wp_\pi(\psi)$ and $wlp_\pi(\psi)$, respectively, the weakest and the weakest liberal preconditions of π and ψ . Since we have to compute these preconditions, we assume that the programming language that we consider has the *weak precondition property*: for every path π and every assertion ψ , $wp_\pi(\psi)$ exists and can effectively be computed. One can also compute $wlp_\pi(\psi)$ since it is equivalent to $wp_\pi(\psi) \vee \neg wp_\pi(\top)$. The precondition for paths of pairs of programs can also be derived from the precondition of paths of single programs.

DEFINITION 3.2. Let I be an assertion function and Π be a set of nontrivial paths such that, for every $\pi \in \Pi$, we have $start(\pi)$ and $end(\pi)$ to be I -observable. Denote by $\Pi|_{(p, p')}$ the set of paths in Π whose first pair of points is (p, p') .

The *weak verification condition* $\mathbb{W}$ associated with I and Π consists of assertions of the form

$$I(start(\pi)) \Rightarrow wlp_\pi(I(end(\pi))),$$

where $\pi \in \Pi$ and assertions of the form

$$I(p) \Rightarrow \bigvee_{\pi \in \Pi|_{(p, p')}} wp_\pi(\top)$$

where (p, p') is I -observable. □

The first kind of assertion is a standard assertion for proving partial correctness of path. The second kind of assertion expresses that, whenever a configuration at p satisfies $I(p)$, the computation from this configuration will *inevitably* follows at least one path in Π .

THEOREM 3.3. Let $\mathbb{W}$, I and Π be as in Definition 3.2. If every assertion in $\mathbb{W}$ is valid, then I is weakly extendible. □

The notion of weak verification condition is our notion for certificates that certify inter-program properties. In the next section we will use inter-program assertions to describe correspondences between observable variables. Later, to prove an RCR between two commands, one has to prove other inter-program properties between transition graphs of the commands. These program properties altogether describe the RCR. To prove such properties, we define an assertion function and prove that the function is weakly extendible. The certificates certifying these properties form a certificate for the RCR.

4 Proving Representation Correspondences

For our discussion on proving RCRs, we consider the application models S and T described in Section 2. To prove an RCR between S and T , we are only concerned with

command procedures, that is, for each corresponding command procedures, we prove an RCR between the procedures.

For two models S and T , there is usually a one-to-one correspondence Obs between global observable variables of S and T such that the values of each corresponding variables coincide at the entry and normal exit configurations of every command run. To this end, let us consider some command procedure P . Let Obs_p, Obs_f, Obs_a be one-to-one correspondences specified for the end configurations of transitions emitting, respectively, a $Pass_P$, a $Fail_P$, an $Abrupt_P$ event. For simplicity of presentation, in the sequel let $Obs_p = Obs_f$. Let Obs_c be a one-to-one correspondence specified for the start configurations of transitions emitting $Call_P$. We require that Obs is included in Obs_p and Obs_c . We say that a correspondence f is included in a correspondence g if for every mapping $x \mapsto y$ in f is a mapping in g .

Denote by P^S and P^T , respectively, the command P in S and in T . Given a function f , we denote by $dom(f)$ the domain of f . For simplicity of notation, given a one-to-one correspondence g , we abbreviate the assertion $\bigwedge_{x \in dom(g)} x = g(x)$ to simply g . To prove an RCR between P^S and P^T , we do the following steps:

1. Let α be an assertion, such that the assertion $\alpha \Rightarrow Obs_c$ is valid. That is, α describes the correspondence Obs_c . The assertion α can also describe invariants specific to S or T . We prove that α is satisfied by the initializations of global variables.
2. We assert α at $(entry(\mathbf{G}_{P^S}^n), entry(\mathbf{G}_{P^T}^n))$ and α' at $(exit(\mathbf{G}_{P^S}^n), exit(\mathbf{G}_{P^T}^n))$ such that the assertions $\alpha' \Rightarrow Obs_p$ and $\alpha' \Rightarrow \alpha$ are valid. That is, we assume that the correspondence expressed by α holds in the entry configurations of the procedures, and is preserved in the exit configurations.
3. Let ψ, ψ' be assertions asserted at, respectively, $(entry(\mathbf{G}_{P^S}^a), entry(\mathbf{G}_{P^T}^a))$ and $(exit(\mathbf{G}_{P^S}^a), exit(\mathbf{G}_{P^T}^a))$ such that the assertion $\psi' \Rightarrow Obs_a$ is valid. That is, the correspondence Obs_a holds when procedure runs terminate abruptly.
4. We prove that for every finite run of $\mathbf{G}_{P^T}^n$, there is a finite run of $\mathbf{G}_{P^S}^n$ from configurations satisfying α , and vice versa, such that the final configurations of the runs satisfy the assertion ψ .

One can demonstrate (1) easily since it amounts to proving that the initializations of global variables satisfy α . In the sequel we focus on the steps (2), (3), and (4).

We present our proof technique for proving RCRs of commands by means of a *real* example of a command called checkPIN that is used for authenticating users. In this paper we only consider proving RCRs between the SPM and the FSP of the command. Proving RCRs between the FSP and the TDS follows the same steps above. The SPM is written in a domain-specific language, called command description language, that resembles a subset of Java. Each command can be thought of as a method that has clauses: one **pass** clause describing conditions and state updates of successful completion of a run of the command; one or more **fail** clauses describing logic failures and the corresponding state updates; and one **abrupt** clause describing abrupt behavior of the command. For each command procedure P , the **pass** and **fail** clauses of the command constitute the transition graph $\mathbf{G}_P^n$, while the **abrupt** clause constitutes the transition graph $\mathbf{G}_P^a$.

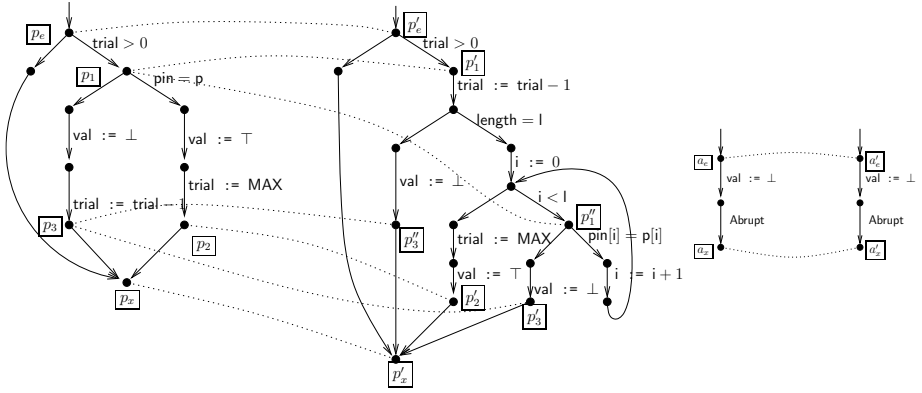


Fig. 1. SPM and FSP of checkPIN

The FSP is written in a subset of Java. Each command procedure P is a method of the form:

```
P(...) { try { ... } catch(CardTearException) { ... } }
```

The **try** part constitutes G_P^n , while the **catch** part constitutes G_P^a . Details of SPM and FSP can be found in our technical report [12].

EXAMPLE 4.1. We prove that there is an RCR between two corresponding commands procedure P_c by considering their transition graphs $G_{P_c}^n$ and $G_{P_c}^a$ separately. The left-hand pair of transition graphs in Figure 1 is G_{checkPIN}^n of the SPM, on the left of the pair, and G_{checkPIN}^n of the FSP, on the right of the pair. As a shorthand, we call the former P_1 and the latter P'_1 . For disjointness, we assume that all variables in P'_1 are primed.

First the global variables of the SPM that we want to observe are `trial`, `pin`, `val`, and `MAX`. They correspond to their primed counterparts in the FSP. Additionally, at the entries of P_1 and P'_1 , the input `pin p` corresponds to `p'`, and at the exits of P_1 and P'_1 , the event variable ε corresponds to ε' . Next, we have to define the equality between scalar PIN and array PIN. Every array PIN p is associated with a length l ; we write this association as (p, l) . We introduce predicate $\equiv$ between such pairs such that, given array PINs (p, l) and (p', l') , we say that $(p, l) \equiv (p', l')$ if $l = l'$, $l \geq 0$, and for all $i = 0, \dots, l - 1$, we have $p[i] = p'[i]$. We introduce a predicate $\sim$ which is axiomatized as follows: for every scalar PINs w, x and for every array PINs y, z ,

$$x \sim y \Rightarrow (y \equiv z \Leftrightarrow x \sim z) \quad x \sim y \Rightarrow (w = x \Leftrightarrow w \sim y).$$

The predicate $\sim$ defines the equality between a scalar PIN and an array PIN.

The following assertions express the correspondence between observable variables:

$$\begin{array}{lll} \phi_1 \Leftrightarrow \text{trial} = \text{trial}' & \phi_3 \Leftrightarrow \text{pin} \sim (\text{pin}', \text{length}') & \phi_5 \Leftrightarrow p \sim (p', l') \\ \phi_2 \Leftrightarrow \text{val} = \text{val}' & \phi_4 \Leftrightarrow \text{MAX} = \text{MAX}' & \phi_6 \Leftrightarrow \varepsilon = \varepsilon' \end{array}$$

Next, we define an assertion function I_1 of (P_1, P'_1) as follows:

$$\begin{aligned}
 I_1(p_e, p'_e) &= \bigwedge_{i=1}^5 \phi_i & I_1(p_x, p'_x) &= \bigwedge_{i=1}^6 \phi_i \\
 I_1(p_1, p'_1) &= \bigwedge_{i=1}^5 \phi_i \wedge \text{trial} > 0 \\
 I_1(p_1, p'_1) &= \bigwedge_{i=2}^5 \phi_i \wedge \text{trial} > 0 \wedge \text{trial} = \text{trial}' + 1 \\
 &\quad \wedge \text{length}' = l' \wedge i' < l' \wedge (\forall j. 0 \leq j < i' \Rightarrow \text{pin}'[j] = p'[j]) \\
 I_1(p_2, p'_2) &= \bigwedge_{i=1}^5 \phi_i \wedge \text{pin} = p \wedge (\text{pin}, \text{length}) \equiv (p, l) \\
 I_1(p_3, p'_3) &= I_1(p_3, p''_3) = \bigwedge_{i=1}^5 \phi_i \wedge \text{pin} \neq p \wedge (\text{pin}, \text{length}) \not\equiv (p, l)
 \end{aligned}$$

In this example, we prove an interesting part of RCR, that is, without any presence of card tears, for every run R of P_1 , there is a run R' of P'_1 from compatible states, such that R and R' are observationally equivalent. We denote by $\pi_{p,p'}$ a path from p to p' , and by π_p a trivial path consisting only of point p . We prove that I_1 is weakly extendible by the following reasoning. First, for every run of (P_1, P'_1) from an entry configuration that satisfies $I_1(p_e, p'_e)$, the run can reach (p_1, p'_1) by following the path $(\pi_{p_e, p_1}, \pi_{p'_e, p'_1})$ such that the end configuration satisfies $I_1(p_1, p'_1)$. From this configuration, the run can be extended either by following the path $(\pi_{p_1, p_3}, \pi_{p'_1, p'_3})$ or by following the path $(\pi_{p_1}, \pi_{p'_1, p''_1})$ such that the end configuration satisfies I_1 . From the configuration that satisfies $I_1(p_1, p'_1)$, the run can be extended either by following $(\pi_{p_1}, \pi_{p'_1, p'_1'})$, or by following $(\pi_{p_1, p_2}, \pi_{p'_1, p'_2})$, or by following $(\pi_{p_1, p_3}, \pi_{p'_1, p'_3})$. Without any of these paths, I_1 would not be weakly extendible. Thus, we have shown that, using the notion of weak extendibility, these paths show that the loop in P'_1 terminates.

Note that every possible transition of P_1 is described by the nontrivial paths that constitute the first elements of all pairs of paths above. Therefore, we have proved that for every run R of P_1 , there is a run R' of P'_1 from compatible states, such that R and R' are observationally equivalent. We can use the same reasoning for proving the other direction. Indeed, by taking the set of all the above pairs of paths, one can prove that all assertions of the weak verification condition associated with I_1 and the set are valid.

Consider now the righthand pair of transition graphs in Figure 1 is $\mathbf{G}_{\text{checkPIN}}^a$ of the SPM, on the left of the pair, and $\mathbf{G}_{\text{checkPIN}}^a$ of the FSP on the right of the pair. As a shorthand, we call the former P_2 and the latter P'_2 . The SPM and FSP only have to guarantee that the validation status is set to false in case of power loss. That is, the only observable variables are `val` and its primed counterpart.

We define an assertion function I_2 of (P_2, P'_2) such that we have $I_2(a_e, a'_e) = \top$ and $I_2(a_x, a'_x) = (\text{val} = \text{val})$. It is easy to that I_2 is weakly extendible, which means that if a card tear occurs and the configurations of the runs at (a_e, a'_e) satisfies I_2 , then both runs will emit the same event, which is $\text{Abrupt}_{\text{checkPIN}}$ and they both terminate in compatible states.

Finally we have to prove that for every finite run of P_1 with end state σ , there is a finite run of P'_1 with end state σ' , and vice versa, such that (σ, σ') satisfies $I_2(a_e, a'_e)$. Since $I_2(a_e, a'_e)$ is satisfied by every state, then we have finished our proof. $\square$

Proving RCRs between an FSP and a TDS is challenging due to the features of the language of the TDS. A TDS is written in a subset of Java Card [15], which includes transient and persistent memory as well as transaction mechanism. When a card tear occurs, data stored in persistent memory will be kept in the memory, while those stored

in transient memory will be lost. Variables whose values are stored in persistent memory are called *persistent variables*, while those whose values are stored in transient memory are called *transient variables*.

Transactions are managed by methods `beginTransaction`, `commitTransaction`, and `abortTransaction` with standard functionalities. The depth of a transaction is at most 1. When a transaction is in progress, the updates of persistent variables are conditional, in the sense that the updates will be materialized if `commitTransaction` is called. Regardless a transaction is in progress or not, the updates of transient variables are unconditional. To model card tears and transactions, we use the desugaring method in [9]. Each command in the TDS is a Java method, and desugaring the command means translating the method into the same form as that of the FSP, that is, the method has a big **try-catch** construct. The **catch** construct sets all transient variables to their default values, and cancel the updates of persistent variables if the card tear occurs when a transaction is in progress.

One might have to relax Definition 2.2 of RCRs to prove RCRs between an FSP and a TDS. Let us consider the following toy commands:

P_1 :	P_2 :	P'_1 :	P'_2 :
$x := 5;$	ϵ	if (inTrans) return ERROR;	if (inTrans) {
$y := 6;$		$xb := x';$	$x' := xb;$
		$yb := y';$	$y' := yb;$
		inTrans := $\top$;	}
		$x' := 6;$	
		$y' := 5;$	
		inTrans := $\perp$;	

The programs (or transition graphs) P_1 and P_2 constitute the **try** and **catch** parts of the FSP, respectively. (P_2 has no instruction.) Similarly for P'_1 and P'_2 of the desugared form of the TDS. Suppose that x' and y' are observable persistent variables that correspond to x and y , respectively. The variables xb and yb are back-up variables for x' and y' . The boolean variable inTrans indicates whether a transaction is in progress or not; assume that it is false at the entry of P'_1 . In case of abrupt terminations, we want to ensure that the above correspondence holds. To this end, we have to assert at the entries of P_2 and P'_2 the assertion ϕ below:

$$(\neg \text{inTrans} \Rightarrow x = x' \wedge y = y') \wedge (\text{inTrans} \Rightarrow x = xb \wedge y = yb)$$

For every finite run R' of P'_1 from a state satisfying inTrans = $\perp$, there is a finite run R of P_1 such that the final configurations of the runs satisfy ϕ . For example, if R' reaches the middle of transaction, e.g., the entry of $y' := 6$, then R simply stays at the entry of P_1 . However, showing the other way around is not possible. When a run R reaches the entry of $y := 6$, then there is no finite run R' of P'_1 such that the final configurations satisfy ϕ . Thus, according to Definition 2.2 there is no RCR between the commands.

To handle such an above case, one can relax Definition 2.2. That is, we only require that for every run R of P^T from a configuration γ , there is a run R' of P^S from a configuration γ' , where γ and γ' are compatible, such that R and R' are observationally equivalent. The drawback of this relaxed definition is that if P^T does not terminate

and the assertion at the entries of abrupt graphs is valid, then there is always an RCR between P^T and P^S . Nevertheless, with this relaxed definition, we can still preserve security properties for S in T , as shown in the following section.

5 Property Preservation

In this section we show how security properties of the SPM can be preserved in the FSP using RCRs. Property preservation between the FSP and the TDS can be explained in the same way. We are only concerned with security properties that can be characterized as partial correctness properties: a procedure P is *partially correct* with respect to a precondition α and a postcondition β , denoted by $\{\alpha\}P\{\beta\}$, if for every run of P from a state satisfying α and reaching an exit configuration, this configuration satisfies β .

Consider again the application models S and T and the one-to-one correspondences $Obs_p, Obs_f, Obs_a, Obs_c$ described at the beginning of Section 4. We show property preservation by the following theorem:

THEOREM 5.1. *Let α and β be, respectively, a precondition and a postcondition for a procedure P^S such that $\{\alpha\}P^S\{\beta\}$. Let α' and β' be, respectively, a precondition and a postcondition for a procedure P^T such that the assertions*

$$\begin{aligned} Obs_c &\Rightarrow (\alpha \Leftrightarrow \alpha') \\ (Obs_p \wedge \varepsilon = \text{Pass}_P) \vee (Obs_f \wedge \varepsilon = \text{Fail}_P) \vee (Obs_a \wedge \varepsilon = \text{Abrupt}_P) &\Rightarrow (\beta \Leftrightarrow \beta') \end{aligned}$$

are valid. If there is an RCR between P^S and P^T , then $\{\alpha'\}P^T\{\beta'\}$. $\square$

As an example, consider again the command and assertions in Example 4.1. Suppose that the property that we want to preserve is as follows: for any run of checkPIN, the value of variable `val` at the exit configuration of the run is true *if and only if* the run emits a `PasscheckPIN` event.

Let ψ be the assertion $(\text{val} = \top \Leftrightarrow \varepsilon = \text{Pass}_{\text{checkPIN}})$ and φ be the conjunction of the following assertions: (1) $\text{MAX} > 0$, (2) $0 \leq \text{trial} \leq \text{MAX}$, and (3) $\text{trial} < \text{MAX} \Rightarrow \text{val} = \perp$. The above property can be expressed as a partial correctness property $\{\varphi\}\text{checkPIN}\{\psi\}$. One can use standard Floyd-Hoare proof technique [6,8] to prove the property for both the SPM and the FSP.

Suppose that we have proved that the property holds for the SPM. We have shown in Example 4.1 that there is an RCR between the command `checkPIN` of the SPM and of the FSP. Let P be the command `checkPIN` in the FSP. Recall again the assertions $\phi_1, \dots, \phi_6$ in the example. Given an assertion α , let us denote by $p(\alpha)$ the assertion obtained from α by replacing each variable in α with its primed notation. Now, since the following assertions

$$\begin{aligned} \bigwedge_{i=1}^5 \phi_i &\Rightarrow (\varphi \Leftrightarrow p(\varphi)) \\ (\bigwedge_{i=1}^6 \phi_i \wedge (\varepsilon = \text{Pass}_P \vee \varepsilon = \text{Fail}_P)) \vee (\phi_2 \wedge \varepsilon = \text{Abrupt}_P) &\Rightarrow (\psi \Leftrightarrow p(\psi)) \end{aligned}$$

are valid, then by Theorem 5.1 we have $\{p(\varphi)\}P\{p(\psi)\}$.

6 Related Work and Conclusion

We developed a method for proving RCRs in the CC EAL7 certification of smart-card applications. We presented a modelling framework by which the representations of the SPM, the FSP, and the TDS can be modelled uniformly. Our framework is an extension of the modelling framework of procedural programs in [18], in the sense that we model abrupt behavior of procedures. Our definition of RCRs is mutual simulations between two application models. We apply the theory of inter-program properties in [16] for proving RCRs and providing certificates about them. The theory has been used for proving properties in translation validation approach to compiler verification [13,19,16]. In this paper we have shown another venue for the application of the theory. The application is beneficial since the theory provides a notion of certificate, which is essential in the CC EAL7 certification.

There have been a few works on formal specification and verification in the CC framework; closely related to ours is [4]. Their work is based on B method. Their definition of RCRs is similar to ours, in the sense that, for each command, they have a mapping between input-output relationships of two application models. Their work does not address complex data abstractions like our PIN, and their commands do not contain loops. However, their work has gone beyond ours in the sense that they included a model of Java Card API for APDU commands [15].

Another related work is by Heitmeyer et. al. on verifying enforcement of data separation in the kernel of a software-based embedded device [7]. Similar to ours, their work uses a state machine model consisting of events as a specification. Concrete code is partition into event code, trusted code, and other code. Event code corresponds to an event in the state-machine specification and such code is annotated with preconditions and postconditions. Their work construct two mappings: one is between events of the state machine and of the code, and the other is between assertions describing preconditions and postconditions of corresponding events. RCRs are proved for each corresponding events, that is, the precondition and the postcondition of an event in the code imply, respectively, the precondition and postcondition of the corresponding event in the specification. In their work, event code contains no loops, and they do not prove the relation between the code and its precondition and postcondition. Moreover, the mapping between assertions is based only on syntactic matching. Unlike ours, their work deals with real C code.

Other works on the CC certification have not addressed RCRs, or have only given little efforts on RCRs [3,17]. One distinguish feature in our work that has not been addressed by others is proving property preservation using RCRs.

There has been some work related to the specification and verification of smart-card applications, but not in the CC certification. Paper [14] describes a verification of Mondex electronic purse based on abstract state machine (ASM). The work is not in the CC, but it uses a notion of refinement simulation between ASMs to show correctness of a concrete implementation. The operations (similar to commands) in Mondex are simple and contains no loops and no complex data abstractions. The work in [2] describes a case study in the specification and verification of an electronic purse application. The work is concerned only with the specification and verification of commands in the implementation code. The work can complement our work in proving properties of the implementation code.

In this paper we do not address RCRs between the TDS and the implementation code. We assume that existing work on certified and certifying compilers [11,13] can be used to provide RCRs between the TDS and the implementation code. We are currently developing certification tools based on the method described in this paper. We take JML approach [10] to specifying assertion function. That is, we use special comments to put labels denoting program points in the programs, and write the assertion function in a separate file. We use off-the-shelf data-flow analyses, such as global value numbering, to assist users in defining assertion functions, so that users only concentrate on one-to-one correspondences between observable variables. We are developing heuristics based on observable events to alleviate the burdens of specifying paths in weak verification conditions; this is the topic of our future work. Assertions in the verification conditions can then be proved using SMT solvers, such as Yices [5].

References

1. Common Criteria for Information Technology Security Evaluation, Version 3.1, CCMB-2007-09-003 (2007)
2. Breunesse, C.-B., Cataño, N., Huisman, M., Jacobs, B.: Formal methods for smart cards: an experience report. *Sci. Comput. Program.* 55(1-3), 53–80 (2005)
3. Chetali, B., Nguyen, Q.-H.: Industrial use of formal methods for a high-level security evaluation. In: *Formal Methods*, pp. 198–213 (2008)
4. Dadeau, F., Potet, M.-L., Tissot, R.: A B formal framework for security developments in the domain of smart card applications. In: *Security Conference*, pp. 141–155 (2008)
5. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) *CAV 2006*. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
6. Floyd, R.W.: Assigning meaning to programs. In: Schwartz, J.T. (ed.) *Proceedings of Symposium in Applied Mathematics*, pp. 19–32 (1967)
7. Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.: Formal specification and verification of data separation in a separation kernel for an embedded system. In: *CCS 2006: Proceedings of the 13th ACM conference on Computer and communications security*, pp. 346–355. ACM, New York (2006)
8. Hoare, C.A.R.: An axiomatic basis for computer programming. *CACM* 12(10), 576–580 (1969)
9. Hubbers, E.-M.G.M., Poll, E.: Reasoning about card tears and transactions in Java Card. In: Wermelinger, M., Margaria-Steffen, T. (eds.) *FASE 2004*. LNCS, vol. 2984, pp. 114–128. Springer, Heidelberg (2004)
10. Leavens, G., Cheon, Y.: Design by contract with JML (2003)
11. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *SIGPLAN Not.* 41(1), 42–54 (2006)
12. Narasamdya, I., Périn, M.: Certification of smart-card applications in common criteria. Technical Report TR-2008-14, Verimag (September 2008)
13. Rinard, M., Marinov, D.: Credible compilation with pointers. In: *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy (July 1999)
14. Schellhorn, G., Grandy, H., Haneberg, D., Reif, W.: The mondex challenge: Machine checked proofs for an electronic purse. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) *FM 2006*. LNCS, vol. 4085, pp. 16–31. Springer, Heidelberg (2006)

15. Sun Micro systems, Inc, Palo Alto, California. Java Card 3.0 Platform Specification (2008), <http://java.sun.com/javacard/3.0/>
16. Voronkov, A., Narasamdya, I.: Proving inter-program properties. Technical Report TR-2008-13, Verimag (2008)
17. Wilding, M., Greve, D.A., Hardin, D.: Efficient simulation of formal processor models. *Formal Methods in System Design* 18(3), 233–248 (2001)
18. Zaks, A., Pnueli, A.: CoVaC: Compiler validation by program analysis of the cross-product. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) *FM 2008*. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008)
19. Zuck, L.D., Pnueli, A., Goldberg, B.: VOC: A methodology for the translation validation of optimizing compilers. *J. UCS* 9(3), 223–247 (2003)

Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks

Frank Hermann, Hartmut Ehrig, and Claudia Ermel

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin

{frank.hermann,hartmut.ehrig,claudia.ermel}@tu-berlin.de

Abstract. E-government services usually process large amounts of confidential data. Therefore, security requirements for the communication between components have to be adhered in a strict way. Hence, it is of main interest that developers can analyze their modularized models of actual systems and that they can detect critical patterns. For this purpose, we present a general and formal framework for critical pattern detection and user-driven correction as well as possibilities for automatic analysis and verification at meta-model level. The technique is based on the formal theory of graph transformation, which we extend to transformations of type graphs with inheritance within a type graph hierarchy. We apply the framework to specify relevant security requirements.

The extended theory is shown to fulfil the conditions of a weak adhesive HLR category allowing us to transfer analysis techniques and results shown for this abstract framework of graph transformation. In particular, we discuss how confluence analysis and parallelization can be used to enable parallel critical pattern detection and elimination.

1 Introduction

Software systems for e-government services have to provide a platform, where internal and external users can input and process large amounts of confidential data. Therefore it is important that considerable efforts are made to secure such data. To improve the security of software systems, recent research has identified that security analysis should be integrated into software engineering techniques and security should be considered from the early stages of the software systems development process [2]. Existing security modelling frameworks such as the UML profile *UMLsec* [3] support the design of security-sensitive systems by offering stereotypes to describe policies of system parts like communication channels or subsystems. Models then can be analyzed to check the satisfaction of security policies, such as access control conditions. Common techniques to elicit security requirements are based on use case modeling and goal-oriented approaches [4]. The problem is that these techniques are better suited for the elicitation of functional requirements. Security requirements being non-functional requirements are closely related to system architecture design and frequently require

architectural changes as reactions to detected critical patterns. Moreover, the *UMLsec* profile specifies only core security requirements and has to be refined for more specific application fields like secure e-government services.

In order to be able to specify flexible architectural changes as reactions to detected critical patterns in the design of e-government systems, we propose in this paper a dynamic, general modelling approach based on typed graph transformation for critical pattern detection and elimination.

Public administration is based on a strict hierarchical structure of e-government networks. We reflect this fundamental design paradigm in our modelling approach by supporting hierarchies along a chain of meta-model layers. The common approach of *meta-modelling* uses UML class diagrams equipped with OCL constraints to model a domain-specific language's (DSL's) abstract syntax in a declarative way (see e.g. the MOF approach by the OMG [5]). Graph grammars [6] are a more constructive alternative, based on a formal categorical framework which can also be used for formal analysis and verification. A DSL here is modelled by a type graph capturing the definition of the underlying symbol and relation types. Instances of a DSL are given by graphs typed over (i.e. conforming to) the type graph, and can be further restricted by defining rule-based instance generation operations. A DSL type graph corresponds closely to a meta-model, i.e. also inheritance relations are used¹. Hence, the main technical contribution of this paper lies in solving the challenge of transformation of graphs with inheritance hierarchies.

As running example, we consider an e-government system application which is based on a standard given by the *E-Government Manual of the Federal Office for Information Security* in Germany. In particular, we here focus on Chapter IV [8]. There are four main zones in the architecture of an e-government system (depicted in Fig. 1), one client zone for the external view and three security zones, which are under control of the corresponding government institution.

E-government services are installed on web servers in zone one, which can access actual applications of the public agency in zone two, but they are not directly connected to confidential data. Hiding these data in zone three improves the security against external attacks. If the data was stored in zone two already, an intrusion on a web server could directly enable scans of the data file system and further more critical changes.

In the following sections we discuss how this standard structure of an e-government network can be refined, customized and analyzed on the basis of formal type graph transformation with inheritance. Transformations and analysis are performed on the type graph of the e-government network visualized in Fig. 1. The overall model consists of a hierarchy of models with several meta-levels, all formalized by type graphs. Type graphs with inheritance and typed graph transformation have been introduced already in [6,9] but without transformation of the meta-levels including inheritance. The new formal approach in this paper concerns a generalization of typed graph transformation to the

¹ Note that the type graphs used for network modelling in our previous paper [7] did not yet allow the use of inheritance.

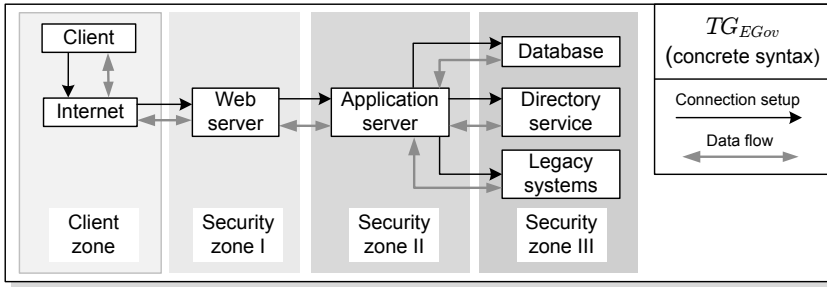


Fig. 1. Scenario: structure of E-Government networks

transformation of type graphs with inheritance. The key concepts thus are graphs with inheritance, called *I*-graphs, and *I*-graph morphisms based on clan morphisms [9], coming up with a new category **IGraphs**, which is shown to fulfil the requirements of weak adhesive HLR categories [6]. This allows us to make use of formal techniques for confluence and dependency analysis to analyze critical pattern detection and elimination in the e-government network model.

Graphs with inheritance could also be transformed by encoding the graphs to plain graphs with the help of a special edge type for the inheritance relation and performing standard graph transformation on them. But this leads to several problems. All inheritance *paths* have to be translated to direct edges, and after performing a transformation step the resulting graph would have to be extended by the edges which form the transitive closure of the inheritance relation. Furthermore, extending matching to inheritance hierarchies, as considered in this paper, is not possible if inheritance is encoded by special edges in plain graphs.

The paper is structured as follows: In Sec. 2 we show how type graph rules and transformations including the handling of inheritance can be used to model network configurations for secure client-server architectures for e-government networks [8]. Thereafter, we define the basic formal constructions for transforming type graphs with inheritance and show important properties in Sec. 3, which will then be used in Sec. 4 for analyzing the e-government network model. Sec. 5 discusses related work, and Sec. 6 concludes the paper. Our technical report [22] contains the full proofs for the presented results.

2 Modelling E-Government Networks

In this section we show how type graph transformations including the handling of inheritance can be applied for developing and maintaining meta-models for e-governments networks [8].

Example 1 (Type Graphs for Network Configurations). Graph G_{EGov} in the lower left corner of Fig. 2 is an instance-level graph typed over the type graph TG_{EGov} for network configurations in the area of e-government. Graph G_{EGov} is shown in concrete syntax in the lower right corner of Fig. 2 and describes a client,

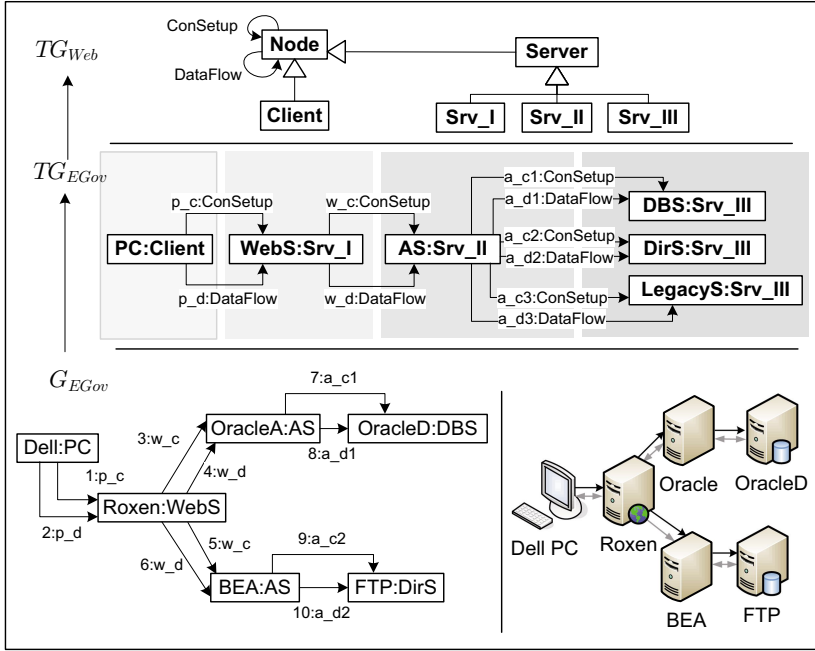


Fig. 2. Instance Graph G_{EGov} and Type Graph Hierarchy $TG_{EGov} \rightarrow TG_{Web}$

which is connected to services of the e-government institution. TG_{EGov} itself is typed over the more abstract type graph TG_{Web} which models domain specific languages of client-server architectures. Type mappings like $TG_{EGov} \rightarrow TG_{Web}$ are denoted by the type name following the respective node or edge name after the colon, e.g. the node “PC:Client” in TG_{EGov} is mapped to the node “Client” in TG_{Web} .

The main idea of graph transformation is the rule-based modification of graphs, which represent the abstract syntax of models. While standard graph transformation [6] considers transformations of instances typed over a given type graph only, we present an extension in Sec. 3 to deal with more general transformations including transformations of type graphs with inheritance, which may be typed over a type graph of the next meta level.

The core of a graph transformation rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ as defined in [6] is a triple of graphs (L, K, R) , called left-hand side, interface and right-hand side, and two injective graph morphisms $L \xleftarrow{l} K$ and $K \xrightarrow{r} R$. Interface K contains the graph objects which are not changed by the rule and hence occur both in L and in R . Applying rule p to a graph G means to find a match m of L in G and to replace this matched part $m(L)$ in G by the corresponding right-hand side R of the rule, thus leading to a graph transformation step $G \xrightarrow{p, m} H$.

Note that a rule may only be applied if the *gluing condition* is satisfied, i.e. the rule application must not leave *dangling edges*, and for two objects which are identified by m , the rule must not preserve one of them and delete the other

one. Furthermore, a rule p may be extended by a set of positive or negative *application conditions* (PACs and NACs) [10,6]. Intuitively, a NAC forbids the presence of a certain pattern in graph G , while a PAC requires it.

A match $L \xrightarrow{m} G$ satisfies a NAC with the injective NAC morphism $n : L \rightarrow NAC$, if there is no injective graph morphism $NAC \xrightarrow{q} G$ with $q \circ n = m$ (where “ $\circ$ ” denotes composition of morphisms), as shown in the diagram to the right. Analogously, a PAC is satisfied if there exists such an injective graph morphism $PAC \xrightarrow{q} G$. Our notion of graph transformation is called double-pushout approach (DPO) since both squares in the diagram are pushouts in the category of graphs, where D is the intermediate graph after removing $m(L)$ in G and in (PO_2) H is constructed as gluing of D and R along K .

$$\begin{array}{ccccc} NAC & \xleftarrow{n} & L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ & \searrow q & \downarrow m & & \downarrow (PO_1) & & \downarrow (PO_2) \\ & & G & \xleftarrow{m^*} & D & \xrightarrow{m^*} & H \end{array}$$

The following examples show how changes of type graphs *with inheritance*, like TG_{Web} and TG_{EGov} in Fig. 2, can be defined in a formal and concise way.

Example 2 (Rules for Editing Network Meta-Models). Fig. 3 and the top line of Fig. 4 show some typical editing rules, typed over TG_{Web} , where numbers specify the rule morphisms. Interface K contains the numbered elements in L only and is not shown explicitly in Fig. 3. The first two rules insert new nodes and connections. Note that rule “createCS()” can be applied to any pair of nodes, because the node types are specified abstractly. Rule “setUpdateConnection()” contains a NAC and defines the controlled extension of connections, i.e. a pair of links of types “ConSetup” and “DataFlow”, starting at a server node in zone 3. A new connection for requesting server updates can be established, but only if there is no incoming connection via the same server, because this would ease an attack from an external Internet connection.

Finally, rule “insertSupertype()” given by the top line in Fig. 4 specifies a sample refactoring operation, where a new super type node is created having three nodes of type “Srv_III” as specializations.

Example 3 (EGov Type Graph Transformation Step). Fig. 4 shows a graph transformation step, where rule “insertSupertype()” is applied to graph G_1 , a part of graph TG_{EGov} from Fig. 2, resulting in the transformed graph G_2 .

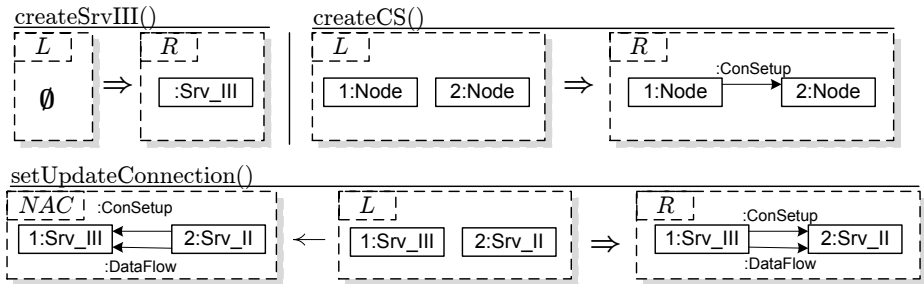


Fig. 3. Rules for Editing Type Graph TG_{E-Gov}

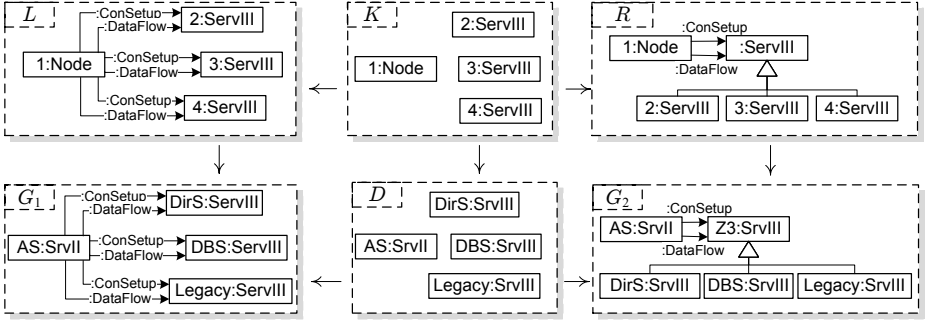


Fig. 4. Type Graph Transformation Step of rule insertSupertype()

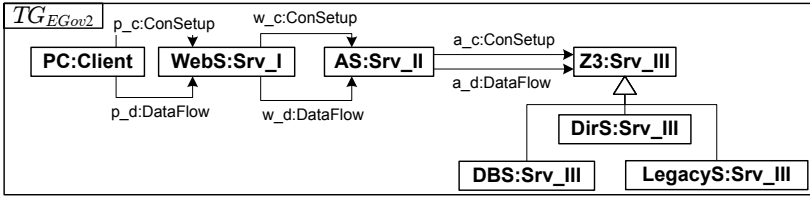


Fig. 5. Resulting Type Graph $TG_{EG_{Ov2}}$ as update of $TG_{EG_{Ov}}$

The result of applying the rule to the complete type graph $TG_{EG_{Ov}}$ yields the type graph $TG_{EG_{Ov2}}$ as shown in Fig. 5.

The examples show how transformations of type graphs with inheritance in e-government networks can be defined in a concise way. After presenting the underlying formalization in the next section we continue the example in Sec. 4 to show the relevant features of the approach for ensuring security in e-government networks.

3 Transformation of Graphs with Inheritance

Graph transformation with node type inheritance [6,9] provides main aspects of inheritance, in particular inheritance of attributes and edge types from parent node types to children node types. In this section we lift transformations from the instance level to the meta levels in order to support a formal basis for editing and analyzing meta-models, i.e. type graphs with inheritance within the framework of graph transformation. Recall further that meta-modelling is captured by graph transformation using the concept of type graph hierarchy [7,11].

Note that we use the algebraic notion of graphs, where a graph $G = (V, E, s, t)$ is given by a set of nodes V , a set of edges E and functions $s, t : E \rightarrow V$ specifying source and target nodes for each edge. A graph morphism $f : G_1 \rightarrow G_2$ is a pair of mappings $(f_V : V_1 \rightarrow V_2, f_E : E_1 \rightarrow E_2)$ compatible with source and target functions, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. In

order to improve readability of the paper we present our inheritance concepts first for graphs without attribution, but in [22] we show how all concepts and results can be extended to attributed graphs. Note that the following notion of I -graphs slightly differs from [6] by using a relation for capturing the inheritance information (instead of a separate graph with distinguished abstract nodes) in order to simplify further constructions.

Definition 1 (I -Graph). *Graph with Inheritance*, short I -Graph, is given by $GI = (G, I)$. It consists of graph G and inheritance relation $I \subseteq G_V \times G_V$, where for $v \in G_V$ $\text{clan}_I(v) = \{v' \in G_V \mid (v', v) \in I^*\}$ with I^* being the reflexive and transitive closure of I .

Remark 1. According to [6,9] as well as MOF [5] and UML [12] we do not require that the inheritance relation is cycle free.

I -graph morphisms - not considered in [6] - are based on clan-morphisms [6] taking into account inheritance.

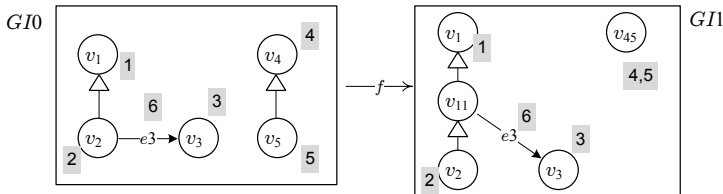
Definition 2 (Clan-Morphism). Given graph $G1$ and I -graph $GI2 = (G2, I2)$ a pair of mappings $f = (f_V, f_E) : G1 \rightarrow G2$ is called *clan-morphism*, written $f : G1 \rightarrow GI2$, if $\forall e1 \in G1_E$:

$$f_V \circ s_{G1}(e1) \in \text{clan}_{I2}(s_{G2} \circ f_E(e1)) \quad \wedge \quad f_V \circ t_{G1}(e1) \in \text{clan}_{I2}(t_{G2} \circ f_E(e1)).$$

I -graphs and I -graph morphisms define the category **IGraphs**.

Definition 3 (Category IGraphs). Given I -graphs $GI1 = (G1, I1)$ and $GI2 = (G2, I2)$, an I -graph morphism $f : GI1 \rightarrow GI2$ is given by a clan-morphism $f : G1 \rightarrow G2$, which is I -compatible, i.e. $(v, w) \in I1$ implies $(f(v), f(w)) \in I2^*$. The composition of I -graph morphisms $f : GI1 \rightarrow GI2$ and $g : GI2 \rightarrow GI3$ is defined by $g \circ f : GI1 \rightarrow GI3$ with $(g \circ f)_V = g_V \circ f_V : G1_V \rightarrow G3_V$ and $(g \circ f)_E = g_E \circ f_E : G1_E \rightarrow G3_E$. The category of I -graphs and I -graph morphisms is denoted by **IGraphs**.

Example 4 (I -graph Morphism). The following example shows I -graph morphism $f : GI0 \rightarrow GI1$ where grey numbers indicate the mappings. According to I -compatibility the identification of nodes v_4 and v_5 contained in $GI0$ is possible, because $(v_{45}, v_{45}) \in I1^*$. Furthermore, inheritance between $v1$ and $v2$ of $GI0$ can be refined into several steps as shown by node v_{11} in $GI1$. The clan morphism f can additionally map edges to edges between nodes of super types as shown by $e3$.



Remark 2. 1. I -compatibility is equivalent to
 $(v, w) \in I1^*$ implies $(f_V(v), f_V(w)) \in I2^*$.

2. Given I -graph morphisms f and g then: $g \circ f : GI1 \rightarrow GI3$ is an I -graph morphism, because I -compatibility of f and g implies that of $g \circ f$ and we can show for all $e_1 \in G1_E : (g \circ f)_V \circ s_{G1}(e_1) = g_V \circ f_V \circ s_{G1}(e_1) \in \text{clan}_{I3}(s_{G3} \circ (g \circ f)_E(e_1))$.
3. Each clan-morphism $f : G1 \rightarrow GI2$ is also an I -graph morphism $f : GI1 \rightarrow GI2$ with $GI1 = (G1, I1)$ and $I1 = \emptyset$, because in this case I -compatibility is trivial. This implies also that the composition of a clan-morphism $f : G1 \rightarrow GI2$ with an I -graph morphism $g : GI2 \rightarrow GI3$ is a clan morphism $g \circ f : G1 \rightarrow GI3$.

In order to enable automatic critical pattern detection and user driven transformation for meta-models we lift graph transformation from the instance level to all meta levels within the abstract framework of weak adhesive HLR categories [6]. This way we can apply the well-known results for the abstract framework, e.g. analysis and correction can be parallelized and distributed to meta-model parts in case of several e-government networks.

For defining a weak adhesive HLR category we need to distinguish a suitable class $\mathcal{M}$ fulfilling certain properties. We propose the class $\mathcal{M}_{S\text{-}refl}$ of subtype-reflecting morphisms, because on the one hand DPO-rules based on these morphisms are powerful enough to generate all kinds of cycle-free inheritance graphs on the meta-model level and on the other hand $(\mathbf{IGraphs}, \mathcal{M}_{S\text{-}refl})$ can be shown to be a weak adhesive HLR category with componentwise construction of pushouts and pullbacks. Note that this fails to be true for the class $\mathcal{M}$ of all injective I -graph morphisms.

The notion of *subtype reflection*, *short S -reflection*, defines the condition that for each node n in the image of a morphism f it holds that all subtypes of n are in the image of f as well. We will need this condition for the proof of Thm. 1.

Definition 4 (S -reflecting Morphism). An S -reflecting morphism $f1 : GI0 \rightarrow GI1$ is an I -graph morphism $f1 : GI0 \rightarrow GI1$, where $f1$ is an injective graph morphism and has the S -reflection property: $\forall (v_{11}, v_1) \in I1^*, v_0 \in GI0_V : v_1 = f1_V(v_0) \Rightarrow \exists v_{01} \in GI0_V : f1_V(v_{01}) = v_{11} \wedge (v_{01}, v_0) \in I0^*$.

All rules in Figures 3 and 4 are S -reflecting, i.e. their rule morphisms are S -reflecting. Note that standard graph transformation rules, i.e. rules without inheritance, can be interpreted as S -reflecting rules by adding empty inheritance relations to their graphs. According to Thm. 1 and Thm. 2 in [22] pushouts and pullbacks along S -reflecting I -graph morphisms can be constructed componentwise and the class $\mathcal{M}_{S\text{-}refl}$ is closed under pushouts and pullbacks. Therefore, DPO transformations of S -reflecting rules are well defined and can be constructed componentwise in $\mathbf{IGraphs}$. Furthermore theses properties are part of the conditions for weak adhesive HLR categories and in fact, the category $(\mathbf{IGraphs}, \mathcal{M}_{S\text{-}refl})$ is a weak adhesive HLR category (see Remark 3 below).

Theorem 1 ($(\mathbf{IGraphs}, \mathcal{M}_{S\text{-}refl})$ is Weak Adhesive HLR Category). The category $\mathbf{IGraphs}$ of graphs with inheritance together with the class $\mathcal{M}_{S\text{-}refl}$ of S -reflecting morphisms is a weak adhesive HLR category.

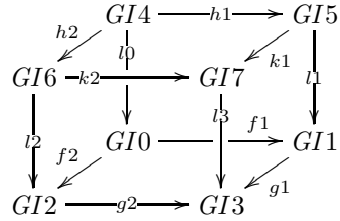
Remark 3 (Weak adhesive HLR category). According to the definition of weak adhesive HLR categories (see Definition 4.13 in [6]) $(\mathbf{IGraphs}, \mathcal{M}_{S-refl})$ has this property if

1. $\mathcal{M}_{S-refl}$ is a class of monomorphisms closed under isomorphisms, composition and decomposition
2. $\mathbf{IGraphs}$ has pushouts and pullbacks along $\mathcal{M}_{S-refl}$ -morphisms and $\mathcal{M}_{S-refl}$ is closed under pushouts and pullbacks
3. $(\mathbf{IGraphs}, \mathcal{M}_{S-refl})$ has the weak VK -property, i. e. given a cube as below, where the bottom face is a pushout with $f1 \in \mathcal{M}_{S-refl}$ and the back faces are pullbacks and one of the following two cases is satisfied, then we have: top square is pushout $\Leftrightarrow$ front squares are pullbacks.

case 1 Also $f2 \in \mathcal{M}_{S-refl}$.

case 2 Also $l1, l2, l3 \in \mathcal{M}_{S-refl}$.

We can conclude for each direction of the equivalence by item 2 in case 1: also $g1, g2, h1, h2, k1, k2 \in \mathcal{M}_{S-refl}$ and in case 2: also $g2, h1, k2, l0 \in \mathcal{M}_{S-refl}$.



Proof (see [22]).

Corollary 1 (Results for $(\mathbf{IGraphs}, \mathcal{M}_{S-refl})$). The following results for graph transformation based on $(\mathbf{IGraphs}, \mathcal{M}_{S-refl})$ are valid:

- Local Church Rosser Theorem for pairwise analysis of sequential and parallel independence (Thm. 5.12 in [6])
- Parallelism Theorem for applying independent rules and transformations in parallel (Thm. 5.18 in [6])
- Concurrency Theorem for applying E -related dependent rules simultaneously (Thm. 5.23 in [6])
- Embedding and Extension Theorem for transferring transformations and analysis results to more complex scenarios (Thms. 6.14 and 6.16 in [6])
- Local Confluence Theorem and Completeness of critical pairs for analyzing conflicts and for showing local Confluence (Thm. 6.28 and Lemma 6.22 in [6])

Proof (Idea). These results are shown in [6] for weak adhesive HLR categories with some additional properties (see Remark 6 in [22]) and are valid for $(\mathbf{IGraphs}, \mathcal{M}_{S-refl})$ by Thm. 1.

Before we show how the results in Corollary 1 can be applied in our scenario of e-government networks let us discuss other approaches which may avoid to work in the category $\mathbf{IGraphs}$. The intuitive semantics of an I -graph GI is the graph $\overline{GI}$ defined by closure or flattening of the inheritance relation I (see Def. 6 in [22]) as considered already for type graphs with inheritance in [6]. The inheritance closure is a cofree construction (see Thm. 7 in [22]) leading to a cofree functor

from **IGraphs** to **Graphs**. This implies that pullbacks are preserved, but as shown in Example 5 in [22] - pushouts are not preserved in general. For this reason, transformations with inheritance cannot easily be reduced to standard graph transformation by flattening (see more details in [22]).

4 Analysis of E-Government Network Meta-Models

During each phase of system design critical patterns may occur, which can imply unwanted behaviour and possibilities for a loss of security. The earlier they can be detected and the earlier they can be corrected the lower is the risk of a system containing critical parts in its implementation. This motivates to apply analysis techniques as early and as abstract as during the meta-model development. This section shows how critical patterns can be specified and automatically eliminated. In order to explain our approach we first describe a specific attack to an e-government system. Even though the cause of this attack is hard to detect on the implementation level the elimination of a suitable critical pattern in the meta-model ensures that this attack cannot occur. For the attack we assume that an intruder got access to the web server already.

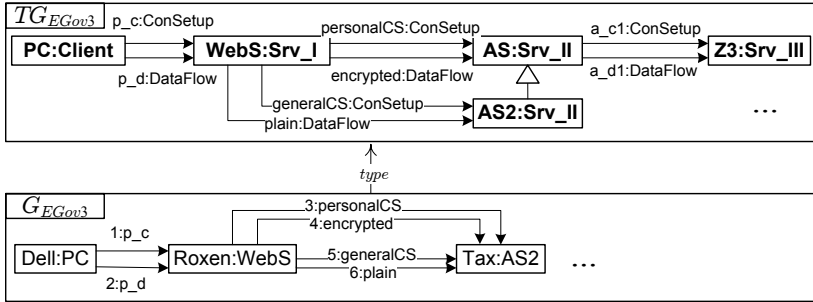


Fig. 6. Configuration for possible attack

Example 5 (Intrusion Attack). Fig. 6 shows a meta-model TG_{EGov3} and an instance G_{EGov3} with clan morphism $type$. There are two types for possible connection setups from server “Roxen” to server “Tax”, because of the inheritance relation between “AS2” and “AS” in TG_{EGov3} . Assume that the application server “Tax” in G_{EGov3} processes both confidential requests for receiving and updating personal information for tax declaration via secure encrypted data channel “4:encrypted” and requests for general information regarding dates, laws and submission address for preparing a tax declaration via unencrypted channel “6:plain”. The following sequence describes the intrusion:

- A user requests general information, stays connected and performs a log-in to request in addition also personal information.
- Because of high load of channel 4 a scheduling algorithm on web server “Roxen” decides to transfer some personal data via channel 6.

- The user receives the data, which is not encrypted during the communication.
- The intruder with access to the web server may now observe the insecure communication and intercept some confidential data.

A successful interception of the response is hidden. Even if misuse of confidential data for another service is detected at a later stage, locating the error is hard. Even though the channels were initially assigned correctly according to the kind of data the intrusion happened, because of a side effect of the scheduling algorithm, which is hidden to the model. Hence, possibilities for side effects on the implementation basis should be minimized.

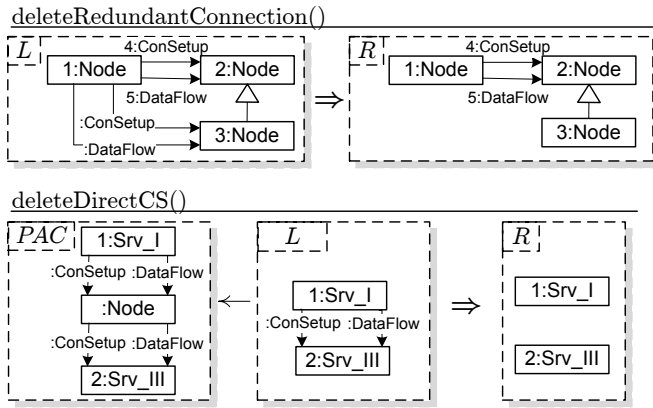


Fig. 7. Checking rules for analysis

Rule “deleteRedundantConnection()” in Fig. 7 can detect the critical pattern of web servers that can communicate via different types of connections simultaneously. A valid match of the rule states a detection and the developer of the model may apply the rule for automatic correction causing the deletion of the more specific connection type. This deletion of edges “generalCS” and “plain” in TG_{EGov3} implies in particular that instance G_{EGov3} is not typed correctly any more, because the edges 5 and 6 cannot be mapped type consistently.

A further rule for analysis and correction is given by “deleteDirectCS()” in Fig. 7. The positive application condition PAC requires a possible connection setup via a proxy node, while the left hand side L already matches a direct connection setup link between a server of zone I and a server of zone III. This situation may easily occur, if verbal requirements for the model are realized directly. Since communication shall only be possible between neighbouring zones this pattern is critical and has to be corrected by applying rule “deleteDirectCS()”. Note especially that the pattern is very flexible, because the proxy node is of the general type “Node”.

In the following we show how we can apply the well-known results for adhesive HLR systems (see Cor. 1).

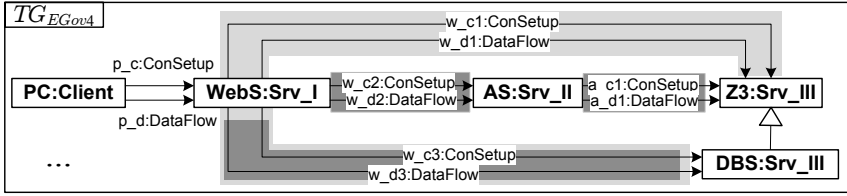


Fig. 8. Conflict situation for rules $deleteDirectCS()$ and $deleteRedundantConnection()$

Example 6 (Critical Pair). Fig. 8 shows graph TG_{EGov4} , which demonstrates a conflict situation for the rules “ $deleteDirectCS()$ ” and “ $deleteRedundantConnection()$ ” (see Fig. 7). Both rules can be applied to this graph and the matches are indicated by dark respectively light grey marked regions, where the first match is a proper clan-morphism. Both matches overlap on edges “ w_c3 ” and “ w_d3 ” that will be deleted by the rule applications. Thus, these rule applications are parallel dependent and there is a conflict of deciding which one to apply. This leads to a critical pair.

If in other situations the rule applications overlap only in their interfaces they are parallel independent and according to the Local Church Rosser and Parallelism Theorem (see Corollary 1) we can apply the rules in any order or in parallel.

According to the general result on completeness of critical pairs (see Corollary 1) there is a critical pair for each possible conflict. Hence, it suffices to calculate all critical pairs using tool support, which is available for standard graph transformation already [1]. If all critical pairs are strictly confluent we can apply the Local Confluence Theorem (see Corollary 1) in order to show that different applications of the analysis rules lead to the same result. Otherwise the aim is to group the analysis rules, such that there is no critical pair between two of the same group. In this way the analysis in each group can be applied in parallel using one parallel rule according to the Parallelism Theorem.

In practical situations meta-models are more complex, which results in a higher amount of node and edge types. Since critical patterns do normally contain only few nodes and edges it is quite usual that several rules are independent from each other and can be put in the same independence group. Therefore, our approach scales up for complex systems, where an automatic critical pattern detection and elimination is highly desirable. Note in particular that pure critical pattern detection without correction will never involve conflicts, since there is no deletion. For this reason it can be parallelized and distributed without calculating critical pairs.

Altogether we can use the results in Corollary 1 for parallel critical pattern detection and analyze how far different orders of the elimination of these patterns lead to the same result.

5 Related Work

In this paper we consider rule-based meta-model transformations in order to change meta-models in a way that makes them adhere to security requirements. This includes refactoring steps, such as inserting supertype nodes. Usually, model refactorings are performed at instance model level. Various approaches exist using graph transformation to provide a formal specification of model refactorings [13,14,15,16]. It has the advantage of defining refactorings in a generic way, while still being able to provide tool support in commonly accepted modeling environments such as EMF [17]. In addition, the theory of graph transformation allows the modeller to formally reason about dependencies between different types of refactorings. Synchronized rules are applied in parallel to keep coherence between models. Considering the special case where exactly two parts (one model diagram and the program or two model diagrams) are related, the triple graph grammar (TGG) approach by Schürr et al. [18] is used frequently.

Our transformation approach at meta-model level is most useful during meta-model development to ensure security requirements before instance graphs are created. An interesting line of research is the co-evolution of meta-models of higher levels and the corresponding meta-models at lower levels, down to instance models. Changing one meta level may cause implications for model updates of lower levels to keep them consistent (*migration problem*). A promising approach for automatic migration of instances is described in [19], where meta-model changes are transferred to lower levels by pullback constructions using non-injective morphisms. In this case, the rule morphisms $K \xrightarrow{L} L$ for the meta level transformations have to be non-injective. This leads to non-functional behaviour of DPO rewriting. In [20], SqPO rewriting is introduced, which is an extension of DPO rewriting taking into account this problem.

6 Conclusion

The formal basis for type graphs with inheritance was presented already in [21,6,9] and the semantics given by the closure construction coincides with the one of the inheritance concept of the meta-modelling language MOF [5]. For this reason, the presented extension of the theory to transformations of type graphs with inheritance enables DSL modellers to define modifications of meta-models which contain inheritance information. Apart from the presented case study of e-government network security, a wide range of meta-model based application domains are conceivable, in particular hierarchical and integrated systems of meta-models.

The paper showed that graphs with inheritance together with the introduced class of S -reflecting morphisms forms a weak adhesive category. Hence, the introduced formalization of transformations of meta-models allows modellers to apply various techniques for analysis of the meta-modelling process, due to the fact that well-known results for confluence analysis and conflict detection exist for weak adhesive HLR systems [6]. For instance, in the case of the sample

scenario, when the necessary meta-model changes of several modellers conflict each other, the formal techniques for merging and conflict detection support a consistent synchronization. And in the case of local changes of parts or views of the model, the changes can be embedded into the overall model if the consistency condition of the Embedding Theorem is fulfilled. Note that the presented approach is suited also in other application domains for checking formally the fulfillment of security requirements during design phase.

Future work on the theoretical formalization will include an analysis of the gluing condition and characterization of critical pairs for transformations of graphs with inheritance. Moreover, the migration problem discussed in Sec. 5 is an important problem when meta-models have to be modified where instance models exist which have to be kept consistent. The SqPO rewriting approach [20] seems to be a good candidate for future extensions of the presented theory in the context of model migration. Finally the critical pair analysis of the tool AGG shall be extended to the case of graphs with inheritance.

References

1. AGG Homepage, <http://tfs.cs.tu-berlin.de/agg>
2. Mouratidis, H., Giorgini, P. (eds.): Integrating Security and Software Engineering: Advances and Future Vision. Idea Group, IGI Publishing Group (2006)
3. Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)
4. Haley, C., Moffett, J., Nuseibeh, B.: Security Requirements Engineering: A Framework for Representation and Analysis. IEEE Trans. on Software Engineering 34(1), 133–153 (2008)
5. Object Management Group: Meta-Object Facility (MOF), Version 2.0. (2006), <http://www.omg.org/technology/documents/formal/mof.htm>
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science. Springer, Heidelberg (2006)
7. Braatz, B., Brandt, C., Engel, T., Hermann, F., Ehrig, H.: An approach using formally well-founded domain languages for secure coarse-grained IT system modelling in a real-world banking scenario. In: Proc. 18th Australasian Conference on Information Systems (ACIS 2007) (2007)
8. Federal Office for Information Security (BSI): Chapter IV: Secure Client-Server Architectures for E-Government. In: E-Government Manual. INTESIO 1–179 (2006), <http://www.bsi.bund.de/english/topics/egov/6verb>
9. Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed Graph Transformation with Node Type Inheritance. Theoretical Computer Science 376(3), 139–163 (2007)
10. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Special issue of Fundamenta Informaticae 26(3,4), 287–313 (1996)
11. Ehrig, H., Ehrig, K., Ermel, C., Prange, U.: Consistent Integration of Models Based on Views of Visual Languages. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 62–76. Springer, Heidelberg (2008)
12. Object Management Group: Unified Modeling Language: Superstructure – Version 2.1.1. formal/07-02-05 (2007), <http://www.omg.org/technology/documents/formal/uml.htm>

13. Mens, T., Taentzer, G., Müller, D.: Model-driven software refactoring. In: Rech, J., Bunse, C. (eds.) *Model-Driven Software Development: Integrating Quality Assurance*, pp. 170–203. Idea Group Inc. (2008)
14. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. *Software and System Modeling* 6(3), 269–285 (2007)
15. Grunske, L., Geiger, L., Zündorf, A., Van Eetvelde, N., Van Gorp, P., Varro, D.: Using Graph Transformation for Practical Model Driven Software Engineering. In: Beydeda, S., Book, M., Gruhn, V. (eds.) *Model-driven Software Development*, pp. 91–118. Springer, Heidelberg (2005)
16. Bottoni, P., Parisi-Presicce, P., Mason, G., Taentzer, G.: Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformations. In: van Bommel, P. (ed.) *Handbook on Transformation of Knowledge, Information, and Data: Theory and Applications*, pp. 95–125. Idea Group Publishing (2005)
17. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) *MoDELS 2006*. LNCS, vol. 4199, pp. 425–439. Springer, Heidelberg (2006)
18. 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)
19. Löwe, M., König, H., Peters, M., Schulz, C.: Refactoring Information Systems. In: *Proc. Software Evolution through Transformations: Embracing the Chance (SeTra 2006)*. Electronic Communications of the EASST, vol. 3 (2006)
20. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-Pushout Rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) *ICGT 2006*. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006)
21. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating Meta-Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In: Wermelinger, M., Margaria-Steffen, T. (eds.) *FASE 2004*. LNCS, vol. 2984, pp. 214–228. Springer, Heidelberg (2004)
22. Hermann, F., Ehrig, H., Ermel, C.: Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks (Long Version). Technical Report 2008/07, TU Berlin, Fak. IV (2008), <http://iv.tu-berlin.de/TechnBerichte/2008/2008-07.pdf>

A Formal Connection between Security Automata and JML Annotations^{*}

Marieke Huisman¹ and Alejandro Tamalet²

¹ University of Twente, Netherlands

² University of Nijmegen, Netherlands

Abstract. Security automata are a convenient way to describe security policies. Their typical use is to monitor the execution of an application, and to interrupt it as soon as the security policy is violated. However, run-time adherence checking is not always convenient. Instead, we aim at developing a technique to verify adherence to a security policy statically. To do this, we consider a security automaton as specification, and we generate JML annotations that inline the monitor – as a specification – into the application. We describe this translation and prove preservation of program behaviour, *i.e.*, if monitoring does not reveal a security violation, the generated annotations are respected by the program.

The correctness proofs are formalised using the PVS theorem prover. This reveals several subtleties to be considered in the definition of the translation algorithm and in the program requirements.

1 Introduction

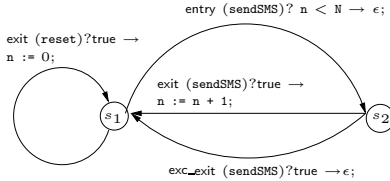
With the emergence of a new generation of trusted personal devices (mobile phones, PDAs, *etc.*), the demand for techniques to guarantee application security has become even more prominent. A common approach is to monitor executions with a security automaton [13]. Upon entry or exit of a security-critical method, the security automaton updates its internal state. If it reaches an “illegal” state, the application will be stopped and a security violation will be reported. This approach is particularly suited for properties that are expressed as sequences of legal method calls, such as life cycle properties, or constraints that express how often or under which conditions a method can be called. However, such a monitoring approach is not suited for all applications, depending on their nature and use; sometimes statical means to enforce security are necessary.

Security experts typically express security requirements by a collection of security automata or temporal logic formulae. However, many program verification tools use a Hoare logic style for the specifications (*i.e.*, pre- and postconditions). Therefore, as a first step towards static verification of such security properties, this paper proposes a translation from security properties expressed as an automaton into program annotations.

^{*} This work is partially funded by the IST FET programme of the European Commission, under the IST-2005-015905 *Mobius* project. Research done while the authors where at INRIA Sophia Antipolis.

The translation in this paper is defined for Java programs. It is defined in several steps. For each step we provide a correctness proof. (i) We translate a *partial automaton* to a *total automaton* that contains a special trap state to model that an error has occurred. We show that the behaviour of a program monitored with a partial automaton is equivalent to the behaviour of the program monitored with the total automaton. (ii) Using an extension of JML [9], we generate annotations that capture the behaviour of the total automaton. These are special method-level set-annotations that are evaluated upon entry or exit of a method. We show that run-time monitoring of the program only throws a (new) exception to signal an annotation violation if the monitor reaches the trap state, otherwise the annotated program has the same executions as the monitored program. (iii) We inline the set-annotations from the method specification to the method body and prove equivalence of the run-time checking behaviour. All results in the paper have been established formally using the PVS theorem prover [11]. The complete formalisation is available via <http://www.cs.ru.nl/tamalet/>.

To prove correctness, the order in which method specifications are evaluated is important. Further, we had to add an explicit requirement that **finally** blocks could not override annotation violation exceptions thrown inside **try** or **catch** statements (see also [8]). The last complication that we encountered was how to specify conveniently that specification-only constructs and steps taken by the monitor did not have any side-effects on the program state. More detailed information about the proofs is given in Section 4.



Automaton vars = $\{n\}$ Program vars = $\emptyset$

Fig. 1. Example Property Automaton

reset should not be called from within **sendSMS**. Even though very basic, this example is representative of a wide range of important resource-related security properties.

The rest of this paper is organised as follows. Section 2 formalises the automaton format and defines completion. Next, Section 3 defines the semantics of monitored and annotated programs. Section 4 defines the translation and proves correctness. Sections 5 and 6 discuss related and future work and conclusions.

2 Modelling Security Properties with Automata

The automata that we use to express security properties are called Property Automata (PA). These are extended finite state machines particularly suited for monitoring, since transitions do not only depend on the automaton's state (*i.e.*,

Throughout this paper, we use the *limited SMS* example property of Fig. 1 (where ϵ denotes a skip) to illustrate the different translations: the method **sendSMS** can be called and terminate successfully at most N times in between calls to **reset**. The counter is not increased if **sendSMS** terminates because of an uncaught exception (with label **exc_exit(sendSMS)**), and

the current control point and a valuation for the automaton's variables), but also on the state of the monitored program. Transitions are labelled with guards, events and a list of actions. Events specify the method whose entry and/or exit is being monitored, with a distinction between normal and exceptional exits. Guards describe the conditions under which a transition can be applied. They depend on (i) the automaton state, (ii) the state of the program that is being monitored, and (iii) the argument of the method, in case the event is method entry; the result of the method, in case the event is normal method exit; or the exception with which the method returns, in case the event is exceptional method exit. Actions describe how the automaton state is updated by a transition.

Throughout, we assume that CP and $\mathcal{N}$ are possibly infinite, but countable non-empty sets of control points and names. PA and programs share the definitions of values, types and exceptions, denoted $\mathcal{V}$, $\mathcal{T}$ and $\mathcal{E}$, respectively. These are defined by the following grammar, where $\mathbb{B}$ and $\mathbb{Z}$ denote the standard sets of booleans and integers, respectively¹.

$$\begin{aligned}\mathcal{V} &= \mathbb{B}(b : \mathbb{B}) \mid \mathbb{I}(i : \mathbb{Z}) \mid \text{Null} \mid \mathbb{R}(i : \mathbb{Z}) \mid \mathbb{1} \mid \perp \\ \mathcal{T} &= \text{Bool} \mid \text{Int} \mid \text{Ref} \mid \text{Void} \\ \mathcal{E} &= \text{Throwable} \mid \text{RunTimeException} \mid \text{JMLException}\end{aligned}$$

The type **Void**, inhabited by $\mathbb{1}$, models methods without results; a reference can be **Null** or contain a number representing the location where the object is stored; $\perp$ is used to denote the outcome of an expression whose evaluation is undefined (in Java this would typically result in an exception).

A PA consists of (i) a name, (ii) a class name, to specify which class is being monitored, (iii) a finite set of control points, (iv) an initial control point, (v) a set of events, to specify which methods are being monitored, (vi) a set of PA variable declarations, to describe the internal state of the automaton, (vii) a set of program variable declarations, to specify which program variables will be inspected by the monitor, and (viii) a set of transitions. Transitions relate source and target control points; they are labelled with events, a guard and a list of actions. An event is a tuple of an event type (entry, exit or exceptional exit), and a method name. Each action assigns the result of an expression (containing both program and PA variables) to a PA variable. Notice that we only monitor classes here. This is often the case in practice, because security-critical methods are often static API methods. However, a more precise formalisation of Java's semantics would allow to monitor objects as well. Formally, a PA is defined as follows.

$$\begin{aligned}\text{Decl} &= [\# \text{ type} : \mathcal{T}, \text{ name} : \mathcal{N}, \text{ init} : \mathcal{V} \#] \\ \text{Event} &= [\# \text{ etype} : (\text{entry} \mid \text{exit} \mid \text{exc\_exit}), \text{ mname} : \mathcal{N} \#] \\ \text{Trans} &= [\# \text{ source}, \text{ dest} : CP, \text{ event} : \text{Event}, \text{ action} : ([\# \text{ target} : \mathcal{N}, \text{ expr} : \text{Expr} \#])^*, \\ &\quad \text{ guard} : \text{PAState} \times \text{PState} \times (\mathcal{V} \mid \mathcal{E}) \rightarrow \mathbb{B} \#] \\ \text{PA} &= [\# \text{ name}, \text{ cname} : \mathcal{N}, \text{ cps} : \mathcal{P}(CP), \text{ init} : CP, \text{ events} : \mathcal{P}(\text{Event}), \\ &\quad \text{ pa\_var\_decl} : \mathcal{P}(\text{Decl}), \text{ prog\_var\_decl} : \mathcal{P}(\text{Decl}), \text{ trans} : \mathcal{P}(\text{Trans}) \#]\end{aligned}$$

¹ We will use a PVS-like notation to declare abstract data types and records (enclosed by $[\#$ and $\#]$). Further, if x is a record with field y , we use $x.y$ to access field y , and $x(\#y := z\#)$ to denote the record x with the field y updated to z .

We require a PA to be *deterministic*, *i.e.*, for every source control point and event there is always at most one guard that holds. A PA is *total* if for any source control point and event, there is always a guard that holds; otherwise it is *partial*. Every deterministic PA can be completed into a total one (by function complete): add a special control point **halted**, together with transitions for every control point and every event to **halted**, where the guard is the negation of the disjunction of all other guards for this control point and event. Additionally, add unconditional transitions from **halted** to **halted** for every possible event.

A PA is *wellformed* if: (i) variable names are unique and are not reserved words, (ii) guards do not have side-effects, (iii) guards and actions only use declared variables, and (iv) control points and events in transitions are declared.

The state of a PA consists of a current control point, and the store of automaton variables (the program store is not part of the automaton state): $PAState = [\# \text{current} : CP, \text{store}_A : Store \#]$. Given PA a , the transition function Δ_a specifies how an automaton state σ_A is updated for a given program state σ_P , an event e , and a value or exception v (where ε is the arbitrary choice operator, and **apply** is a function that updates the automaton store according to a list of actions in the obvious way).

$$\begin{aligned} \Delta_a : PAState \times PState \times Event \times (\mathcal{V} \mid \mathcal{E}) &\hookrightarrow PAState \\ \Delta_a(\sigma_A, \sigma_P, e, v) = & \\ \text{let } t = \varepsilon(\{t \in \text{trans}(a) \mid t.\text{source} = \sigma_A.\text{current} \wedge t.\text{event} = e \wedge & \\ t.\text{guard}(\sigma_A.\text{store}_A, \sigma_P.\text{fields}.\text{store}, v)\}) \text{ in} & \\ (\# \text{current} := t.\text{dest}, \text{store}_A := \text{apply}(t.\text{action}, \sigma_A.\text{store}_A, \sigma_P.\text{fields}.\text{store}) \#) & \end{aligned}$$

In a total PA a , the transition function Δ_a is total. A partial automaton gets stuck on a certain input if and only if the completed PA reaches the state **halted**.

$$\Delta_a(\sigma_A, \sigma_P, e, v) = \perp \Leftrightarrow \Delta_{\text{complete}(a)}(\sigma_A, \sigma_P, e, v).\text{current} = \text{halted} \quad (1)$$

Example. The property specified in Fig. 1 is encoded by the following PA², while Fig. 2 shows the completed PA (where new transitions are dashed).

```
(# name := LimitSMS, cname := Messaging, cps := {s1, s2}, init := s1,
  events := {(# etype := e, mname := sendSMS #) | e ∈ {entry, exit, exc_exit}} ∪
             {(# etype := exit, mname := reset #)},
  pa_var_decl := {(# name := n, type := Int, init := 0 #)}, prog_var_decl := ∅,
  trans := { (# source := s1, dest := s2, guard := λ(σA, σP, v).σA(n) < N,
              event := (# etype := entry, mname := sendSMS #) #),
             (# source := s2, dest := s1, action := [(# target := n, expr := n + 1 #)]
              event := (# etype := exit, mname := sendSMS #) #),
             (# source := s2, dest := s1,
              event := (# etype := exc_exit, mname := sendSMS #) #),
             (# source := s1, dest := s1, action := [(# target := n, expr := 0 #)],
              event := (# etype := exit, mname := reset #) #) } #)
```

² Where we leave the default guard $\lambda(\sigma_A, \sigma_P, v).\text{true}$ and empty action ϵ implicit.

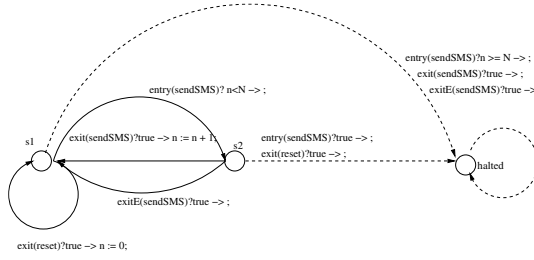


Fig. 2. Automaton of Fig. 1, after completion

$$\begin{aligned}
 Expr &= Plus(n_1, n_2 : Expr) \mid Var_{\mathbb{I}}(n : \mathcal{N}) \mid Not(b : Expr) \mid And(b_1, b_2 : Expr) \mid \\
 &Eq(e_1, e_2 : Expr) \mid Var_{\mathbb{B}}(n : \mathcal{N}) \mid Var_{\mathbb{R}}(n : \mathcal{N}) \mid CondExpr(c, e_1, e_2 : Expr) \mid \\
 &Assign(n : \mathcal{N}, e : Expr) \mid Call(o : Expr, mn : \mathcal{N}, p : Expr) \mid Const(v : \mathcal{V}) \\
 Stmt &= Skip \mid Sequence(s_1, s_2 : Stmt) \mid IfThenElse(c : Expr, s_1, s_2 : Stmt) \mid \\
 &While(c : Expr, s : Stmt) \mid StmtExpr(e : Expr) \mid Throw(e : \mathcal{E}) \mid \\
 &TryCatchFinally(t : Stmt, e : \mathcal{E}, c, f : Stmt) \mid Set(n : \mathcal{N}, e : Expr) \mid \\
 &CaseSet(b : list[Expr \times Stmt]) \mid Assert(e : Expr)
 \end{aligned}$$

Fig. 3. Abstract syntax of expressions and statements

3 Programs and Semantics

This section first defines an abstract syntax of programs, followed by their semantics. Both are fairly standard, except that the semantics is parametrised on the treatment of specifications. In particular, we define a run-time checking and a monitoring semantics, that evaluate differently upon method call and exit.

3.1 Program Syntax

Our language is a restricted subset of (sequential) Java, abstracting away from typical object-oriented features, and in particular from method resolution; instead we assume that the annotated class contains method bodies for the relevant methods, thus method lookup is trivial. We consider only a few exceptions, and assume that methods have only one parameter. We believe, however, that our formalisation contains all constructs that are relevant for proving correctness of our inlining algorithm for class-based monitoring, and implementing the algorithm for the full language is mainly an engineering issue.

Figure 3 defines expressions and statements (we use the term *body* to denote either an expression or a statement), *e.g.*, `Call` represents a call to method mn on target o with argument p . Notice that we define several special language constructs to represent JML annotations: `Set`, to update ghost variables (*i.e.*, specification-only variables), `CaseSet`, to abbreviate a list of conditional ghost variable updates, and `Assert`, to evaluate a condition on the program state. A standard program semantics ignores these statements, whereas the annotated program semantics evaluates them.

$$\begin{aligned}
\textit{Method} &= [\# \text{ name} : \mathcal{N}, \text{ param} : \textit{Decl}, \text{ lvars} : \mathcal{P}(\textit{Decl}), \text{ body} : \textit{Stmt}, \\
&\quad \text{ res} : \textit{Expr}, \text{ res\_type} : \mathcal{T}, \text{ pre}, \text{ post} : \textit{Expr} \rightarrow \textit{Expr}, \\
&\quad \text{ pre\_set}, \text{ post\_set} : \textit{Expr} \rightarrow \textit{Stmt}, \text{ exc\_set} : \mathcal{E} \rightarrow \textit{Stmt} \#] \\
\textit{Class} &= [\# \text{ name} : \mathcal{N}, \text{ super} : \mathcal{N}_\perp, \text{ fields} : \mathcal{P}(\textit{Decl}), \text{ methods} : \mathcal{P}(\textit{Method}), \\
&\quad \text{ inv} : \textit{Expr}, \text{ ghost\_vars} : \mathcal{P}(\textit{Decl}) \#] \\
\textit{Program} &= [\# \text{ classes} : \mathcal{P}(\textit{Class}) \#]
\end{aligned}$$
Fig. 4. Abstract Syntax for Programs

Figure 4 describes the syntax for methods, classes and programs. To ensure that every method has an appropriate return expression, it is part of the method signature. Furthermore, methods can be annotated with pre- and postconditions, and classes with invariants. To support our annotation generation algorithm, we define special annotations called **pre_set**, **post_set** and **exc_set**. These annotations describe the updates to the ghost variables at method entry, exit and exceptional exit, respectively. Pre- and postcondition and the different method specification-level set annotations have a function type to allow the use of the method parameter, the method result, or the returned exception, respectively.

A program is said to be *wellformed* if (i) names of fields, local variables and ghost variables are disjoint and are not reserved words; (ii) class names are unique; (iii) method names are unique; (iv) every variable name that is used is declared; and (v) only ghost variables are the target of **Set** statements.

3.2 Natural Semantics

The behaviour of a program is described via a big step semantics. We closely follow Von Oheimb's formalisation of Java [10], with simplifications wherever possible, due to our simplified program syntax. A judgement $P \vdash \langle e, \sigma \rangle \triangleright \langle v, \sigma' \rangle$ means that the body e evaluates to v , while transforming the state σ into σ' , in the context of the program P . Note that v is $\mathbb{1}$ for normally terminating *statements*, while v is $\perp$ whenever evaluation finishes in an exceptional state.

A basic program state $PState$ is composed of an optional exception and a store. The store maps every field and local variable to a value.

$$\begin{aligned}
PState &= [\# \text{ exc} : \textit{Excp}_\perp, \text{ store} : PStore \#] \\
PStore &= [\# \text{ fields} : \mathcal{N} \mapsto \mathcal{V}, \text{ loc\_vars} : \mathcal{N} \mapsto \mathcal{V} \#]
\end{aligned}$$

Since annotated or monitored programs contain more information than unannotated programs, the evaluation rules are parametrised with types *FullProgram* and *FullState*. For each instantiation we give mappings **program** and **prog_state** to the basic program type *Program* and the basic program state *PState*. Further, we add parameters that specify the actions that are taken upon method entry or (normal or exceptional) exit (γ_{IN} , γ_{NORM} , and γ_{EXC} , respectively), and the handling of annotations (δ_{SET} , δ_{ASSERT} , and δ_{CASE}). In a standard program semantics, where specifications are ignored, these are all instantiated with the identity relation.

The evaluation rules are fairly standard, and we refer to Von Oheimb and the PVS formalisation for more details. Evaluation of normally terminating method

calls is described by the following rule (where for clarity of presentation, we left out several checks that intermediate states are not exceptional).

$$\begin{array}{c}
\sigma_0.\text{prog\_state.exc} = \perp \quad P \vdash \langle o, \sigma_0 \rangle \triangleright \langle r, \sigma_1 \rangle \quad P \vdash \langle p, \sigma_1 \rangle \triangleright \langle \text{act}, \sigma_2 \rangle \\
r \neq \text{Null} \quad md = \text{lookup\_mthd}(P, r, mn) \\
old\_lvs = \sigma_2.\text{prog\_state.store.loc\_vars} \quad \sigma_3 = \text{update\_lvs}(\sigma_2, r, md.\text{lvars}, md.\text{param}, \text{act}) \\
\gamma_{\text{IN}}(P, md, r, \text{Const}(\text{act}), \sigma_3, \sigma_4) \quad P \vdash \langle md.\text{body}, \sigma_4 \rangle \triangleright \langle \mathbb{1}, \sigma_5 \rangle \\
P \vdash \langle md.\text{res}, \sigma_5 \rangle \triangleright \langle v, \sigma_6 \rangle \quad \gamma_{\text{NORM}}(P, md, r, \text{Const}(v), \sigma_6, \sigma_7) \\
\hline
P \vdash \langle \text{Call}(o, mn, p), \sigma_0 \rangle \triangleright \langle v, \sigma_7(\text{prog\_state.store.loc\_vars} := old\_lvs) \rangle
\end{array}$$

First the receiver is evaluated, resulting in non-null reference r . Next, the parameter is evaluated, resulting in value act . Using r , the method definition md is looked up. The local variable store is updated assigning r to **this**, initialising the method's local variables and assigning the actual parameter to the formal parameter. The old local variable store is remembered as old_lvs . Next, an appropriate action upon method entry is taken, as specified by the relation γ_{IN} . Then the method body, and method result expression are evaluated. Since this rule applies to normal method termination only, the parameter for normal method termination γ_{NORM} is evaluated. Last, the local store is set back to old_lvs . In addition, rules exist that specify behaviour of a method call when it is called upon a null reference, the body contains an uncaught exception *etc.*

Annotated Program Semantics. The program state of an annotated program is extended with a store for ghost variables:

$$AState = [\# \text{pstate} : PState, \text{ghost\_vars} : \mathcal{N} \mapsto \mathcal{V} \#]$$

The types *FullProgram* and *Program* coincide, while *FullState* is instantiated as *AState*, and the mapping **prog_state** is defined as **pstate**. Figure 5 shows some of the instantiations of the semantics parameters; the other instantiations are similar. The relation γ_{IN} uses the auxiliary relation β which checks a boolean expression e and raises a special **JMLEException** if it evaluates to false. Upon method entry, the class invariant and precondition are evaluated. We assume that **lookup_inv** returns the complete class invariant, including those invariants that are inherited from superclasses. If they fail, a **JMLEException** is thrown, otherwise the method's **pre_set** statement is executed. Finally, we ensure that the program store is not changed. The function δ_{SET} updates a ghost variable: it first evaluates the expression and if this did not result in an exceptional state, it updates the value of the ghost variable³ appropriately.

Monitored Program Semantics. The parametrised program semantics is also instantiated for monitored programs. This semantics is only defined when the PA is compatible with the program. PA a is said to be compatible with a program P , denoted $a \sqsubseteq P$, if (i) the program contains the class c that is being monitored, (ii) all variables declared as program variables in a are fields of the class

³ Where $\tau(\text{ghost\_vars}.n := v)$ abbreviates that the value of $\text{ghost\_vars}(n)$ in τ is updated to v .

$$\begin{array}{c}
\frac{
\begin{array}{c}
\text{inv} = \text{lookup\_inv}(P, r) \quad \beta(P, \text{inv}, \sigma_1, \tau_1) \quad \beta(P, \text{md.pre}(\text{act}), \tau_1, \tau_2) \\
P \vdash \langle \text{md.pre\_set}(\text{act}), \tau_1 \rangle \triangleright \langle v, \tau_2 \rangle \quad v \in \{\perp, \mathbb{1}\} \quad \sigma_1.\text{pstate.store} = \sigma_2.\text{pstate.store}
\end{array}
}{\gamma_{\text{IN}}(P, \text{md}, r, \text{act}, \sigma_1, \sigma_2)} \\
\\
\frac{
P \vdash \langle e, \sigma_1 \rangle \triangleright \langle v, \tau \rangle \quad \text{if } v = \mathbf{B}(\text{true}) \text{ then } \sigma_2 = \tau \text{ else } \sigma_2 = \tau(\text{exc} := \text{JMLException})
}{\beta(P, e, \sigma_1, \sigma_2)} \\
\\
\frac{
P \vdash \langle e, \sigma_1 \rangle \triangleright \langle v, \tau \rangle \quad \text{if } \tau.\text{pstate.exc} = \perp \text{ then } \sigma_2 = \tau(\text{ghost\_vars}.n := v) \text{ else } \sigma_2 = \tau
}{\delta_{\text{SET}}(P, \text{Set}(e, n), \sigma_1, \sigma_2)}
\end{array}$$

Fig. 5. Instantiation of semantics for runtime annotation evaluation

c with the correct type, and (iii) every event name corresponds to a method in the class. A monitored program is a product of a PA and a program. The state of a monitored program consists of the states of the PA and the program (including ghost variables)⁴, and a flag *stuck*. If the PA is partial, the flag *stuck* is set when Δ_a is not defined for a certain input. If the flag is set, this means that the security policy is violated, and the program should be stopped (by some external observer). If the PA is total, the *stuck* flag will never be set. Instead, violation of the security policy is modelled by the PA reaching the trap state *halted* (in which case the external observer again is supposed to stop execution).

$M\text{Program} = [\# \text{pa} : PA, \text{program} : \text{Program} \#]$

$M\text{State} = [\# \text{pa\_state} : PA\text{State}, \text{pstate} : P\text{State}, \text{ghost\_vars} : \mathcal{N} \mapsto \mathcal{V}, \text{stuck} : \mathbb{B} \#]$

Thus, *FullProgram* gets instantiated as *MProgram* and *FullState* as *MState*, with mappings *program* and *pstate*. Now we can give appropriate instantiations for the γ - and δ -relations. The δ -relations are the same as for the annotated program semantics, but the γ -relation also updates the state of the monitor. For example, γ_{IN} is defined in terms of γ_{IN} for annotated programs, as defined in Fig. 5.

$$\frac{
\gamma_{\text{IN}}^{\text{AP}}(P, \text{md}, r, \text{act}, \sigma_1, \tau)
}{
\text{if } \tau.\text{pstate.exc} = \perp \text{ then } \sigma_2 = \gamma_{\text{PA}}(\text{entry})(P, \text{md}, \text{act}, \tau) \text{ else } \sigma_2 = \tau
}
\gamma_{\text{IN}}(P, \text{md}, r, \text{act}, \sigma_1, \sigma_2)$$

where

$$\begin{aligned}
\gamma_{\text{PA}}(ev)(P, \text{md}, \text{act}, \sigma) = & \text{let } e = (\# \text{etype} := ev, \text{mname} := \text{md.name} \#), \\
& \tau = \Delta_{P, \text{pa}}(\sigma.\text{pa\_state}, \sigma.\text{prog\_state}, e, \text{act}) \text{ in} \\
& \text{if } \sigma.\text{stuck} \vee \tau = \perp \text{ then } \sigma(\text{stuck} := \text{true}) \text{ else } \sigma(\text{pa\_state} := \tau)
\end{aligned}$$

4 Annotation Generation

Given a security property encoded as a PA, the annotation generation procedure generates JML-annotations that capture this property, *i.e.*, if the program respects the property encoded by the monitor, then it does not violate the generated annotations. As explained above, the procedure is defined in several steps:

⁴ For convenience, we assume that a monitored program also evaluates annotations, but this instantiation is in fact orthogonal to the annotated program semantics.

(i) the monitor is completed; (ii) the annotations are generated at the method specification level, as special set-annotations; and (iii) the method specification-level set-annotations are inlined in the method body. Notice that the special `CaseSet` annotation could be translated into standard JML annotations as well.

For each step we prove that the new program simulates the old program, *i.e.*, we show for every translation step α there exists a relation R such that:

$$\begin{aligned} \forall b, \sigma_1, \sigma_2, \tau_1, v_1. P \vdash \langle b, \sigma_1 \rangle \triangleright \langle v_1, \sigma_2 \rangle \wedge R(\sigma_1, \tau_1) \Rightarrow \\ \exists \tau_2, v_2. \alpha(P) \vdash \langle b, \tau_1 \rangle \triangleright \langle v_2, \tau_2 \rangle \wedge R(\sigma_2, \tau_2) \end{aligned}$$

Additionally, we show that the initial program states are related by R , and from this we can conclude that for any reachable state of the monitored program, there exists a related state, reachable in the translated program. As a side-remark, for translation steps (i) and (iii), we can even prove that relation R is a bisimulation, while for step (ii) we can only prove existence of a simulation (since non-terminating monitored programs – for which no derivation exists in the natural semantics – might terminate after annotation generation, because of an annotation violation).

A natural way to prove the simulation is by induction over the derivation length. However, induction can only be applied when the body is unchanged. Since the translation introduces new (ghost) variables to encode the PA, this is not always the case. For these cases, separate preservation lemmas have to be proven. Further, to be able to complete the proof, we need to ensure that in both bodies the same branches of conditional expressions and statements are taken, and that the same values get assigned to the store. Therefore, we prove a stronger result, adding that also the values v_1 and v_2 are the same (for step (ii): provided the monitor did not reach the `halted` state).

Completion of the automaton. The first translation step does not change the program itself, it only completes the PA. Suppose that P is a monitored program, where the monitor $P.\text{pa}$ is deterministic and wellformed. Then the translation to a monitored program with a total PA, $\alpha_1(P)$, is defined as:

$$\alpha_1(P) = (\# \text{ pa} := \text{complete}(P.\text{pa}), \text{program} := P.\text{program} \#)$$

The relation that is preserved between executions of P and $\alpha_1(P)$ is the following (where σ is a state of P and τ is a state of $\alpha_1(P)$):

$$\begin{aligned} R(\sigma, \tau) = & (\text{if } \sigma.\text{stuck} \text{ then } \tau.\text{pa\_state.current} = \text{halted} \\ & \text{else } \sigma.\text{pa\_state.current} = \tau.\text{pa\_state.current}) \wedge \neg \tau.\text{stuck} \wedge \\ & (\sigma.\text{pa\_state.store}_A = \tau.\text{pa\_state.store}_A) \wedge \\ & (\sigma.\text{pstate} = \tau.\text{pstate}) \wedge (\sigma.\text{ghost\_vars} = \tau.\text{ghost\_vars}) \end{aligned}$$

To prove that this relation is preserved for any body b , we use equivalence (1) on Page 343 and we observe further that (i) if `stuck` has been set, it remains set, (ii) for a total PA, if `halted` is reached, it is never left, and (iii) for a total PA, `stuck` is never set. Formally, where P is a monitored program, and Q is a monitored program with total PA:

$$\begin{aligned}
\alpha_2(P) &= (\# \text{ classes} := \{\alpha_{2,C}(c, P.\text{pa}) \mid c \in P.\text{program.classes}\} \#) \\
\alpha_{2,C}(c, a) &= \text{if } c.\text{name} \neq a.\text{cname} \text{ then } c \\
&\quad \text{else } c \text{ (} \# \text{ ghost\_vars} := c.\text{ghost\_vars} \cup \text{new\_vars}(a) \\
&\quad \quad \text{inv} := \text{And}(\text{Not}(\text{Eq}(\text{cp}, \text{halted})), c.\text{inv}) \\
&\quad \quad \text{methods} := \{\alpha_{2,M}(m, a) \mid m \in c.\text{methods}\} \#) \\
\alpha_{2,M}(m, a) &= m \text{ (} \# \text{ pre\_set} := m.\text{pre\_set}; \alpha_{2,E}(\text{entry}, m.\text{name}, a); \\
&\quad \quad \text{Assert}(\text{Not}(\text{Eq}(\text{cp}, \text{halted}))), \\
&\quad \quad \text{post\_set} := m.\text{post\_set}; \alpha_{2,E}(\text{exit}, m.\text{name}, a) \\
&\quad \quad \text{exc\_set} := m.\text{exc\_set}; \alpha_{2,E}(\text{exc\_exit}, m.\text{name}, a) \#) \\
\alpha_{2,E}(e, n, a) &= \alpha_{2,T}(\{t \mid t \in a.\text{trans} \wedge t.\text{etype} = (\# \text{ event} := e, \text{mname} := m \#)\}, a) \\
\alpha_{2,T}(ts, a) &= \text{CaseSet}(\{(\text{Eq}(\text{cp}, q), \alpha_{2,S}(ts, q)) \mid q \in a.\text{cps}\}) \\
\alpha_{2,S}(ts, q) &= \text{CaseSet}(\{(t.\text{guard}, \text{Set}(\text{cp}, t.\text{dest}; t.\text{action})) \mid t \in ts \wedge t.\text{source} = q\})
\end{aligned}$$

Fig. 6. Formal definition of translation PA into annotations

$$\begin{aligned}
&\sigma_1.\text{stuck} \wedge P \vdash \langle b, \sigma_1 \rangle \triangleright \langle v, \sigma_2 \rangle \Rightarrow \sigma_2.\text{stuck} \\
&\sigma_1.\text{pa\_state.current} = \text{halted} \wedge Q \vdash \langle b, \sigma_1 \rangle \triangleright \langle v, \sigma_2 \rangle \Rightarrow \sigma_2.\text{pa\_state.current} = \text{halted} \\
&\neg \sigma_1.\text{stuck} \wedge Q \vdash \langle b, \sigma_1 \rangle \triangleright \langle v, \sigma_2 \rangle \Rightarrow \neg \sigma_2.\text{stuck}
\end{aligned}$$

To illustrate how the annotation generation algorithm works on the LimitSMS automaton in Fig. 1, assume we have declared a class **Messaging**, containing the methods used by the automaton, plus a method **receiveSMS**. Applying translation α_1 means that this class, instead of being monitored by the partial PA in Fig. 1, is monitored by the total PA in Fig. 2.

From PA to Annotations. Figure 6 contains the formal definition of the second translation step: from PA to method-level set-annotations. Given a monitored program P where $P.\text{pa}$ is total, the annotation generation algorithm α_2 applies $\alpha_{2,C}$ to all classes. This function checks whether the class is the one being monitored. If so, appropriate ghost variables are added to the class using the function **new_vars** (see the PVS formalisation for its formal definition). Basically (i) for each automaton control point q , a (final) ghost variable declaration **q** is generated, initialised to a unique value; (ii) a ghost variable **cp** is declared, initialised to the value of the ghost variable representing the initial control point; and (iii) for each automaton variable declaration, a ghost variable is declared with corresponding name, type and initialisation. Further, $\alpha_{2,C}$ adds the condition that the current control point should not be **halted** to the class invariant⁵, and it annotates all methods in the class using $\alpha_{2,M}$. For each method, **pre_set**, **post_set** and **exc_set** are extended with updates to the ghost variables encoding the automaton. In addition, at the end of **pre_set**, an **Assert** statement is added to verify whether the transition reached the **halted** state: in that case program execution should terminate immediately. Without this **Assert**, the property violation would only be detected after the body is executed. To encode the updates

⁵ For readability, we do not explicitly write the translation from PA control points to ghost variables.

```

class Messaging {
  //@ ghost int halted = 0, s1 = 1, s2 = 2, N = 5, cp = s1, n = 0;
  //@ public invariant cp != halted;

  /*@ pre_set   CaseSet [(cp == s1, CaseSet [(n < N, cp = s2),
                                              (n >= N, cp = halted)]),
                        (cp == s2, CaseSet [(true, cp = halted)]),
                        (cp == halted, CaseSet [(true, cp = halted)]]];
    Assert cp != halted;
  post_set CaseSet [(cp == s1, CaseSet [(true, cp = halted)]),
                    (cp == s2, CaseSet [(true, cp = s1; n = n + 1)]),
                    (cp == halted, CaseSet [(true, cp = halted)]]];
  exc_set  CaseSet [(cp == s1, CaseSet [(true, cp = halted)]),
                    (cp == s2, CaseSet [(true, cp = s1)]),
                    (cp == halted, CaseSet [(true, cp = halted)]]]; @*/

void sendSMS(){/* body sendSMS */} /*@ pre_set

CaseSet []; Assert cp != halted;
  post_set CaseSet []; exc_set CaseSet []; @*/
void receiveSMS(){/* body receiveSMS */} /*@

pre_set CaseSet []; Assert cp != halted; exc_set CaseSet [];
  post_set CaseSet [(cp == s1, CaseSet [(true, cp = s1; n = 0)]),
                    (cp == s2, CaseSet [(true, cp = halted)]),
                    (cp == halted, CaseSet [(true, cp = halted)]]]; @*/
void reset() {/* body reset */} }

```

Fig. 7. Method-level set annotations generated for class Messaging

to the ghost variables, $\alpha_{2,\mathcal{E}}$ computes the set of relevant transitions (*i.e.*, those where the event and method name correspond). For these transitions, a `CaseSet` statement is generated, where the different cases correspond to the current control point being equal to a control point q , for any q in the automaton. For each such q , $\alpha_{2,\mathcal{S}}$ selects the transitions where $t.source$ is q and generates a `CaseSet` statement, that tests whether the guard holds, and if so, sets the control point cp to $t.dest$, and executes the actions associated with this transition. Notice that the order in which the different cases are generated is not important: since the PA is total and deterministic there is always exactly one case that applies.

The formalisation does not specify how guards and actions are translated. Instead, we assume there exists a translation into expressions in the programming language that (i) are wellformed, (ii) give the same result, (iii) do not have side-effects, (iv) do not throw exceptions, and (v) do not contain method calls. From this we can derive that in the annotated program, the generated statements in `pre_set` can only throw a `JMLEException` (because of the concluding `Assert`), while the generated statements in `post_set` and `exc_set` do not throw any exception.

As an example, consider again the class `Messaging` and the completed PA, encoding the *limited SMS* policy, in Fig. 2. Figure 7 shows the generated annotations that result from applying translation $\alpha_{2,\mathcal{C}}$ on this class and this PA. Notice that for methods and events that are not involved in the property, an empty `CaseSet` is generated – this is equivalent to a `Skip` statement.

To show correctness of the translation, we show that the following relation is preserved (where P is the monitored program, σ a state of the monitored program, and τ a state of the annotated program):

$$\begin{aligned}
 R(\sigma, \tau) &= \neg \sigma.\text{stuck} \wedge \\
 &\quad \text{if } \sigma.\text{pa\_state.current} = \text{halted} \text{ then } \tau.\text{pstate.exc} = \text{JMLException} \text{ else } S(\sigma, \tau) \\
 S(\sigma, \tau) &= (\text{unique}(\sigma.\text{pa\_state.current}) = \tau.\text{ghost\_vars}(\text{cp})) \wedge \\
 &\quad \forall q \in P.\text{pa.cps}. (\text{unique}(q) = \tau.\text{ghost\_vars}(q)) \wedge \\
 &\quad \forall n \in \mathcal{N}. (\sigma.\text{pa\_state}(n) \neq \perp \Rightarrow \sigma.\text{pa\_state}(n) = \tau.\text{ghost\_vars}(n)) \wedge \\
 &\quad \sigma.\text{pstate} = \tau.\text{pstate} \wedge \\
 &\quad \forall n \in \mathcal{N}. (\sigma.\text{ghost\_vars}(n) \neq \perp \Rightarrow \sigma.\text{ghost\_vars}(n) = \tau.\text{ghost\_vars}(n))
 \end{aligned}$$

This relation specifies that if the monitor has reached control point **halted**, the annotated program must have thrown a **JMLException**. Otherwise, the state of the annotated program corresponds to the state of the original program, extended with the modelling of the monitor's state. This means that the program states (fields, local variables and exceptions) have to coincide, just as the values of the ghost variables that are declared in the original program P . Further, the current control point is represented by the value stored in ghost variable **cp**, and all PA control points and variables correspond to ghost variables. Notice that if an annotation already present in P causes a **JMLException**, both the monitored and the annotated program will throw it. Therefore, we cannot prove that the annotated program throws a **JMLException** *if and only if* **halted** is reached.

To prove that this relation is a preserved, it is strengthened with the following property: if the control point is not **halted**, then the derivations also produce the same value. The crucial part in the proof is of course what happens upon method call and termination. For example, when a method is called, first the invariant and the precondition are evaluated. Assuming that **halted** is not yet reached, the new conjunct of the invariant evaluates to true, and induction allows to derive that after evaluation of the precondition, the states are related by R . Next, the original **pre_set** annotations are evaluated, and again the induction hypothesis allows to conclude that the resulting states are related. Next, the monitored program makes a PA transition, and the annotated program executes the newly generated set annotations, followed by an **Assert** to check whether **halted** has been reached. Here we cannot use the induction hypothesis, but instead we show manually that relation R is preserved. Notice that in **post_set** or **exc_set** we do not have an **Assert** statement. Since the invariant is evaluated immediately after the set-annotations, the reaching of **halted** will be detected immediately. For this part of the proof it is crucial that the newly added invariant is evaluated first.

Finally, to complete the proof, we have to add a restriction to programs. We follow the Java Language Specification in describing its behaviour [6]. This means in particular that if the *finally* block in the statement terminates abnormally (because of an exception, or any other reason for abrupt completion), it overrides a possible exception thrown in the *try* or *catch* block. Thus, for example, if **halted**

$$\alpha_{3,\mathcal{M}}(P, m) = m(\# \text{ pre\_set} := \text{Skip}, \text{post\_set} := \text{Skip}, \text{exc\_set} := \text{Skip}, \\ \text{lvars} := \{\text{result}\} \cup m.\text{lvars}, \text{res} := \text{lookup}(\text{result}), \\ \text{body} := \text{TryCatchFinally}(\\ \quad \text{TryCatchFinally}(m.\text{pre\_set}; m.\text{body}; \\ \quad \quad \text{Assign}(\text{result}, m.\text{res}); m.\text{post\_set} \\ \quad \text{Throwable}, m.\text{exc\_set}, \text{Skip}), \\ \text{RunTimeException}, m.\text{exc\_set}, \text{Skip}) \#)$$

Fig. 8. Formal definition of annotation inlining for methods

is reached in the *try* block, and hence a `JMLException` is thrown, this exception might be overwritten by an exception thrown in the *finally* block (see also [8] for a discussion of this problem), which would mean that the violation of the security policy is not signalled to the user, and instead execution continues (with another exception). To avoid this, for all `TryCatchFinally` statements in the program, we require that if the *try* or *catch* block throws a `JMLException`, the whole statement also terminates exceptionally because of a `JMLException`.

Inlining the Annotations. Once the set-annotations at method specification level are generated, the next step is to inline them into the method bodies. To ensure that the appropriate set-statements are always executed at the end of the method body, the body is wrapped in a `TryCatchFinally` statement. The translation α_3 applies $\alpha_{3,c}$ to all classes, which in turn applies $\alpha_{3,\mathcal{M}}$ to all methods in the class. This function generates one new local variable⁶ `result`. The body of the method is changed as follows: all code is wrapped in two `TryCatchFinally` statements, to catch `Throwable` and `RunTimeException` exceptions⁷. In the *try* block, first `pre_set` is executed, followed by the body of the method. Then the result expression from the original body is evaluated, and assigned to `result`. Next, `post_set` is executed. Notice that the latter is only executed if the body actually terminates normally, otherwise the exception will simply be propagated. Finally, in the *catch* clauses, `exc_set` is executed. The new result expression of the method is the look up of the variable `result`. To conclude, `pre_set`, `post_set` and `exc_set` in the method specification are set to `Skip`. Figure 8 gives the formal definition of $\alpha_{3,\mathcal{M}}$ (where P is a program, and m a method).

To prove correctness of this translation, we use the following relation: all fields and ghost variables coincide, exceptions coincide, and all local variables that are declared in the original program coincide. In the correctness proof, we use that the `post_set` and `exc_set` annotations do not throw any exceptions, and `pre_set` may only throw a `JMLException`. Moreover, we use that the set-annotations do not contain method calls, from which we can conclude that they do not modify any variables that are not explicitly mentioned in them. In particular, this allows to conclude that the new local variable is not changed by the set annotations.

⁶ In fact, this should be a local *ghost* variable, but these are not yet supported by our formalisation, therefore we formalise it as a standard local variable.

⁷ For simplicity, we do not model the exception hierarchy and thus `TryCatchFinally` can only catch a single exception, but in practice only one *try-catch-finally* instruction would be necessary.

5 Related Work

Security automata [13] are widely used for monitoring security properties. The originality of our work lies in considering them as specifications in a general specification language, with the ultimate goal of static verification.

Closely related to our approach is work by Aktug *et al.* [3,1,2], who define a formal language for security policy specifications, ConSpec, that is similar to our PAs. They prove that a monitor can be inlined into the program's bytecode, by adding first-order logic annotations, and then they use a weakest precondition computation that essentially works the same as the annotation propagation algorithm that we plan to use [12] to produce a fully annotated, verifiable program. In contrast, our algorithm is defined for source code, and connects with the general-purpose specification language JML. This allows the use of JML verification tools, to verify the actual policy adherence. And of course, correctness of our inlining algorithm has been proven with a theorem prover.

Cheon and Perumendla propose an extension of JML to specify allowed sequences of methods calls in a regular expression-like notation [4]. This results in succinct specifications, but of limited expressiveness. Even our Limited SMS example is out of their scope, because it contains a counter used only by the specification. Further, they only target runtime verification.

Several tools exist that translate temporal properties into JML annotations: AutoJML [7] translates finite state machine specifications into JML annotations and can also generate a code skeleton for a smart card applet; JAG [5] translates properties in (a subset of) temporal logic, including liveness properties. However, they typically do not distinguish between method entry and exit, and moreover, correctness of the translation algorithm has not been proven.

For more information about policy languages, monitor inlining and specifying policy adherence, we refer to Section 4.10 of Aktug's thesis [1].

6 Conclusions and Future Work

This paper presents an algorithm to inline security automata, in the form of JML annotations. The translation is defined in several steps, thanks to the introduction of method-level set-annotations as extension to JML. All steps are formalised and proven correct, using the PVS theorem prover. The algorithm might seem trivial, but several subtleties complicate the proof, *i.e.* evaluating the specifications in the right order, dealing with side-effect-freeness of annotations and the possibility that a *finally* block hides exceptions.

The formalisation has been developed for a subset of Java. We believe that extending it to full (sequential) Java would be relatively straightforward. However, generalising to properties that are not restricted to a single class or that are related to multithreading might be more challenging.

The ultimate goal of our work is to statically verify adherence to security policies. To achieve this, a weakest precondition calculus can be used to generate pre- and postconditions, based on the generated **Set** annotations. In earlier work,

we presented such a propagation algorithm [12], and proved correctness for a limited case (instance variables and branches are not considered). It is future work to overcome these limitations.

Acknowledgements. We thank Erik Poll for his useful comments on an earlier draft of this paper, and Igor Siveroni, who started the work on this topic and came up with the idea to use method-level set-annotations.

References

1. Aktug, I.: Algorithmic Verification Techniques for Mobile Code. PhD thesis, Royal Institute of Technology (KTH), Sweden (2008)
2. Aktug, I., Dam, M., Gurov, D.: Provably correct runtime monitoring. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 262–277. Springer, Heidelberg (2008)
3. Aktug, I., Naliuka, K.: ConSpec: A Formal Language for Policy Specification. In: Run Time Enforcement for Mobile and Distributed Systems (REM 2007). Electronic Notes in Theoretical Computer Science, vol. 197-1, pp. 45–58 (2007)
4. Cheon, Y., Perumendla, A.: Specifying and Checking Method Call Sequences of Java Programs. *Software Quality Journal* 15, 7–25 (2007)
5. Giorgetti, A., Gros Lambert, J.: JAG: JML Annotation Generation for verifying temporal properties. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 373–376. Springer, Heidelberg (2006)
6. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. The Java Series. Addison-Wesley, Reading (2005)
7. Hubbers, E., Oostdijk, M., Poll, E.: From finite state machines to provably correct Java Card applets. In: Gritzalis, D., De Capitani di Vimercati, S., Samarati, P., Katsikas, S.K. (eds.) IFIP Information Security Conference, pp. 465–470. Kluwer Academic Publishers, Dordrecht (2003), <http://autojml.sourceforge.net>
8. Huisman, M.: Run-time verification can miss errors - why finally clauses can be dangerous (manuscript, 2008)
9. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Kiniry, J.: JML Reference Manual. In: Progress. Department of Computer Science, Iowa State University. (July 2005), <http://www.jmlspecs.org>
10. von Oheimb, D.: Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic. PhD thesis, Technische Universität München (2001)
11. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K.: PVS: Combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
12. Pavlova, M., Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L.: Enforcing high level security properties for applets. In: Quisquater, J.-J., Paradinas, P., Deswarte, Y., El Kalam, A.A. (eds.) Cardis 2004, pp. 1–16. Kluwer Academic Publishers, Dordrecht (2004)
13. Schneider, F.B.: Enforceable security policies. Technical Report TR99-1759, Cornell University (October 1999)

Algorithms for Automatically Computing the Causal Paths of Failures

William N. Sumner and Xiangyu Zhang

Department of Computer Science, Purdue University
`{wsumner,xyzhang}@cs.purdue.edu`

Abstract. We have proposed an automated debugging technique that explains a failure by computing its causal path leading from the root cause to the failure. Given a failing execution, the technique first searches for a dynamic patch. Fine-grained execution comparison between the failing run and the patched run is performed to isolate the causal path. The comparison is enabled by precisely aligning the two executions. We herein propose and study two algorithms aiming at efficiency. We also evaluate the effectiveness and cost of our technique on a set of real bugs, including requirement bugs in which no a single or small set of statements can be blamed as the root cause. In such cases, understanding a failure is more important.

Keywords: debugging, automated debugging, execution indexing.

1 Introduction

During debugging, developers often have a “correct” oracle execution in mind with which they compare a failing execution to identify faulty state and then understand the failure. We have proposed an automated debugging technique that mimics such a procedure [1]. Our technique computes the causal path of a failure, a subsequence of the failing run that explains the failure. It first searches for a dynamic patch for the failing run. The patch mutates the values of certain variables or control flow at runtime to produce the desired output. If such a patch can be found, which is true in most cases, the technique aligns the failing run and the patched run by establishing a mapping between instruction instances in the two runs. The states at aligned points are compared to identify faulty variables, and causality testing is performed to isolate subsets of faulty variables that are essential. The sequence of essential faulty states explains the failure.

The technique follows the direction pioneered by Zeller et al. in [2, 3]. In their work, the idea is to isolate the state transitions that are critical for a failure by *comparing the faulty execution with a similar but correct execution*. Compared to their work, we made progress in the following directions. *First*, we found that state comparisons need to be performed at rigidly corresponding points in the two respective executions. Due to the differences between the two executions, construction of such correspondence is challenging. In [2, 3], it was carried out in an ad-hoc way such that points that are not compatible may be

selected for comparison. As a result, the computed chain may not be relevant to the failure. We proposed using *execution indexing* (EI) [4] to align the two executions before comparison. EI is a technique that constructs a hierarchical tree of an execution based on program structure. Executions can be aligned through their trees. *Second*, our experience shows that using a different execution as the reference oracle often fails to explain the causality of the failure, as the reference execution is semantically different from the failing execution. Thus, comparing these two executions often exposes their inherent semantic differences instead of causality information for the failure. Our solution is to use a patched run derived from the same failure inducing input for comparison.

In our prior work [1], we have built a formal model of the technique, proposed an objective evaluation metric, and evaluated the proposed technique using the SIR suite [5]. Results show that our technique can produce high quality causal paths that often capture the root cause at the beginning, the failure point at the end, and causality information between consecutive steps. Comparison with the technique in [3] shows that our approach provides much better failure explanations. Details can be found in [1].

In this paper, we make the following contributions.

- We propose and study two algorithms that focus on cost-effectiveness. Both algorithms compute the same causal paths but differ in their efforts for reducing the number of state comparisons and causality tests. The first algorithm relies on the execution index tree to compute causal paths in a hierarchical manner. The second algorithm speculatively takes shortcuts during causal path derivation based on program dependence information.
- We evaluate our approach on a set of real world bugs collected from the Internet. Results show that our technique is effective in explaining failures. The shortcutting algorithm is orders of magnitude faster than the hierarchical algorithm.
- We use concrete examples to explicitly explain the unique features of our technique, including using EI to align executions before comparison and using a patched execution rather than a different execution as the reference.

2 Background

The goal of our technique is to compute the causal path of a failure. Assume we have the corrected version of the faulty program. The *ideal* causal path of a failure is computed by comparing the failing execution and the execution of the corrected program with the same input. The comparison is done by differencing the states at corresponding points in the two respective runs, starting from the failure point and proceeding backwards. In particular, faulty variables at a step are determined by comparing their values in the failing run with those in the correct run. Not all faulty variables are essential to the failure. Thus, we define the *failure inducing state* (FIS) of a step in the failing run as the minimal set of faulty variables that cause the failure or the FIS of the next step. Consider the example in Fig. 1. There is a fault at line 10. Provided with the input $a=2$ and

$b=3$, the program produces a failure observed at line 18, i.e. printing the incorrect value 0. The trace and the states at each execution step of the failing run are presented in the second and the third columns, respectively. The two columns on the right show the trace and the states for the correct execution. At 15_2 of the failing run, meaning the second instance of statement 15, the faulty variables are x and z , as they have different values at 15_2 in the correct run. However, x is not essential to the failure, as replacing its value with that of $(x \mapsto 10)$ in the correct run does not mask the failure. Therefore, the FIS of 15_2 contains only z . Observe that the technique hinges on correctly identifying corresponding points in the two executions. This is defined as the *execution alignment* problem. While it is clear that 18_1 and 15_2 in the failing run align with 18_1 and 15_2 in the correct run, the alignments of 14_1 , 5_1 , 14_2 and 5_2 of the failing run are less clear. A simple strategy, which was used in [3, 2], is to align two execution points in the two respective executions that have the same statement, calling context, and instance count. Following such a strategy, 14_2 and 5_2 in the failing run do not have aligned points in the correct run and the 14_1 s and 5_1 s in the two respective executions align as shown in the figure. Such alignments are undesirable because they result in $FIS(14_1)=FIS(5_1)=\{i \mapsto 0\}$, i.e., the failure will not occur if the value of i is replaced with that of i at 14_1 or 5_1 in the correct run. In other words, the benign state $\{i \mapsto 0\}$ is mistakenly identified as failure inducing state.

In our prior work [1], we proposed using execution indexing [4] to align executions. This uses a tree, called the *index tree*, that represents the hierarchical structure of an execution so that executions can be aligned by aligning their

Code	Failing Execution (a=2, b=3)		Ideal Execution (a=2, b=3)	
	Trace	State	Trace	State
		a, b, x, y, z, i, (x>b)		a, b, x, y, z, i, (x>b)
1. int F(int v, int w) {	8 ₁ input(a,b);	2, 3, 0, 0, 0, 0, -	8 ₁ input(a,b);	2, 3, 0, 0, 0, 0, -
2. return v+w;	9 ₁ F(a,b);	2, 3, 0, 0, 0, 0, -	9 ₁ F(a,b);	2, 3, 0, 0, 0, 0, -
3. }	2 ₁ y=a+b;	2, 3, 0, 5, 0, 0, -	2 ₁ y=a+b;	2, 3, 0, 5, 0, 0, -
4. int G(int v, int w) {	10 ₁ x=a+4;	2, 3, 6, 5, 0, 0, -	10 ₁ x=a;	2, 3, 2, 5, 0, 0, -
5. return v-w;	11 ₁ z=10;	2, 3, 6, 5, 10, 0, -	11 ₁ z=10;	2, 3, 2, 5, 10, 0, -
6. }	12 ₁ for (i=...) {	2, 3, 6, 5, 10, 0, -	12 ₁ for (i=...) {	2, 3, 2, 5, 10, 0, -
7.	13 ₁ if (x>b)	2, 3, 6, 5, 10, 0 T	13 ₁ if (x>b)	2, 3, 2, 5, 10, 0 F
8. input(a,b);	14 ₁ G(z, y);	2, 3, 6, 5, 10, 0 -	15 ₁ x=x+4;	2, 3, 6, 5, 10, 0, -
9. y= F(a,b);	5 ₁ z=z-y;	2, 3, 6, 5, 5, 0 -	12 ₂ for (i=...) {	2, 3, 6, 5, 10, 1, -
10. x=a+4; // should be x=a;	15 ₁ x=x+4;	2, 3, 10, 5, 5, 0, -	13 ₂ if (x>b)	2, 3, 6, 5, 10, 1, T
11. z=10;	12 ₂ for (i=...) {	2, 3, 10, 5, 5, 1, -	14 ₁ G(z, y);	2, 3, 6, 5, 10, 1, -
12. for (i=0; i<2; i++) {	13 ₂ if (x>b)	2, 3, 10, 5, 5, 1, T	5 ₁ z=z-y;	2, 3, 6, 5, 5, 1, -
13. if (x>b)	14 ₂ G(z, y);	2, 3, 10, 5, 5, 1, -	15 ₂ x=x+4;	2, 3, 10, 5, 5, 1, -
14. z=G(z, y);	5 ₂ z=z-y;	2, 3, 10, 5, 0, 1, -	18 ₁ print(z);	2, 3, 10, 5, 5, 2, -
15. x=x+4;	15 ₂ x=x+4;	2, 3, 14, 5, 0, 1, -		
16. }	18 ₁ print(z);	2, 3, 14, 5, 0, 2, -		
18. print(z);				

Fig. 1. Misaligned failure inducing states. The ideal execution is from the corrected program with the same input. The dashed lines represent execution alignments. Boxes denote failure inducing states computed through comparisons. Note that function invocations in the traces are transformed to better reflect their semantics, e.g., variables such as v and w are omitted for brevity in the state columns.

trees. Let us first focus on the tree for the failing run presented on the left in Fig. 2. The root node denotes the entire execution, which is the body of the main function. The main body comprises the execution of statements 8_1 , 9_1 , 10_1 , 11_1 , 12_1 , and 18_1 , which are the nodes at the first level. Observe that statement executions 9_1 and 12_1 have substructures, so substructure nodes are introduced at the second level. The procedure continues until all hierarchical substructures are exposed. Note that the second iteration of loop 12, denoted by the subtree rooted at 12_2 , is considered as part of the first iteration, denoted by node 12_1 . The reason is that the execution of the second iteration is determined by the fact that the first iteration gets executed. Similarly, the index tree of the correct run can be constructed. The two executions are aligned by aligning their index trees. As a result, 14_1 and 5_1 in the left tree do not align with any nodes in the right tree, and 14_2 and 5_2 in the left tree align with 14_1 and 5_1 in the right tree. Thus, $FIS(5_2) = \{z \mapsto (0, 5_2)\}$, with the state of z being a pair comprising its value and the definition point of the value, because it is the minimal subset of the faulty variables at 5_2 that causes the FIS of its next step, 15_2 . Recall that $FIS(15_2) = \{z \mapsto (0, 5_2)\}$. $FIS(14_2) = \{z \mapsto (5, 5_1)\}$, as it induces $FIS(5_2)$. The FISs of all comparable execution points are annotated on the nodes in the left tree. The definition points in these FISs constitute the causal path as highlighted in the left tree, which is $10_1 \rightarrow 13_1 \rightarrow 5_1 \rightarrow 5_2 \rightarrow 18_1$. One can see it starts with the root cause and clearly explains how the fault leads to the failure. In comparison, the dynamic slice [6] of the failure point 18_1 contains all the executed statements, including the causal path.

So far, we have discussed how to compute the causal path in the ideal case where the corrected program is available. In realistic debugging, however, only the buggy program is available. It was proposed in [3, 2] to use a similar but passing run of the buggy program as the reference run to perform the comparison. Unfortunately, since the two executions are derived from two different inputs, the semantic differences often significantly compromise the resulting causal path. Consider the case in Fig. 3. The failing run is the same as before, but since the corrected program is not available, a similar but passing run of the buggy

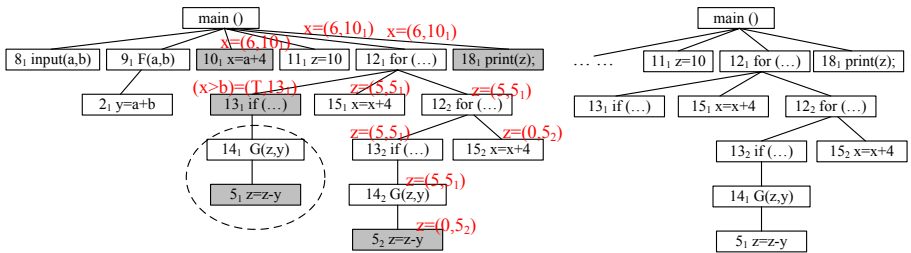


Fig. 2. Index trees for the two executions in Fig. 1. Execution alignment is achieved by aligning the two trees. The circled portion does not align with any part in the other tree. FISs for aligned nodes are annotated. The state of a variable is a pair (*value*, *definition point*). The causal path is highlighted in the index tree of the failing run.

program is used, derived from the input $a=2$ and $b=1$. Note that although the run on the right is from the buggy program, it produces the expected output, i.e., the same output as that produced by the corrected program, because whether x has the value of 6 or 2 at 10_1 does not affect the final output. The two executions have identical control flows and their inputs differ by only one value. The two executions can be trivially aligned. Comparing the states at the aligned steps as mentioned earlier produces the FISs boxed in the figure. Notice that $FIS(15_2) = \{z \mapsto (0, 5_2)\}$, as it is the minimal state difference that induces the failure. $FIS(14_2) = \{y \mapsto (5, 2_1), z \mapsto (5, 5_1)\}$ because it is the minimal state difference that induces $FIS(5_2) = \{z \mapsto (0, 5_2)\}$, even though we know y has a benign value. The definition points of the faulty states at each step constitute the causal path as highlighted on the left. Observe that although it has causality between steps, the path does not explain the failure but rather the difference between the two executions. Particularly, $\{b \mapsto (3, 8_1)\}$ is computed for $FIS(8_1)$ although b has a completely benign value at 8_1 . Similarly, $FIS(2_1) = \{y \mapsto (5, 2_1)\}$ is due to the semantic differences between the two executions. Even worse, x is not part of $FIS(10_1)$ as it has the same value in both executions. In other words, the real faulty state is mistakenly considered as being benign.

In our technique, we first construct a dynamic patch to correct the failing execution and then use the patched execution as the reference run for comparison [1]. A failing execution is *patched* if mutating part of its state at one or multiple execution points leads to the correct output. Since the patched execution is derived from the same input, it precludes state differences caused by the input differences. Predicate switching [7] is our prior work on patching a failing run. It works by systematically changing the branch outcome of a predicate instance and then observing if the mutated execution produces the expected output. It was used as a fault localization technique because, if such a predicate

Failing Execution (a=2, b=3)		A Similar but Passing Execution (a=2, b=1)	
Trace	State	Trace	State
	a, b, x, y, z, i, (x>b)		a, b, x, y, z, i, (x>b)
8 ₁ input (a,b);	2, 3 , 0, 0, 0, 0, -	8 ₁ input (a,b);	2, 1, 0, 0, 0, 0, -
9 ₁ F (a,b);	2, 3 , 0, 0, 0, 0, -	9 ₁ F (a,b);	2, 1, 0, 0, 0, 0, -
2 ₁ y=a+b;	2, 3, 0, 5 , 0, 0, -	2 ₁ y=a+b;	2, 1, 0, 3, 0, 0, -
10 ₁ x=a+4;	2, 3, 6, 5 , 0, 0, -	10 ₁ x=a+4;	2, 1, 6, 3, 0, 0, -
11 ₁ z=10;	2, 3, 6, 5 , 10, 0, -	11 ₁ z=10;	2, 1, 6, 3, 10, 0, -
12 ₁ for (i=...) {	2, 3, 6, 5 , 10, 0, -	12 ₁ for (i=...) {	2, 1, 6, 3, 10, 0, -
13 ₁ if (x>b)	2, 3, 6, 5 , 10, 0, T	13 ₁ if (x>b)	2, 1, 6, 3, 10, 0, T
14 ₁ G (z, y);	2, 3, 6, 5 , 10, 0, -	14 ₁ G (z, y);	2, 1, 6, 3, 10, 0, -
5 ₁ z=z-y;	2, 3, 6, 5 , 0, 0, -	5 ₁ z=z-y;	2, 1, 6, 3, 7, 0, -
15 ₁ x=x+4;	2, 3, 10, 5 , 0, 0, -	15 ₁ x=x+4;	2, 1, 10, 3, 7, 0, -
12 ₂ for (i=...) {	2, 3, 10, 5 , 1, 0, -	12 ₂ for (i=...) {	2, 1, 10, 3, 7, 1, -
13 ₂ if (x>b)	2, 3, 10, 5 , 1, T	13 ₂ if (x>b)	2, 1, 10, 3, 7, 1, T
14 ₂ G (z, y);	2, 3, 10, 5 , 1, -	14 ₂ G (z, y);	2, 1, 10, 3, 7, 1, -
5 ₂ z=z-y;	2, 3, 10, 5 , 0, 1, -	5 ₂ z=z-y;	2, 1, 10, 3, 4, 1, -
15 ₂ x=x+4;	2, 3, 14, 5 , 0, 1, -	15 ₂ x=x+4;	2, 1, 14, 3, 4, 1, -
18 ₁ print(z);	2, 3, 14, 5 , 0, 2, -	18 ₁ print(z);	2, 1, 14, 3, 4, 2, -

Fig. 3. Comparing the failing run with a similar but passing run with different input

instance exists, called the *critical predicate*, it discloses a wealth of information on the fault. Our study [1] showed that 80% of all the failing test cases in the SIR [5] suite and 8 out of 12 real bugs collected from internet can be patched by predicate switching. A patched run serves as an approximation of the ideal run to carry out execution comparison. For instance, if the predicate instance 13_1 of the failing run in Fig. 1 is switched so that 14_1 and 5_1 are not executed, the resulting z value at 18_1 becomes the desired 5. The patched run is very similar to the *ideal* run on the right of Fig. 1. The only difference is that variable x has different values from 10_1 to 13_1 in the two respective runs. The causal path is computed as $13_1 \rightarrow 5_1 \rightarrow 5_2 \rightarrow 18_1$, which captures most of the ideal causal path. Note that the root cause 10_1 is not caught, as the patched run has identical state as the failing run till the switched predicate instance 13_1 . Although the root cause is not captured in this example, it can often be captured by our technique if the switched predicate determines if the root cause gets executed. Our prior study [1] shows that about 45% of the causal paths computed by our technique capture the root cause. Moreover, we argue that presenting the causal path is more informative than pointing at root cause candidates. It is reported in [8] that requirement bugs are the most frequently occurring kind of bug in the field, which often do not have a single or a small set of statements to be blamed as the root cause. For such cases, understanding failure causality is preferable.

3 Algorithms

The major contribution of this paper is a detailed study of two algorithms for the aforementioned causal path computation. The two algorithms produce the same causal paths but achieve efficiency with different approaches. A naïve approach is to first align the two executions by aligning their index trees, then compute the FISs for each aligned step backwards, starting from the failure point. Our experience shows that such an algorithm is extremely expensive due to the large number of aligned execution points. For the failures collected from Linux utilities, listed in Section 4, the algorithm failed to terminate after 8 hours of computation. Note that all these algorithms require the same basic FIS computation that compares states of two aligned steps in the two respective executions and minimizes the state differences using the delta debugging algorithm [9, 3, 2].

3.1 A Hierarchical Algorithm

We have proven in [1] that FISs have a stability property, which states that *if the FISs computed at any two aligned points are the same, there is no need to compute FISs between these two points because they will be identical*. For example, in Fig. 3, as the two aligned steps 5_1 and 14_2 in the failing run have the same FIS, i.e., $\{y \mapsto (5, 2_1), z \mapsto (5, 5_1)\}$, all aligned steps in between have the same FIS, too. Formally, this property requires a more advanced notion of FIS that captures the definition points and values from both the faulty and correct executions, but we elide these details in presentation for simplicity. Based on this

property, a hierarchical algorithm can be designed to compute FISs in a demand-driven fashion. The idea is to carry out state comparison and causality testing top-down along the index tree of the failing execution until the right granularity is reached. Each FIS is computed to induce the successive one except for the last FIS, which induces the observed failure.

Algorithm 1. Hierarchical computation for causal paths of failures.

Primitives:

- **ALIGNEDCHILDREN()**- Finds the children that align with some nodes in the reference run.
- **FIS()**- Computes the FIS of the given aligned node.

Important Variables:

- **target** - The FIS to be induced when computing the preceding FIS.
- **defs** - The sequence of definition points that constitute the causal path.
- **sets**- A set of FISs.

CALULATECAUSALPATH()

```

1 target ← failure
2 defs ← {target}
3 (sets,target) ← SETSINREGION(executionRoot,target)
4 defs ← defs ∪ (∪fis∈sets)
5 return TEMPORALSORT(defs)

```

SETSINREGION(node, target)

INPUT: A node in the index tree and the first FIS that must be induced.

OUTPUT: The set of FISs in the region and the single FIS that induces them.

```

1 sets ← ∅
2 kids ← ALIGNEDCHILDREN(node) \ {children executing with or after target}.
3 while |kids| > 0 do
4   while |kids| > 1 do
5     mid ← |kids| / 2
6     newFIS ← FIS(kidsmid) inducing target
7     if newFIS = target then
8       kids ← kids0...mid-1
9     else
10      kids ← kidsmid...|kids|-1
11   if FIS(kids0) ≠ target then
12     (subsets,target) = SETSINREGION(kids0, target)
13     sets ← sets ∪ subsets
14   kids ← ALIGNEDCHILDREN(node) \ {children executing with or after kids0}.
15 newFIS ← FIS(node) inducing target
16 return (sets ∪ newFIS, newFIS)

```

The algorithm is shown in Algorithm 1. Using the failing execution and the reference execution, **CALULATECAUSALPATH()** generates the complete causal path. This procedure first initializes variables **target**, which is the FIS or the failure to be induced by the next computed FIS, and **defs**, which stores the sequence of definitions constituting the causal path. It then calls **SETSINREGION()** with the root of the index tree and **target** to compute the set of FISs, stored in **sets**, in a top-down manner. Once all FISs have been aggregated, the definitions they contain are sorted by their temporal position and returned at line 5. This sequence of definitions comprises the complete causal path.

SETSINREGION() generates the FISs for the portion of the indexing tree rooted at **node** such that the temporally last generated FIS induces the FIS **target**. At line 2, the algorithm extracts the ordered list of all the child nodes

that have alignments in the reference execution and stores them in **kids**, excluding those executed after or with **target**. If there are no such children, the function skips the loop in lines 3-14 and computes the FIS for the **node** that induces **target**. The loop computes the set of FISs of the subtree. The inner loop in lines 4-10 performs a classic binary search on **kids** list to locate the first child from the end that has an FIS different from **target**. Observe that if the FIS of the midpoint is identical to **target**, as checked at line 7, the algorithm safely skips computing FISs for the right half of the **kids** list and its subtrees. If such a child is found, the function recursively calls itself to compute the FISs in the subtree rooted at the child on line 12. On line 14, the **kids** list updates to reflect a new FIS, so the next round of binary search will be performed on a reduced list. The computation terminates if the list becomes empty.

Consider the example in Fig. 2. The computation starts from the top of the tree on the left, and the failure at 18_1 , i.e., $\{z \mapsto (0, 5_2)\}$, is the **target** to induce. In the first invocation of SETSINREGION(), the **kids** list is initialized to contain $\{8_1, 9_1, 10_1, 11_1, 12_1\}$. The binary search in lines 4-10 identifies the first child with an FIS different than **target** to be 12_1 . It recursively calls itself on 12_1 to compute the FISs in the subtree rooted at 12_1 . This time, the **kids** list is initialized to have $\{13_1, 15_1, 12_2\}$. The algorithm descends along 12_2 , 13_2 , and then 14_2 because their FISs are different than the failure until the closest different FIS inducing the failure, namely 14_2 , is identified. The **target** is updated to be 14_2 . At some point, the recursive call for 12_2 returns with $FIS(12_2) = \{z \mapsto (5, 5_1)\}$ being the **target**. The **kids** list is updated to $\{13_1, 15_1\}$ at line 14 to start a new round of the binary search. This time, the search identifies $FIS(15_1) = FIS(12_2)$, so if 15_1 were the root for a subtree, the algorithm would not descend into the subtree. Computation over the remainder of the tree can be similarly derived.

3.2 A Shortcutting Algorithm

We introduce here another algorithm that exploits the stability property in a different way. That is, given an FIS to induce, we try to identify the earliest aligned point that is likely to have the same FIS and jump directly to that point without computing any FISs in between. The intuition is that the earliest point that has the same FIS is very likely *the last definition point, before the current point, that occurs in any of the previously computed FISs* because all faulty values in previously computed FISs remain intact between their definition points and FISs. For example, in Fig. 3, at the aligned step 14_2 , the set of definitions in all the previously computed FISs is $\{2_1, 5_1, 5_2\}$. Recall that causal path computation proceeds backwards, starting from the failure, with the first FIS inducing the failure and the remaining FISs each inducing the successive FIS. The last definition point that happens before 14_2 is 5_1 . Observe that $FIS(5_1)$ is still $\{y \mapsto (5, 2_1), z \mapsto (5, 5_1)\}$. That implies we are taking the shortcut, jumping directly from 14_2 to 5_1 . Further note that the FIS immediately before is $\{y \mapsto (5, 2_1)\}$. This reflects the change in FIS that the definition at 5_1 causes. By taking dependence shortcuts, fewer searches for computing an FIS are needed.

Pseudocode for this approach is presented in Algorithm 2. The main procedure `CALCULATECAUSALPATH()` derives the causal path for the failing execution, computing backwards starting from the failure. In lines 4-17, the algorithm traverses backwards by taking shortcuts if possible, accumulating FISs in `defs` along the way, until there is no more relevant state, as checked at 4. On line 5, the last definition that was previously caught in the causal path and occurred before the current computation point is stored in `closest`. In lines 6-9, the algorithm handles cases in which no shortcut is available as the captured definition points happen after or at the current traversal point. The algorithm conservatively moves one step backwards. Lines 10-17 handle cases in which a shortcut is possible. Line 13 checks if taking the shortcut is valid by comparing the FIS of `closest` with `target`. If they are identical, the shortcut is valid and then at line 17, the algorithm updates `currentNode`. Note, however, the shortcut may not always be valid, i.e., a different FIS may be computed at the destination of the shortcut. For example, in Fig. 4 the failing run is different from the reference run because it omits the execution of the true branch. Thus, variable x is not updated. Assume $FIS(7_1)$ is computed as $\{x \rightarrow (2, 2_1)\}$. If the algorithm takes the shortcut to 2_1 , the FIS at 2_1 is empty as all variables have the right value. More important, following the shortcut misses the real FIS alteration point 3_1 . This results from the failing run omitting execution that it should have gone through. Note that similar effects can be observed if program state is updated by the OS and thus not visible to our analysis. Fortunately, such cases are reflected in the FIS at the shortcut being different from `target`. If they occur, the algorithm falls back to a revised hierarchical search at line 14.

`CLOSESTADAPTIVE()` performs a revised hierarchical derivation of a single FIS given the `target`, i.e., the FIS to be induced, and the bound on how early the FIS transition could occur. Line 1 finds the common ancestor in the index tree of both the early bound and the index at which the target FIS was derived. Intuitively, execution within the provided bounds must be in the subtree rooted at the ancestor. Lines 2-14 of `CLOSESTADAPTIVE()` simply perform a hierarchical binary search for the resulting FIS. The procedure is very similar to `SETSINREGION` in Algorithm 1. At line 3, the children of the `start` node are retrieved. Those that happen before `closest` and after or at the point where `target` is computed are filtered out. The loop in lines 4-10 is a classic binary

Code	Failing Run		Ref. Run	
		State x p	Trace	State x p
1. ...				
2. x=2;	1 ₁ ...		1 ₁ ...	
3. if (p) {	2 ₁ x=2;	(2, 2 ₁) -	2 ₁ x=2;	(2, 2 ₁) -
4. ...	3 ₁ if (p)	(2, 2 ₁) (F, 3 ₁)	3 ₁ if (p) {	(2, 2 ₁) (T, 3 ₁)
5. x = x-1;			4 ₁ ...	
6. }			5 ₁ x=x-1;	
7. ...;	7 ₁ ...;	(2, 2 ₁) -	7 ₁ ...;	(1, 5 ₁) -

Fig. 4. Taking the shortcut is not successful due to execution omission in the failing run. The horizontal lines denote execution alignment.

search that looks for the first kid with an FIS different from **target**, and then the algorithm traverses the index tree one level down to the kid at line 14.

Algorithm 2. Shortcutting computation for causal paths of failures.

Important Variables:

- **target** - The FIS to be induced when computing the preceding FIS.
- **defs** - The sequence of definition points that constitute the causal path.
- **currentNode** - The current FIS computation point.

CALCULATECAUSALPATH()

```

1  currentNode ← the failure point
2  target ← failure
3  defs ← {target}
4  while target ≠ ∅ do
5    closest ← the temporally last definition in defs that happens before currentNode
6    if closest = ⊥ then
7      currentNode ← the aligned point immediately preceding currentNode.
8      target ← FIS(currentNode)
9      defs ← defs ∪ target
10   else
11     if closest is not aligned then
12       closest ← the aligned point immediately preceding closest.
13     if FIS(closest) ≠ target then
14       target ← CLOSESTADAPTIVE(target, closest)
15       defs ← defs ∪ target
16     else
17       currentNode ← closest
18  return TEMPORALSORT(defs)

```

CLOSESTADAPTIVE(target, closest)

INPUT: A target FIS to induce and a bound on the earliest point it may be induced.

OUTPUT: The FIS that immediately precedes the target in the sequence of FISs.

```

1  start ← the common ancestor in the indexing tree of closest and the index of target.
2  loop
3    kids ← ALIGNEDCHILDREN(start) \ {children before closest, after target, and target itself}.
4    while |kids| > 1 do
5      mid ← |kids| / 2
6      newFIS ← FIS(kidsmid) inducing target
7      if newFIS = target then
8        kids ← kids0...mid-1
9      else
10       kids ← kidsmid...|kids|-1
11   if kids = ∅ then
12     currentNode ← start
13     return FIS(start) inducing target
14   start ← kids0

```

Consider the example in Fig. 2. To start, **target**=**failure**={ $z \mapsto (0, 5_2)$ }, so the shortcut jumps to 5₂. The shortcut is valid, as $FIS(15_2)=\mathbf{target}$. In the next round of the main loop in **CALCULATECAUSALPATH()**, the algorithm traverses one step backwards since no shortcut is available because all captured definitions happen after or at 5₂. The next FIS computation is for 14₂, which has the result $\{z \mapsto (5, 5_1)\}$. Due to the newly caught definition 5₁, a shortcut is available to reach 5₁. However, 5₁ is not aligned and thus its immediate aligned predecessor 13₁ is used to compute the FIS instead, as in line 12 of **CALCULATECAUSALPATH()**. The resulting $FIS(13_1) = \{(x > b) \mapsto (T, 13_1)\}$ differs from **target**= $FIS(14_2)$. That is, the shortcut is not valid and the algorithm must fall back and call **CLOSESTADAPTIVE()**. The binary search eventually identifies 13₁ as the closest point at which a different FIS is computed that induces

$FIS(14_2)$. Now, since there is not a captured definition occurring before 13_1 , the algorithm moves one step backwards and computes $FIS(12_1) = \{x \mapsto (6, 10_1)\}$. A shortcut is available leading to the root cause 10_1 .

4 Evaluation

In order to evaluate the presented algorithms, we employed them against several real world bugs found in the GNU utilities `grep`, `gzip`, `bc`, `find`, `diff`, and `tar`. The debugging infrastructure comprises source to source transformation via CIL, Python, and the public Python and GDB infrastructure from [3]. The two algorithms were implemented with CIL and Python. The tests were run on a 2GHz dual core machine with 2GB of RAM. For each analyzed bug, Table 1 presents the program and version the bug applies to, a bug report if available, the number of definitions the causal path comprises, and the time in seconds required for each algorithm to derive the causal path. *Time 1* shows to the time taken by Algorithm 1 and *Time 2* shows the time taken by Algorithm 2.

Observe that the time taken by Algorithm 2, using shortcutting, is consistently and substantially less than the time required by Algorithm 1, 7% as long or less on average. The most significant factors in the runtime of the algorithm are the number of causality tests and the size and complexity of the program states that must be analyzed during causality tests. Causality testing is part of the FIS computation. It determines if a subset of state differences induce the successive FIS. It is done through re-executing the program with the state differences applied. The efficiency difference in Table 1 results predominantly from decreasing the number of causality tests via shortcutting. In practice, execution omission and unobserved state force the shortcutting approach to degenerate into a hierarchical binary search for some individual elements of the path, but

Table 1. Examined bugs and their causal path properties. Time n is the time in seconds taken to derive the causal path using Algorithm n . ‘Bug’ is the Internet address of a bug report, if applicable.

Program	Version	Bug	Path Length	Time 1	Time 2
bc	1.06	bugs.gentoo.org/51525	2	1130	135
diff	2.8	gnu....utils/2002-12/msg00067.html	8	2320	368
find	4.3.0	savannah.gnu.org/bugs/?18222	5	2906	48
grep	2.5.1	savannah.gnu.org/bugs/?11579	5	1220	159
grep	2.5.1	savannah.gnu.org/bugs/?9519	7	>4 hours	250
grep	2.5.1	savannah.gnu.org/bugs/?13920	7	6865	217
grep	2.5.1	savannah.gnu.org/bugs/?9768	5	6167	221
grep	2.5.1	savannah.gnu.org/bugs/?19491	9	>4 hours	244
grep	2.5.3	savannah.gnu.org/bugs/?15620	4	>4 hours	56
grep	2.5.3	-h -H with a single file	4	>4 hours	55
gzip	1.3.9	gnu....gzip/2007-05/msg00003.html	15	7583	651
tar	1.13.25	gmane....comp.gnu.tar.bugs/491	7	8823	531

most shortcutting efforts are successful. Both algorithms are significantly faster than a naïve algorithm that computes FISs linearly by traversing backwards step by step. Note that all these algorithms compute the same causal path.

In the second experiment, we evaluate the effectiveness of our technique on a set of real bugs from Linux utility programs. We collected these bugs by looking into their CVS repositories and on-line bug reports. Some of these cases are explained in detail below.

Grep. Version 2.5.3 of the **grep** regular expression matching utility incorrectly handles command-line options **-h** and **-H**, which respectively disable and enable printing out the filename of a file containing a matched expression. If both options are given, only the last should be obeyed, but both options are independently enabled, yielding inconsistent results. Interestingly, there are multiple ways that this fault can manifest, but they have the same causal path, which identifies them as stemming from the same fault. If options **-H -h** are used when searching multiple files, file names prefix all resulting output with matches in those files, but they should not. If **-H -h** are used when searching one file containing short lines of text, some of the resulting output lines have the filename prefix, while others do not. It is not immediately apparent that the same fault affects both executions. The causal paths and faulty code for these executions, however, are the same, as shown in Fig. 5(a). The switched predicate on line 5 becomes false, preventing the flag for printing filename prefixes from becoming enabled. The resulting causal path shows that in the failing runs, the predicate on line 5 enables the flag `out_file` for printing filename prefixes on line 6. Much later in the execution, this causes the predicate on line 7 to be true, which then results in failure when the filenames are printed at 8. The switched predicate and causal path reveal that the predicate should not evaluate to true, but the **-H** command forces it to via the variable `with_names`, as it is mistakenly not disabled by the later option **-h**. The applied fix was then to disable the opposing commands on lines 2 and 3.

Find. Find is a tool that locates all files matching provided criteria and also performs an action at all such files. Version 4.3.0 contains a bug when multiple directories to search are specified. If the given action is `-printf '%H %P\n'`, which prints out the specified directory name before each file found in the directory, then every directory name printed is no longer than the first. For instance, if the directories `dir1` and `directory` are specified, the contents of `dir1` will be printed with the prefix `'dir1'`, and the contents of `directory` will have the prefix `'dire'`. The causal path is presented in Fig. 5 (b). At the first step of the causal path, the variable `s.starting_length` is set to 4, recording the length of the first specified directory. The critical predicate is on line 1, which corresponds to the second invocation of the function `consider_visiting()`, handling the second specified directory. In the failing run, it evaluates to false, so when it gets switched, `s.starting_length` is updated to 9 and thus leads to the correct output. The causal path also captures that `s.starting_length` is used at line 6 to cut off the second directory name and eventually produce the wrong output. The assignment to `cc` at line 5 is in the causal path, even though it may not

Code Snippet:

```
main(argc,argv):
1 switch (options) {
2   case 'H': with_names = true;break;
3   case 'h': no_names = true;break;
4   : if ((num_files>1 && !no_names) ||
5       with_names)
6     out_file = true;
print_line_head(beg, lim, sep):
7   if (out_file)
8     print_filename();
```

Causal Path:

At 5, (...) || with_names) is true
 At 6, out_file is given true
 At 7, (out_file) is true
 Thus the filename is printed at 8.

(a)

Code Snippet:

```
consider_visiting(p,ent):
1 if (0 == s.starting_length)
2   s.starting_length = ent->fts_pathlen;
pred_fprintf(pathname,stat_buf,pred_ptr):
3 switch (kind & 0xff) {
4   case 'H':
5     cc = pathname[s.starting_length];
6     pathname[s.starting_length] = '\0';
7     printf(pathname);
8     pathname[s.starting_length] = cc;
9     break;
```

Causal Path:

At 2, s.starting_length is given 4
 At 1, (0 == s.starting_length) is false
 At 5, cc is given 'c'
 At 6, pathname is given "dire"
 So "dire" is printed at 7.
 "dire" is printed at 7 again.

(b)

Fig. 5. Causal paths for (a) `grep` and (b) `find`

seem needed for failure induction, because the wrong directory name '`dire`' was printed multiple times, and the assignment at line 5 is critical to continually printing the wrong directory name. This case also shows how a causal path does not necessarily start with the critical predicate. Namely, the definition at line 2 is the first step of the causal path because it is in $FIS(5)$ and $FIS(6)$.

Diff. The `diff` tool compares two files or the contents of two directories and reports differences. In version 2.8, an option to ignore the case within filenames works incorrectly when comparing directory contents. Thus, if one directory contains the file `bar` while another contains `BAR`, comparing the directories with the `--ignore-file-name-case` option should print nothing, but it reports the above files as being different. The causal path and code in Fig. 6 for this bug show that this is due to an incorrect name comparison algorithm. When comparing directories, the list of files from one directory is compared to the list in another using the `diff_dirs` function. This uses `compare_names()` to compare individual files. The critical predicate on line 2 shows that when two file names are case-insensitively equal, that equality is immediately disregarded because `(r)` is false. Thus, the `compare_names()` function continues and returns that the names are case-sensitively unequal via `-7` at line 4 and into `order` at 6. This is used to determine that one of the files is unique when `(order<0)` is true at line 8 causes `*fname1` to be zeroed out at 9. Thus, when the `compare_files()` function is called to show the results, the argument `name1` is given 0, and `(!(name0 && name1))` evaluates to true, forcing one of the equivalent filenames, `name0`, to be printed as a difference. The causal path indicates that the right patch should be to change the algorithm in `compare_names()` to make the comparison at line 4 case insensitive when the option is set.

Code Snippet:

```

compare_names(name1,name2);
1  r = strcmp(cmp(name1,name2));
2  if (r)
3      return r
4  return strcmp(name1,name2);
diff_dirs(cmp,handle_file):
5  while (fname0 || fname1) {
6      order = compare_names(*fname0,*fname1);
7      ... if (order < 0)
8          *fname1 = 0;
9  v1 = compare_files(cmp,*fname0,*fname1);
compare_files(parent,name0,name1):
11 if (!(name0 && name1))
12     print(name0 == 0 ? name1 : name0);

```

Causal Path:

```

At 2, (r) is false
At 4, compare_names returns -7
At 6, order is given -7
At 8, (order < 0) is true
At 9, *fname1 is given 0
At 10, name1 is given 0
At 11, (!(name0 && name1)) is true
So the filename is shown at 12

```

Fig. 6. Causal path for diff

5 Related Work

Delta Debugging. The work most relevant to ours is that by Zeller et al. [9, 2, 3]. The project in [2] is the first one to propose comparing two similar executions using delta debugging [9] to compute cause-effect chains, which is a concept similar to causal paths of failures. Later in [3], the technique is extended to link cause transitions to a faulty statement. Compared to these works, we make significant progress on the following: we identify execution indexing as a key technique, use a patched execution instead of a different execution to reduce noise from semantic differences, and develop efficient algorithms. Our prior evaluation [1] using the SIR suite [5] showed that our technique is superior.

Fault Localization. Fault localization computes fault candidates by looking at many executions, both passing and failing, as exemplified by [10, 11, 12, 13, 14, 15, 16, 17]. Details of these techniques cannot be presented due to space limits. Compared to our technique, fault localization techniques are less effective in explaining failures. They produce a ranked candidate set, usually containing static statements. Reasoning about the candidates and the failure often falls onto the programmer.

Dynamic Slicing. Dynamic slicing was introduced as an aid to debugging [6]. Compared to fault localization, slicing features the capability of capturing causality through program dependencies. However, slicing tends to produce fat slices containing not only the failure inducing dependencies but also benign dependencies. Although various techniques have been proposed to prune dynamic slices [18, 19], without using a reference execution to exclude benign chains, inspecting pruned slices still requires non-trivial human effort. Dicing [20] aggregates slices from multiple executions. However, the simple set manipulations in dicing undermine causality and make resulting slices hard to understand. Furthermore, it does not handle cases in which a faulty statement occurs in both the benign and faulty slices. In comparison, our work does not rely strictly on program dependence but rather on semantic causality. The use of a reference execution effectively excludes benign state.

Acknowledgments

This work is supported by NSF grants CNS-0720516 and CNS-0708464 to Purdue University.

References

- [1] Sumner, W.N., Zhang, X.: Automatic failure inducing chain computation through aligned execution comparison. Tech. Rep. 08-023, Purdue University (2008), http://www.cs.purdue.edu/homes/wsumner/CSD_TR_08-023.pdf
- [2] Zeller, A.: Isolating cause-effect chains from computer programs. In: FSE (2002)
- [3] Cleve, H., Zeller, A.: Locating causes of program failures. In: ICSE (2005)
- [4] Xin, B., Sumner, W.N., Zhang, X.: Efficient program execution indexing. In: PLDI (2008)
- [5] Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal* 10(4)
- [6] Korel, B., Laski, J.: Dynamic program slicing. *Information Processing Letters* 29(3) (1988)
- [7] Zhang, X., Gupta, N., Gupta, R.: Locating faults through automated predicate switching. In: ICSE (2006)
- [8] Jackson, D., Thomas, M., Millett, L.I.: *Software for Dependable Systems: Sufficient Evidence?* The National Academies Press
- [9] Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28(2) (2002)
- [10] Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L.: An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability* 10(3)
- [11] Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: ICSE (2002)
- [12] Renieris, M., Reiss, S.: Fault localization with nearest neighbor queries. In: ASE (2003)
- [13] Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: PLDI (2003)
- [14] Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.: Sober: statistical model-based bug localization. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557. Springer, Heidelberg (2005)
- [15] Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: ICSE (2004)
- [16] Chesley, O.C., Ren, X., Ryder, B.G., Tip, F.: Crisp—a fault localization tool for java programs. In: ICSE (2007)
- [17] Wang, T., Roychoudhury, A.: Automated path generation for software fault localization. In: ASE (2005)
- [18] Gupta, N., He, H., Zhang, X., Gupta, R.: Locating faulty code using failure-inducing chops. In: ASE (2005)
- [19] Zhang, X., Gupta, N., Gupta, R.: Pruning dynamic slices with confidence. *SIGPLAN Not.* 41(6) (2006)
- [20] Chen, T.Y., Cheung, Y.Y.: Dynamic program dicing. In: ICSM (1993)

Mining API Error-Handling Specifications from Source Code

Mithun Acharya and Tao Xie

Department of Computer Science, North Carolina State University, Raleigh, NC, USA, 27695
{acharya,xie}@csc.ncsu.edu

Abstract. API error-handling specifications are often not documented, necessitating automated specification mining. Automated mining of error-handling specifications is challenging for procedural languages such as C, which lack explicit exception-handling mechanisms. Due to the lack of explicit exception handling, error-handling code is often scattered across different procedures and files making it difficult to mine error-handling specifications through manual inspection of source code. In this paper, we present a novel framework for mining API error-handling specifications automatically from API client code, without any user input. In our framework, we adapt a trace generation technique to distinguish and generate static traces representing different API run-time behaviors. We apply data mining techniques on the static traces to mine specifications that define correct handling of API errors. We then use the mined specifications to detect API error-handling violations. Our framework mines 62 error-handling specifications and detects 264 real error-handling defects from the analyzed open source packages.¹

1 Introduction

Motivation. A software system interacts with third-party libraries through various Application Programming Interfaces (API). Throughout the paper, we overload the term API to mean either a set of related library procedures or a single library procedure in the set – the actual meaning should be evident from the context. Incorrect handling of errors incurred after API invocations can lead to serious problems such as system crashes, leakage of sensitive information, and other security compromises. API errors are usually caused by stressful environment conditions, which may occur in forms such as high computation load, memory exhaustion, process related failures, network failures, file system failures, and slow system response. As a simple example of incorrect API error handling, a *send* procedure, which sends the content of a file across the network as packets, might incorrectly handle the failure of the *socket* API (the *socket* API can return an error value of `-1`, indicating a failure), if the *send* procedure returns without releasing system resources such as previously allocated packet buffers and opened file handlers. Unfortunately, error handling is the least understood, documented, and tested part of a system. Toy’s study [14] shows that more than 50% of all system failures in

¹ This work is supported in part by ARO grant W911NF-08-1-0443.

a telephone switching application are due to incorrect error-handling algorithms. Cristian's survey [7] reports that up to two-thirds of a program may be devoted to error detection and recovery. Hence, correct error handling should be an important part of any reliable software system. Despite the importance of correct error handling, programmers often make mistakes in error-handling code [4, 10, 17]. Correct handling of API errors can be specified as formal specifications verifiable by static checkers at compile time. However, due to poor documentation practices, API error-handling specifications are often unavailable or imprecise. In this paper, we present a novel framework for statically mining API error-handling specifications automatically from software packages (API client code) implemented in C.

Challenges. There are three main unique challenges in automatically mining API error-handling specifications from source code. (1) Mining API error-handling specifications, which are usually temporal in nature, requires identifying API *details* from source code such as (a) *critical* APIs (APIs that fail with errors), (b) different error checks that should follow such APIs (depending on different API error conditions), and (c) proper error handling or clean up in the case of API failures, indicated by API errors. Furthermore, clean up APIs might depend on the APIs called before the error is handled. Static approaches [17, 16] exist for mining or checking API error-handling specifications from software repositories implemented in object-oriented languages such as Java. Java has explicit *exception-handling* support and the static approaches mainly analyze the `catch` and `finally` blocks to mine or check API error-handling specifications. Procedural languages such as C do not have explicit exception-handling mechanisms to handle API errors, posing additional challenges for automated specification mining: API details are often scattered across different procedures and files. Manually mining specifications from source code becomes hard and inaccurate. Hence, we need inter-procedural techniques to mine critical APIs, different error checks, and proper clean up from source code to automatically mine error-handling specifications. (2) As programmers often make mistakes along API error paths [4, 10, 14, 17], the proper clean up, being common among error paths and normal paths, should be mined from normal traces (i.e., static traces without API errors along normal paths) instead of error traces (i.e., static traces with API errors along error paths). Hence, we need techniques to generate and distinguish error traces and normal traces, even when the API error-handling specifications are not known *a priori*. (3) Finally, API error-handling specifications can be *conditional* – the clean up for an API might depend on the actual return value of the API. Hence, trace generation has to associate conditions along each path with the corresponding trace.

Contributions. To address the preceding challenges, we develop a novel framework for statically mining API error-handling specifications directly from software packages (API client code), without requiring any user input. Our framework allows mining system code bases for API error-handling violations without requiring environment setup for system executions or availability of sufficient system tests. Furthermore, our framework detects API error-handling violations, requiring no user input in the form of specifications, programmer annotations, profiling, instrumentation, random inputs, or a set of relevant APIs. In particular, in our framework, we apply data mining techniques on generated static traces to mine specifications that define correct handling of errors for

the APIs used in the analyzed software packages. We then use the mined specifications to detect API error-handling violations. In summary, this paper makes the following main contributions:

- **Static approximation of different API run-time behaviors.** We adapt a static trace generation technique [2] to distinguish and approximate different API run-time behaviors (e.g., error and normal behaviors), thus generating error traces and normal traces inter-procedurally.

- **Specification mining and violation detection.** We apply different mining techniques on the generated error traces and normal traces to identify clean up code, distinguish clean up APIs from other APIs, and mine specifications that define correct handling of API errors. To mine conditional specifications, we adapt trace generation to associate conditions along each path with the corresponding trace. We then use the mined specifications to detect API error-handling violations.

- **Implementation and Experience.** We implement the framework and validate the effectiveness of the framework on 10 packages from the Redhat-9.0 distribution (52 KLOC), postfix-2.0.16 (111 KLOC), and 72 packages from the X11-R6.9.0 distribution (208 KLOC). Our framework mines 62 error-handling specifications and detects 264 real error-handling defects from the analyzed packages.

The remainder of this paper is structured as follows. Section 2 starts with a motivating example. Section 3 explains our framework in detail. Section 4 presents the evaluation results. Section 5 discusses related work. Finally, Section 6 concludes our paper.

2 Example

In this section, we use the example code shown in Figures 1(b) and 1(c) to define several terms and notations (summarized in Figure 1(a)) used throughout the paper. We also provide a high-level overview of our framework using the example code.

API errors. All APIs in the example code are shown in bold font. In Figure 1(c), `InitAAText` and `EndAAText` are *user-defined procedures*. In the figure, user-defined procedures are shown in italicized font. The user-defined procedure in which an API is invoked is called the *enclosing procedure* for the API. In Figure 1(c), `EndAAText`, for instance, is the enclosing procedure for the APIs `XftDrawDestroy` (Line 27), `XftFontClose` (Line 28), and `XftColorFree` (Line 29). APIs can fail because of stressful environment conditions. In procedural languages such as C, API failures are indicated through *API errors*. API errors are special return values of the API (such as `NULL`) or distinct `errno` flag values (such as `ENOMEM`) indicating failures. For example, in Figure 1(b), API `recvfrom` returns a negative integer on failures. The API error from `recvfrom` is reflected by the return variable `cc`. APIs that can fail with errors are called as *critical APIs*. A condition checking of API return values or `errno` flag in the source code against API errors is called *API-Error Check (AEC)*; we use $AEC(a)$ to denote AEC of API *a*. For example, $AEC(recvfrom)$ is `if(cc<0)`.

Error block. The block of code following an API-error check, which is executed if the API fails is called the *error block*. Error blocks contain error-handling code to handle API failures. We use $EB(a)$ to denote the error block of API *a*. For example, Lines

Definitions and Acronyms			Specification (S)	Error-Check Specification (ErCS)
Library Application Program Interface (API)				Multiple-API Specification (MAS)
API-Error Check (AEC). AEC(a) is the required error check for API a.				Error-Check Violation (ErCV)
Error Block (EB). EB(a) is the error block of API a. AEC(a) precedes EB(a).			Violation (V)	Multiple-API Violation (MAV)
Path (P)	Error Path (ErP)	Error Exit-Path (ErExP)	<pre> ~X11-R6.9.0/x11perf/do_text.c 1 #include <X11/Xft/Xft.h> 2 ... 3 static XftFont *aafont; 4 static XftDraw *aadraw; 5 static XftColor aacolor; 6 ... 7 int InitAAText(XParms xp, Parms p, int reps){ 8 ... 9 aafont = XftFontOpenName (...); 10 if (aafont == NULL) { 11 ... 12 return 0; 13 } 14 aadraw = XftDrawCreate (...); 15 if (!XftColorAllocValue (... , &aacolor)){ 16 ... 17 XftFontClose (xp->d, aafont); 18 XftDrawDestroy (aadraw); 19 ... 20 return 0; 21 } 22 ... 23 } 24 ... 25 void EndAAText(XParms xp, Parms p){ 26 ... 27 XftDrawDestroy (aadraw); 28 XftFontClose (xp->d, aafont); 29 XftColorFree (... , &aacolor); 30 ... 31 } </pre>	
	Normal Path (NP)	Error Return-Path (ErRP)		
Trace (T)	Error Trace (ErT)	Error Exit-Trace (ErExT)		
	Normal Trace (NT)	Error Return-Trace (ErRT)		

(a) Definitions and Acronyms

```

~Redhat-9.0/routed/ripquery/query.c
1 #include <sys/socket.h>
2 int main(...){
3 ...
4 s = socket(...);
5 ...
6 cc = recvfrom(s, ...)
7 ...
8 if (cc < 0){
9 ...
10 close(s);
11 exit(1);
12 }
13 ...
14 close(s)
15 ...
16 }

```

(b) Example code from Redhat-9.0/routed-0.17-14

```

~X11-R6.9.0/x11perf/do_text.c
1 #include <X11/Xft/Xft.h>
2 ...
3 static XftFont *aafont;
4 static XftDraw *aadraw;
5 static XftColor aacolor;
6 ...
7 int InitAAText(XParms xp, Parms p, int reps){
8 ...
9 aafont = XftFontOpenName (...);
10 if (aafont == NULL) {
11 ...
12 return 0;
13 }
14 aadraw = XftDrawCreate (...);
15 if (!XftColorAllocValue (... , &aacolor)){
16 ...
17 XftFontClose (xp->d, aafont);
18 XftDrawDestroy (aadraw);
19 ...
20 return 0;
21 }
22 ...
23 }
24 ...
25 void EndAAText(XParms xp, Parms p){
26 ...
27 XftDrawDestroy (aadraw);
28 XftFontClose (xp->d, aafont);
29 XftColorFree (... , &aacolor);
30 ...
31 }

```

(c) Example code from X11-R6.9.0/x11perf

Fig. 1. Terminologies and example code

9-11 in Figure 1(b), Lines 11-12 and 16-20 in Figure 1(c) represent EB(recvfrom), EB(XftFontOpenName), and EB(XftColorAllocValue), respectively. A given API can have multiple error blocks depending on the different ways that it can fail (not shown in the examples for simplicity).

Paths, Traces, and Scenarios. A control-flow path exists between two program *points* if the latter is reachable from the former through some set of control-flow edges, i.e., Control Flow Graph (CFG) *edges*. Our framework identifies two types of paths - *error path* and *normal path*. There are two types of error paths. Any path from the beginning of the main procedure to an exit call (such as `exit`) in the error block of some API is called the *error exit-path*. For example, all paths ending at the `exit` call at Line 11 in Figure 1(b) are error exit-paths (`exit` call inside EB(recvfrom)). Any path from the beginning of the main procedure to a return call in the error block of some API is called the *error return-path*. For example, in Figure 1(c), all paths ending at the return call at Lines 12 (return call inside EB(XftFontOpenName)) and 20 (return call inside EB(XftColorAllocValue)) are error return-paths. Error exit-paths and error return-paths are together known as *error paths*. A *normal path* is any path from the beginning of the main procedure to the end of the main procedure without any API errors. For example, any path from Line 3 to Line 15 in Figure 1(b) is a normal path. For a given path, a trace is the print of all statements that exist along that path. Error paths,

error exit-paths, error return-paths, and normal paths have corresponding traces: *error traces*, *error exit-traces*, *error return-traces*, and *normal traces*. Error exit-traces and error return-traces are together known as error traces. Two APIs are *related* if they manipulate at least one (or more) common variable(s). For example, in Figure 1(b), APIs `recvfrom` and `close` are related to API `socket`. The `socket` API *produces* `s`, which is *consumed* by the APIs `recvfrom` and `close`. A *scenario* is a set of related APIs in a given trace. A given trace can have multiple scenarios. For example, if there were multiple `socket` calls in Figure 1(b), then each `socket` call, along with its corresponding related APIs, forms a different scenario.

API error-handling specifications. We identify two types of API error-handling specifications that dictate correct error handling along all paths in a program: *error-check specifications* and *multiple-API specifications*. Error-check specifications dictate that correct AEC(*a*)’s (API-Error Checks) exist for each API *a* (which can fail), before the API’s return value is *used* or the main procedure returns. For a given API *a*, the absence of AEC(*a*) causes an *error-check violation*. Multiple-API specifications dictate that the right *clean up* APIs are called along all paths. Clean up APIs are APIs called, generally before a procedure’s return or program’s exit, to free resources such as memory, sockets, pipes, and files or to *rollback* the state of a global resource such as the system registry and databases. For example, in Figure 1(c), `XftFontClose` (Line 17) and `XftDrawDestroy` (Line 18) are the clean up APIs in `EB(XftColorAllocValue)`. In Figure 1(c), one error-check specification (the return value of `XftColorAllocValue` should be checked against `NULL`) and two multiple-API specifications (`XftFontOpenName` should be followed by `XftFontClose`, and `XftDrawCreate` should be followed by `XftDrawDestroy`) are evident. Violation of a multiple-API specification along a given path is a *multiple-API violation*. Multiple-API violations along error exit-paths could be less serious as the operating system might reclaim unfreed memory and resource handlers along program exits. However, there are several cases where explicit clean up is necessary even on program exits. For instance, unclosed files could lose recorded data along an error exit-path if the buffers are not flushed out to the disk. In addition, any user-defined procedure altering a global resource (such as the system registry or a database) should *rollback* along error exit-paths to retain the integrity of the global resource. Next, we present the high-level overview of our framework using the example code.

The only input to our framework is the compilable source code of software package(s) implemented in C. To mine specifications, our framework initially distinguishes and generates API error traces and normal traces, for reasons explained later. Our framework then detects API error-handling violations in the source code using the mined specifications. In particular, our framework consists of the following three stages:

Error/normal trace generation. The trace generation stage distinguishes and generates error traces (error exit-traces and error return-traces) and normal traces inter-procedurally. Along normal paths, it is difficult to distinguish clean up APIs from other APIs. Hence, our framework identifies *probable* clean up APIs from the error traces. For example, in Figure 1(b), our framework identifies the API `close` (Line 10) from the error exit-trace that goes through `EB(recvfrom)`. In Figure 1(c), our framework identifies `XftFontClose` (Line 17) and `XftDrawDestroy` (Line 18) from the error

return-trace that goes through `EB(XftColorAllocValue)`. Note that, in Figure 1(c), the clean up APIs can also be invoked through the user-defined procedure `EndAAText`, inter-procedurally. However, even in the error block, there could be other APIs that are not necessarily clean up APIs (hence the term, *probable*). The final set of actual clean up APIs and the APIs related to them are determined during the specification mining stage.

Specification mining. The specification mining stage generates error-check specifications and multiple-API specifications. Our framework mines error-check specifications from error traces by determining API-error checks (AEC) for each API. For example, our framework determines `AEC(recvfrom)` to be `if (cc < 0)` from the error-exit trace that goes through `EB(recvfrom)`. Programmers often make mistakes along API error paths. Hence, proper clean up, being common among error paths and normal paths, should be mined from normal traces instead of error traces. Once probable clean up APIs are mined from error traces, our framework mines APIs that might be related to the probable clean up APIs from normal traces. For example, in Figure 1(c), our framework determines from normal traces that `XftFontClose` is related to `XftFontOpenName`, and `XftDrawDestroy` is related to `XftDrawCreate` (Figure 1(c), however, does not show normal paths or traces for simplicity). Our framework generates multiple-API specifications by applying sequence mining on normal traces.

Verification. Our static verifier uses the mined specifications (error-check and multiple-API specifications) to detect violations (error-check and multiple-API violations) in the source code. Next, we present our framework in detail.

3 Framework

The algorithm presented in Figure 2 shows the details of our framework. There are 3 stages and 10 steps (numbered 1-10) in our algorithm. Section 3.1 describes the error/normal trace generation stage (Steps 1-6). Section 3.2 (Steps 7-8) explains the steps involved in mining API error-handling specifications from the static traces. Finally, Section 3.3 describes the verification stage for detecting API error-handling violations of the mined specifications (Steps 9-10). Our framework adapts a trace generation technique developed in our previous work to generate static traces representing different API run-time behaviors. The trace generation technique uses *triggers* to generate static traces. Triggers are represented using finite state machines. The static traces generated by the trace generation technique with a given trigger depend on the transitions in the trigger. Readers may refer to our previous work [2] for further details.

3.1 Error/Normal Trace Generation

In this section, we explain how we adapt the trace generation technique [2] for generating API error and normal traces from source code. As shown in Figure 2, the error/normal trace generation stage has six steps: generate error traces (Step 1), process error traces (Steps 2-4), identify critical APIs and probable clean up APIs from error traces (Step 5), and finally, generate normal traces (Step 6). The various steps are explained next.

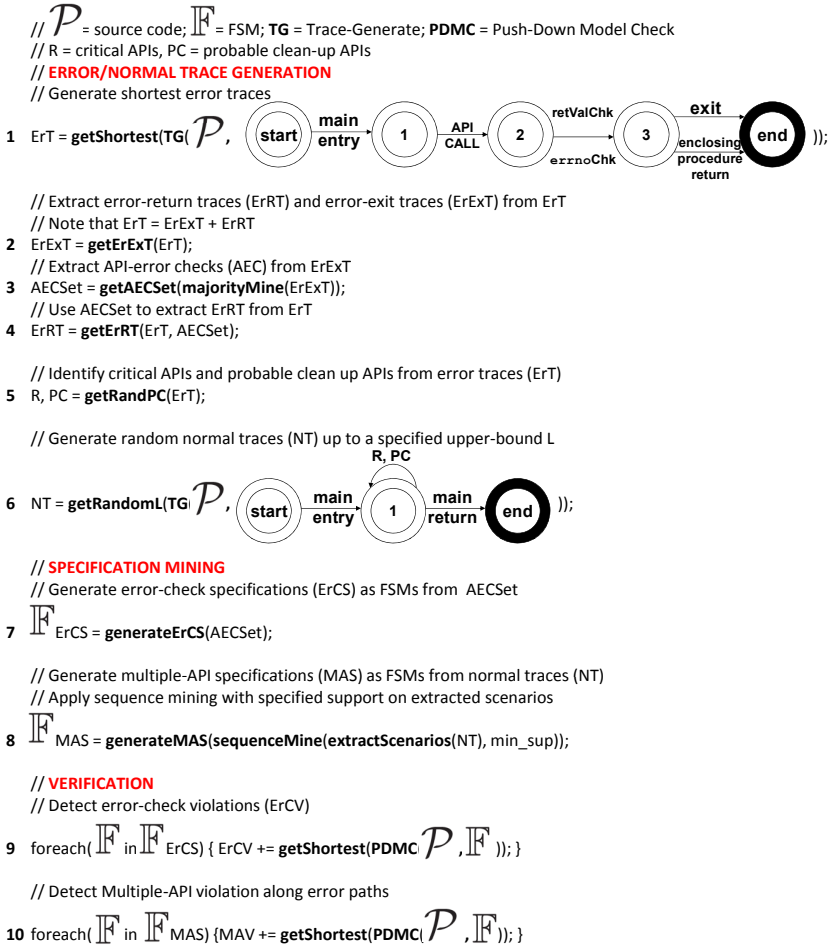


Fig. 2. The algorithm for mining API error-handling specifications

Step 1 - Generate error traces. An error trace starts from the beginning of the main procedure and ends in some API error-block with an exit call (causing the program to exit) or a return call (causing the enclosing procedure to return). The trigger FSM, say $\mathbb{F}$ (Step 1, Figure 2), is used by our trace generator (procedure **TG** in the figure) to generate error traces from the program source code ($\mathcal{P}$). The procedure **TG** represents our trace generation technique, which adapts the push-down model checking (**PDMC**) process. Transitions **retValChk** and **errnoChk** in the trigger $\mathbb{F}$ (from State 2 to State 3) identify the return-value check and error-flag check, respectively, for the API. Transitions from State 3 to the final state (State **end**) in the trigger $\mathbb{F}$ capture code blocks following the **retValChk** or **errnoChk** in which the program exits or the enclosing procedure returns. The procedure **TG** generates all traces in $\mathcal{P}$ that satisfy the trigger $\mathbb{F}$. However, the procedure **getShortest** (Step 1, Figure 2) returns only the shortest

trace from the set of all traces generated by TG . As we are interested only in the API-error check and the set of probable clean up APIs (PC) in the API error block for a given API from error traces, the program statements prior to the API invocation are not needed. Hence, it suffices to generate the shortest path for each API invocation with a following `retValChk` or `errnoChk`. If there are multiple `retValChk` or `errnoChk` for an API call site, then our framework generates the shortest trace for each of the checks. The trigger $\mathbb{F}$ captures the elements of `retValChk`, `errnoChk`, and the code block after these checks, even if these elements are scattered across procedure boundaries. However, the traces generated by this step can also have traces where `retValChk` or `errnoChk` is followed by a normal return of the enclosing procedure. Such traces, which are not error traces, are pruned out in the next step.

Steps 2, 3, and 4 - Process error traces. Our framework easily extracts error exit-traces from error traces (procedure `getErExt`, Step 2, Figure 2): error traces that end with an exit call are error exit-traces. We assume that the API `retValChk` or `errnoChk`, which precedes an exit call in an error-exit trace, is an API-error check. We then distinguish between the *true* and *false* branches of the API-error check. For example, in Figure 1(b), since `exit(...)` appears in the true branch of `AEC(recvfrom)` (`if(cc<0)`), we assume that `<0` is the error return value (API error) of `recvfrom`. For each API, our framework records API-error check with majority occurrences (procedure `majorityMine`, Step 3, Figure 2) among error exit-traces (procedure `getAECSet`, Step 3, Figure 2). As mentioned in the previous step, the traces generated in Step 1 can also have traces where `retValChk` or `errnoChk` is followed by a normal return of the enclosing procedure. Our framework uses the API-error check set computed from error exit-traces to prune out such traces to generate error return-traces (procedure `getErRT`, Step 4, Figure 2).

Step 5 - Identify critical APIs and probable clean up APIs from error traces. Our framework computes the set R (critical APIs) and the set PC (probable clean up APIs) in this step (procedure `getRandPc`, Step 5, Figure 2). The set R of critical APIs is easily computed from error exit-traces and error return-traces. A key observation here is that it is much easier to find clean up APIs along error paths than normal paths. It is because, on API failures, before the program exits or the enclosing procedure returns, the primary concern is clean up. Along normal paths, however, it is difficult to separate clean up APIs from other APIs. Hence, our framework identifies probable clean up APIs (the set PC) from the error traces. The term *probable* indicates that the APIs that occur in error blocks need not always be clean up APIs. The mining phase prunes out the non-clean-up APIs from the set PC . In the next step, we show how our framework identifies APIs related to the probable clean up APIs. These related APIs occur prior to API-error checks in the source code.

Step 6 - Generate normal traces. A normal trace starts from the beginning of the `main` procedure and ends at the end of the `main` procedure. The procedure TG uses the trigger FSM, say $\mathbb{F}$ (Step 6, Figure 2), to generate normal traces from the program source code ($\mathcal{P}$). The edges for State 2 in the trigger $\mathbb{F}$ are critical (set R) and probable clean up APIs (set PC). Our framework generates normal traces (involving critical and probable clean up APIs) randomly up to a user-specified upper bound L (procedure `getRandomL`, Step 6, Figure 2), inter-procedurally. The traces contain the probable clean up APIs and

the APIs related to them, if any. Finally, as API error-handling specifications can be conditional, the clean up for an API might depend on the actual return value of the API. As a simple example, for the `malloc` API, the `free` API is called only along paths in which the return value of `malloc` is not `NULL` (condition). Hence, normal paths (normal traces) are associated with their corresponding conditions involving API return values. The conditions, along with API sequences, form a part of normal traces and are used in the specification mining stage, explained next.

3.2 Specification Mining

The specification mining stage mines error-check and multiple-API specifications from the static traces (Steps 7-8). The scenario extraction and sequence mining are performed in Step 8.

Step 7 - Mine error-check specifications. Our framework generates error-check specifications (procedure `generateErCS`, Step 7, Figure 2) as Finite State Machines (FSM, $\mathbb{F}_{ErCS}$) from the mined API-error check set. The FSMs representing the error-check specifications specify that each critical API should be followed by the correct error checks.

Step 8 - Mine multiple-API specifications. Our framework mines multiple-API specifications from normal traces (procedure `generateMAS`, Step 8, Figure 2) as FSMs ($\mathbb{F}_{MAS}$). Normal traces include the probable clean up APIs (PC), APIs related to the set PC, and the conditions (involving API return values). The main observation used in mining multiple-API specifications from normal traces is that programmers often make mistakes along error paths [4, 10, 14, 17]. Hence, our framework mines related APIs from only normal traces and not from error traces. However, a single normal trace generated by the trace generator might involve several API scenarios, being often interspersed. A scenario (see Section 2) is a set of related APIs in a given trace. Our framework separates different API scenarios from a given normal trace, so that each scenario can be fed separately to our miner. We use a scenario extraction algorithm (procedure `extractScenarios`, Step 8, Figure 2) [2] that is based on identifying *producer-consumer* chains among APIs in the trace. The algorithm is based on the assumption that an API and its corresponding clean up APIs have some form of data dependencies between them such as a producer-consumer relationship. Each producer-consumer chain is generated as an independent scenario. For example, in Figure 1(c), the API `XftFontOpenName` (Line 9) produces `aafont`, which is consumed by the API `XftFontClose` (Line 17). The APIs `XftFontOpenName` and `XftFontClose` are generated as an independent scenario.

Our framework mines multiple-API specifications from independent scenarios using frequent-sequence mining (procedure `sequenceMine`, Step 8, Figure 2). Let IS be the set of independent scenarios. We apply a frequent sequence-mining algorithm [15] on the set IS with a user-specified support min_sup ($min\_sup \in [0, 1]$), which produces a set FS of frequent sequences that occur as subsequences in at least $min\_sup \times |IS|$ sequences in the set IS . Note that our framework can mine the different error-handling specifications for the different errors of a given API as long as the different specifications have enough support among the analyzed client code.

3.3 Verification

Our framework uses the specifications to find API error-handling violations (Steps 9-10).

Steps 9 and 10 - Detect error-check and multiple-API violations. In Steps 1 and 6, we adapt the push-down model checking (PDMC) process for trace generation by the procedure `TG`. Here we use the PDMC process for property verification. The specifications mined by our framework as FSMs ($\mathbb{F}_{ErCS}$ and $\mathbb{F}_{MAS}$) represent the error-handling properties to be verified at this stage. Our framework verifies the property FSMs in $\mathbb{F}_{ErCS}$ and $\mathbb{F}_{MAS}$ against the source code ($\mathcal{P}$). The mined specifications can also be used to verify the correct API error handling in other software packages. For verifying conditional specifications, we adapt the PDMC process to track the value of variables that take the return value of an API call along the different branches of conditional constructs. Our framework generates (procedure `getShortest`) the shortest path for each detected violation (i.e., a potential defect) in the program, instead of all violating traces, thus making defect inspection easier for the users.

4 Evaluation

To generate static traces, we adapted a publicly available model checker called MOPS [6] with procedures (Steps 1-10) shown in Figure 2. We used BIDE [15] to mine frequent sequences. We have applied our framework on 10 packages from the Redhat-9.0 distribution (52 KLOC), postfix-2.0.16 (111 KLOC), and 72 packages from the X11-R6.9.0 distribution (208 KLOC). The analyzed packages use the APIs from the POSIX and X11 libraries. We selected POSIX and X11 clients because the POSIX standard [1] and the Inter-Client Communication Conventions Manual (ICCCM) [13] from the X Consortium standard were readily available. These standards describe rules for how well-behaved programs should use the APIs, serving as an oracle for confirming our mined results. We ran our evaluation on a machine with Redhat Enterprise Linux version 2.6.9-5ELsmp, 3GHz Intel Xeon processor, and 4GB RAM. For specification mining and violation detection, the analysis cost ranges from under a minute for the smallest package to under an hour for the largest one. We next explain the evaluation results (summarized in Figure 3(a)) for the various stages of our framework.

Trace generation. The number of error exit-traces and error return-traces generated by our framework are shown in Columns 3 (**ErExT**) and 4 (**ErRT**) of Figure 3, respectively. To evaluate trace generation, we manually inspected the source code for each error exit-trace produced by our framework and each error exit-trace missed by our framework. Error exit-traces missed by our framework can be determined by manually identifying the exit statements in the analyzed program not found in any of the generated error exit-traces. There are five sub-columns in Column 3 (**ErExT**): Σ (total number of error exit-traces generated or missed by our framework), Σ^{op} (total number of error exit-traces actually generated by our framework), $\mathbf{FN} = \Sigma - \Sigma^{op}$ (total number of error exit-traces missed by our framework), **FP** (false positives: generated traces that are not actually error exit-traces), and **IP** (inter-procedural: the number of traces in which the API invocation, API-error check, and error blocks were scattered across procedure boundaries).

1. Packages		2. LOC	3. ErExT					
			$\sum$	$\sum^{op}$	$FN = \sum - \sum^{op}$	FP	IP	
10-Redhat-9.0-pkgs		52 K	338	320	18		35	18
postfix-2.0.16		111 K	124	92	32		3	124
X11-R6.9.0		208 K	286	248	38		27	164
$\sum$		371 K	748	660	88 (12%)		65(10%)	306(41%)
4. ErRT	5. ErCS		6. ErCV		7. MAS		8. MAV	
	$\sum$	FP	$\sum$	FP	$\sum$	FP	$\sum$	FP
205	31	3	58	1	40	6	4	3
30	31	3	4	2	40	6	0	0
305	31	3	170	13	40	6	56	9
540	31	3(10%)	232	16(7%)	40	6(15%)	60	12(20%)

(a) Traces and violations

(R)XGetVisualInfo	(R)XpQueryScreens	(R)XpGetAttributes
XGetWindowProperty(12)	(R)XScreenResourceString	(R)XpGetOneAttribute
XQueryTree(5)	(R)XGetAtomName	(R)glXChooseVisual
(R)XFetchBytes	(R)malloc	XGetIMValues(3)
(R)XGetKeyboardMapping	XGetWMPprotocols(3)	(R)XGetWMHints

(b) Multiple-API specifications for the clean up API
XFree, mined by our framework

Σ : Total, IP: Interprocedural, FP: False Positives, FN: False Negatives, ErExT: Error Exit-Traces, ErRT: Error Return-Traces, ErCS: Error-Check Specifications, ErCV: Error-Check Violations, MAS: Multiple-API Specifications, MAV: Multiple-API Violations

Fig. 3. Evaluation Results

We observed that the number of false negatives (FN) and false positives (FP) were low, at 12% (88/748) and 10% (65/660), respectively. The main reason for false negatives in the traces generated by our framework is the lack of aliasing and pointer analysis. For example, in `xkbvleds/utls.c`, the variable `outDpy` takes the return value of the API `XtDisplay`. Then the value of `outDpy` is assigned to another variable `inDpy`, and `inDpy` is compared to `NULL`. If `inDpy` is `NULL`, a user-defined procedure `uFatalError` is called, which then calls `exit`. Our framework did not capture the aliasing of `outDpy` to `inDpy`, and hence missed the trace. However, as the number of false negatives was low, our framework still generated enough traces for the mining process. Some of the traces generated by our framework were not error exit-traces, leading to false positives. For example, in `tftp/tftpd.c`, the variable `f` (process id) takes the return value of the API `fork`. The program exits on `f>0` (parent process; not an error). Although the trace was generated by our framework, it is not an error exit-trace (`fork` fails with a negative integer). However, as the number of false positives was low, false error exit-traces were pruned by the mining process. 41% (306/748) of all the error

exit-traces were scattered across procedure boundaries, highlighting the importance of inter-procedural trace generation. Specifically, all error exit-traces from the `postfix` package crossed procedure boundaries.

Our framework identifies the set of probable clean up APIs from the error traces (Step 5, Figure 2). After discarding string-manipulating APIs (such as `strcmp` and `strlen`), printing APIs (such as `printf` and `fprintf`), and error-reporting APIs (such as `perror`), which frequently appear (but unimportant) in error blocks, our framework identified 36 APIs as probable clean up APIs. Our framework used probable clean up APIs in generating normal traces. For each compilable unit in the analyzed packages, our framework randomly generated 20 normal traces, ensuring there are enough distinct traces for mining. Our framework discarded 14/36 APIs after mining the normal traces with one of the following reasons: (1) insufficient call sites and hence an insufficient number of traces to mine from (for example, the API `XEClearCtrlKeys` had only two traces), (2) no temporal dependencies with any APIs called prior to the error block (for example, the API `XtSetArg` appears in an exit trace from `xlogo/xlogo.c`. However, `XtSetArg` does not share any temporal dependencies with APIs called prior to the exit block), or (3) insufficient support among the scenarios. Our framework mined 40 multiple-API specifications from the remaining 22/36 probable clean up APIs (Column 7, MAS).

Error-check specifications. Our framework mined error-check specifications for only those APIs that occur more than three times among the error traces. In all, our framework mined 31 error-check specifications (Column 5, **ErCS**) from the error traces across all the analyzed packages. 3 (10%) out of the 31 (subcolumn Σ) mined specifications were false positives (subcolumn **FP**). For example, the API `geteuid` returns the effective user ID of the current process. The effective ID corresponds to the set ID bit on the file being executed [1]. Our framework encounters `geteuid() != 0` at least 5 times among error traces leading to a false error-check specification – ‘*geteuid fails by returning a non-zero integer*’. But, a non-zero return value simply indicates an unprivileged process.

Error-check violations. The error-check specifications mined from error traces are used in detecting error-check violations along the error paths in the analyzed software packages. Column 6 (**ErCV**) of Figure 3(a) presents the number of error-check violations detected by our framework. We manually inspected the violations reported by our framework. 16 (7%) out of the 232 (subcolumn Σ) reported error-check violations were false positives (subcolumn **FP**). The main reason for false positives in the reported violations is, once again, the lack of aliasing and pointer analysis in our framework. For example, in `twm/session.c` and `smproxy/save.c`, the variable `entry` takes the return value of `malloc`. Then the variable `entry` is assigned to another variable `pentry`. The variable `pentry` is then checked for `NULL`, which was missed by our framework.

Multiple-API specifications. Our framework mines multiple-API specifications from normal traces. Our framework produces a pattern as a multiple-API specification if the pattern occurred in at least five scenarios, with a minimum support (*min_sup*) of 0.8 among the scenarios. Our framework mined 40 multiple-API specifications (Column 7, MAS) across all the packages, with 6 (15%) of them being false positives (subcolumn **FP**). All multiple-API specifications mined by our framework were conditional – the

clean up APIs in conditional multiple-API specifications depend on the return value or a parameter (that holds the return value). As an example of a conditional specification, for the API `XGetVisualInfo`, cleaning up through the API `XFree` is necessary only if the fourth input parameter of `XGetVisualInfo` (the number of matching *visual structures*) is non-zero. False positives among the mined specifications may occur if some patterns occurring in the analyzed source code are not necessarily specifications. This result is a limitation shared by all mining approaches, requiring human inspection and judgement to distinguish real specifications from false ones. For example, our framework considered the APIs `XSetScreenSaver` and `XUngrabPointer` as probable clean up APIs, as both APIs appeared in some error traces generated by our framework. The first parameter of both these APIs is the display pointer produced by the API `XOpenDisplay`. Hence, our framework mined the property “`XSetScreenSaver` and `XUngrabPointer` should follow `XOpenDisplay`”, leading to a false positive. The number of false specifications mined by our framework is low as the code bases used by our framework for mining are sufficiently large.

Our framework mines the maximum number of multiple-API specifications around the clean up API `XFree`. From the static traces, 35 APIs from the X11 library were found to interact with the `XFree` API, leading to 15 multiple-API specifications with sufficient support. The specifications mined around the API `XFree` are shown in Figure 3(b). `XFree` is a general-purpose X11 API that frees the specified data. `XFree` must be used to free any objects that were allocated by X11 APIs, unless an alternate API is explicitly specified for the objects [13]. The pointer consumed by the `XFree` API can either be a return value or a parameter (that holds the return value) of some X11 API. The “(R)” in `(R)XGetVisualInfo`, for instance, indicates that the return value of the API `XGetVisualInfo` should be freed through the API `XFree` along all paths. The “(5)” in `XQueryTree(5)`, for instance, indicates that the fifth input parameter of the API `XQueryTree` should be freed through the API `XFree` along all paths.

Multiple-API violations. Our framework uses the multiple-API specifications mined from normal traces to detect multiple-API violations in the analyzed software packages. Column 8 (MAV) presents the number of multiple-API violations detected by our framework. We manually inspected the violations reported by our framework. 12 (20%) out of the 60 (subcolumn Σ) reported multiple-API violations were false positives (subcolumn FP). To verify conditional specifications, we adapted MOPS to track the value of variables that take the return value of an API call along the different branches of conditional constructs. Tracking API return values while verifying multiple-API specifications decreases the number of false positives, which would have otherwise been reported. As a simple example, verifying conditional specifications causes false positives such as “a file is not closed before the program exits on the failure (NULL) path of the `open` API” not to be reported. Verifying conditional specifications by tracking return values avoided 87 false positives in the analyzed packages, which would have otherwise been reported. In all, our framework mines 62 error-handling specifications and detects 264 real error-handling violations in the analyzed packages. Due to pointer-insensitive analysis, our framework might not mine all the error-handling specifications or detect all the error-check and multiple-API violations in the analyzed software packages, leading to false negatives. For the mined specifications and the detected violations,

we have not quantified the false negatives of our framework. Quantifying the violations missed by our framework (through manual inspection of source code along all possible paths in the presence of function pointers and aliasing) is difficult and error prone.

5 Related Work

Dynamic. Previous work has mined API properties from program execution traces. For example, Ammons et al. [3] mine API properties as probabilistic finite state automata from execution traces. Perracotta developed by Yang et al. [18] mines temporal properties (in the form of pre-defined templates involving two API calls) from execution traces. Different from these approaches, our framework mines specifications from source code of API clients. Dynamic approaches require setup of runtime environments and availability of sufficient system tests that exercise various parts of the program and hence the violations might not be easily exposed. In contrast, our new framework mines API error-handling specifications from static traces without suffering from the preceding issues.

Static. Previous several related static approaches developed by other researchers also mine properties from source code for finding defects. Engler et al. propose *Meta-level Compilation* [8] to detect rule violations in a program based on user-provided, simple, system-specific compiler extensions. Their approach detects defects by statically identifying inconsistencies in commonly observed behavior. PR-Miner developed by Li and Zhou [11] mine programming rules as frequent *itemsets* (ordering among program statements is not considered) from source code. Apart from being intra-procedural, neither approach considers data-flow or control-flow dependences between program elements, required for mining error-handling specifications. Two recent approaches in static specification mining, most related to our framework, are from Chang et al. [5] and Ramanathan et al. [12]. Chang et al.'s approach [5] mines specifications as *graph minors* from *program dependence graphs* by adapting a frequent sub-graph mining algorithm. Specification violations are then detected by their heuristic graph-matching algorithm. The scalability of their approach is limited by the underlying graph mining and matching algorithms. Furthermore, their approach does not mine conditional specifications. Ramanathan et al. [12] mine preconditions of a given procedure across different call sites. To compute preconditions for a procedure, their analysis collects predicates along each distinct path to each procedure call. As their approach is not applicable to postconditions, it cannot mine error-handling specifications. Static approaches [17, 16] exist to analyze programs written in Java, which has explicit exception-handling support. Several proposals [9] exist for extending C with exception-handling support. In contrast, our framework is applicable to applications implemented in procedural languages with no explicit support for exception handling.

6 Conclusions

We have developed a framework to automatically mine API error-handling specifications from source code. We then use the mined specifications to detect API error-handling violations from the analyzed software packages (API client code). We have

implemented the framework, and validated its effectiveness on 10 packages from the `Redhat-9.0` distribution (52 KLOC), `postfix-2.0.16` (111 KLOC), and 72 packages from the `x11-R6.9.0` (208 KLOC). Our framework mines 62 error-handling specifications and detects 264 real error-handling defects from the analyzed packages.

References

1. IEEE Computer Society. IEEE Standard for Information Technology - Portable Operating System Interface POSIX - Part I: System Application Program Interface API, IEEE Std 1003.1b-1993 (1994)
2. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: From usage scenarios to specifications. In: Proc. ESEC/FSE, pp. 25–34 (2007)
3. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: Proc. POPL, pp. 4–16 (2002)
4. Bruntink, M., Deursen, A.V., Tourwe, T.: Discovering faults in idiom-based exception handling. In: Proc. ICSE, pp. 242–251 (2006)
5. Chang, R.Y., Podgurski, A.: Finding what's not there: A new approach to revealing neglected conditions in software. In: Proc. ISSTA, pp. 163–173 (2007)
6. Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties of software. In: Proc. CCS, pp. 235–244 (2002)
7. Cristian, F.: Exception Handling and Tolerance of Software Faults. In *Software Fault Tolerance*, ch. 5. John Wiley and Sons, Chichester (1995)
8. Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proc. SOSP, pp. 57–72 (2001)
9. Gehani, N.H.: Exceptional C for C with exceptions. *Software Practices and Experiences* 22(10), 827–848 (1992)
10. Gunawi, H., Rubio-Gonzalez, C., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Liblit, B.: EIO: Error handling is occasionally correct. In: Proc. USENIX FAST, pp. 242–251 (2006)
11. Li, Z., Zhou, Y.: PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Proc. ESEC/FSE, pp. 306–315 (2005)
12. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: Proc. PLDI, pp. 123–134 (2007)
13. Rosenthal, D.: *Inter-client communication Conventions Manual (ICCCM)*, Version 2.0. X Consortium, Inc. (1994)
14. Toy, W.: Fault-tolerant design of local ESS processors. In: *The Theory and Practice of Reliable System Design*. Digital Press (1982)
15. Wang, J., Han, J.: BIDE: Efficient mining of frequent closed sequences. In: Proc. ICDE, pp. 79–90 (2004)
16. Weimer, W., Necula, G.C.: Finding and preventing run-time error handling mistakes. In: Proc. OOPSLA, pp. 419–431 (2004)
17. Weimer, W., Necula, G.C.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) *TACAS 2005*. LNCS, vol. 3440, pp. 461–476. Springer, Heidelberg (2005)
18. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: Mining temporal API rules from imperfect traces. In: Proc. ICSE, pp. 282–291 (2006)

SNIFF: A Search Engine for Java Using Free-Form Queries

Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen

EECS Department, University of California, Berkeley, CA, USA
{shaunakc,sjuvekar,ksen}@cs.berkeley.edu

Abstract. Reuse of existing libraries simplifies software development efforts. However, these libraries are often complex and reusing the APIs in the libraries involves a steep learning curve. A programmer often uses a search engine such as Google to discover code snippets involving library usage to perform a common task. A problem with search engines is that they return many pages that a programmer has to manually mine to discover the desired code. Recent research efforts have tried to address this problem by automating the generation of code snippets from user queries. However, these queries need to have type information and therefore require the user to have a partial knowledge of the APIs.

We propose a novel code search technique, called SNIFF, which retains the flexibility of performing code search in plain English, while obtaining a small set of relevant code snippets to perform the desired task. Our technique is based on the observation that the library methods that a user code calls are often well-documented. We use the documentation of the library methods to add plain English meaning to an otherwise undocumented user code. The annotated user code is then indexed for the purpose of free-form query search. Another novel contribution of our technique is that we take a type-based intersection of the candidate code snippets obtained from a query search to generate a set of small and highly relevant code snippets.

We have implemented SNIFF for Java and have performed evaluations and user studies to demonstrate the utility of SNIFF. Our evaluations show that SNIFF performed better than most of the existing online search engines as well as related tools.

1 Introduction

Java’s evolution and growth over the years has greatly increased the number of APIs available at a programmer’s disposal. For example, the Java Standard Library, J2SE, contains thousands of classes and more than 20,000 methods [12]. The Java APIs are often designed in a modular manner using small composable units. A programmer needs to combine these APIs using some (often complicated) sequence of classes and method calls, to correctly use a Java library. This fact, compounded with the sheer number of APIs, makes it difficult for a programmer to discover *what* APIs he/she wants and *how* to use those APIs to perform a given programming task. For example, consider a programmer who

wants to read from a file and is unfamiliar with the `java.io` package. Without prior knowledge about the `java.io` package, the programmer will not only find it difficult to discover what classes he/she needs to use, but also to figure out how to use the methods in those classes to read from a file. Even if the programmer manages to write code to read from a file after digging into the `java.io` API, the code written may not be efficient—the code may only use the class `java.io.FileReader` and ignore the use of the `java.io.BufferedReader` class, which is required for an efficient implementation.

In order to solve the above problem, a programmer generally resorts to one of the following two techniques. He/she might try to explore existing code bases and their documentations, and search for code snippets which perform the desired programming task using some APIs. This “mining” effort becomes quite tedious because existing code bases are often very large, making manual search impossible. Moreover, these code bases may not be well-documented making it difficult to locate the relevant code snippets. Some recent tools, such as Prospector [12], have tried to automate the search process; however, they assume that the programmer knows what classes and object types he/she wants to use. This may not be a realistic assumption if the programmer is unfamiliar with the class names.

Alternately, a programmer might search the web using some search engine such as Google. An advantage of using a search engine is that the programmer posts his/her query in plain English (free-form), such as “read from a file in Java” without any prior knowledge about the required packages, classes, or methods. However, the results returned by the search engines contain relevant code interspersed with irrelevant code and plain English text from the webpage. The programmer needs to determine, potentially involving further searches, what part of the returned code snippet is relevant. More recently, PARSEWeb [18] has tried to combine results from web searches with Prospector to improve the quality of search. However, the tool suffers from the same problem as Prospector—the programmer needs to know what classes and object types he/she has to use.

We propose a novel code search technique called `SNIFF`^{1,2}, which retains the flexibility of performing a search in English, while obtaining small and relevant code snippets required to perform the desired task. In `SNIFF`, a programmer issues a query expressing the programming task in English and `SNIFF` returns a small set of relevant code snippets. For example, `SNIFF` returns the code snippet in Table 1 for the query “read a line of text from a file”.

Table 1. Original code with no useful comments

```
FileReader fr = new FileReader(String fileName);
BufferedReader br = new BufferedReader(FileReader fr);
String line = br.readLine()
```

¹ `SNIFF` stands for S`NI`ppet for Free-Form queries.

² It “sniffs” the code database for the relevant code.

The key idea of SNIFF is *to combine API documentation with publicly available Java code*. Specifically, SNIFF takes a large amount of Java source code already available on the web and annotates it by appending each statement containing a method call with the method's Javadoc description (if available). Annotation allows SNIFF to add meaningful comments to otherwise uncommented Java files. SNIFF indexes these annotated Java files in a database. A query to SNIFF looks up this database and collects code chunks that match the query. SNIFF then performs a type-based intersection of these code chunks to retain the most relevant and common part of the code chunks. SNIFF also ranks these pruned chunks using their relevance to the query. We have defined one criterion for the relevance of the code snippets: the frequency of their occurrence in the indexed code base. Ranking in SNIFF is based on this measure of relevance.

SNIFF has several advantages:

1. SNIFF allows free-form English queries about a programming task. This eliminates the need to know the appropriate APIs beforehand.
2. SNIFF facilitates more effective code reuse by eliminating the requirement of much prior knowledge about APIs. Code reuse increases the performance and reliability of the new code.
3. Since SNIFF constructs the most relevant code snippet by performing type-based intersection of several Java code chunks, we get relatively mature and correct code.

We have developed an eclipse plugin for SNIFF and performed a user study that found that programmers could solve the reuse problems 40% faster with SNIFF than with other tools. We have also compared the performance of SNIFF with online code search engines [5,10,3] on a set of user queries posted on a Java developer's forum [8]. Our experiments show that SNIFF returned the most relevant code snippet as the top ranked result for about 88% of queries, whereas the online search engines returned the top result for about 50% of queries. Moreover, these results were buried inside hyperlinked source files and required substantial manual inspection to discover the exact snippets. We have also evaluated the importance of our intersection algorithm for the relevance of the snippets. Our experiments show that intersection helps in effective pruning and better ranking of the code snippets.

2 Overview

We give an overview of SNIFF using a simple example. Consider a Java programmer who is unfamiliar with the Java classes `Runtime` and `Process`. The

Table 2. Executing a system command in Java

<pre> Runtime r = Runtime.getRuntime(); Process p = r.exec(String command); </pre>

programmer wants to execute a system command such as `ls` from inside a Java program. The required code snippet is shown in Table 2.

The code is relatively difficult for the programmer to infer for several reasons. First, it is hard for a programmer unfamiliar with the Java APIs to figure out that the `Runtime` class is required to get the runtime and to execute the command in that runtime. It is even harder for the programmer to discover that the methods `getRuntime` and `exec` should be called in that order to first obtain the runtime and then to execute the command, respectively. Finally, an object of class `Process` is required to create a separate process and execute the command. In this situation, the programmer has a couple of options to discover the code snippet. (1) The programmer could search the web (e.g. using Google) by posting a query such as “execute command in Java.” (2) The programmer could use an existing code synthesizer such as Prospector [12] or PARSEWeb [18]. The results returned by web search would not often give the exact code snippet, but rather some large code fragments that have the relevant statements surrounded by other statements and non-Java text. The programmer then needs to manually examine such code fragments for the exact code snippet. The problem with the code synthesizers is that the programmer needs to know the object types (e.g. `Runtime` and `Process` in this case) that he/she wants to synthesize.

SNIFF retains the flexibility of performing the search in plain English (as in web search), while obtaining small code snippets (as in Prospector or PARSEWeb) that are relevant to the query. In particular, the programmer will type the query “execute command” in SNIFF and SNIFF will return a small set of concise and relevant code snippets, that would execute a system command from inside Java.

SNIFF works as follows: In a nutshell, SNIFF takes a large amount of publicly available Java source code and indexes [2] it in a database. A query to SNIFF examines this database and collects code chunks that match the query. SNIFF then performs a *Java syntax-aware* intersection of these code chunks and returns a small set of concise and relevant code-snippets. Although the above steps look trivial, each step presents a number of technical challenges. For example, during the indexing phase, what should be considered as keywords? A natural answer would be to consider the words in comments and method names as

Table 3. Intersection of Code Snippets

Candidate Codes	Result of Intersection
<pre>Runtime r = Runtime.getRuntime(); String command = "clear"; Process p = r.exec(command);</pre>	<pre>Runtime r = Runtime.getRuntime(); Process p = r.exec(String command);</pre>
<pre>Runtime r = Runtime.getRuntime(); flag = 1; // flag set if branch entered Process p = r.exec("ls");</pre>	

keywords. However, this straightforward approach does not work, because user codes usually have very few comments and the method names do not always reflect the actual functionality of the method. For example, the code in the first column of Table 3 does not reflect that it is meant for executing a system command.

We address this indexing problem in a novel way. We have observed that although the user of the `Runtime` class does not write any comment about the purpose of the code, the `Runtime` class is well-documented and can be used to annotate the user code to help us understand the purpose of the user-written code. For example, given the code chunk in Table 1, one can automatically annotate it with comments as shown in Table 4 by inserting the Javadoc description of a method after each statement that contains the method. For code in table 1, these Javadoc descriptions are collected from the `FileReader` and `BufferedReader` classes. This special method of annotating a user-written and possibly un-commented Java source file adds extra useful information about the user code. SNIFF subsequently indexes the annotated user code in a database for the purpose of query.

A query to SNIFF, like “execute command” searches this database and returns the consecutive lines of code from a single source class, that contain both the keywords *execute* and *command* of the query. Assume that the first column of Table 3 lists two such candidate codes returned for the query “execute command”. Any such candidate contains the relevant code along with possibly irrelevant code. The irrelevant code might either contain completely irrelevant information, like initialization of some arbitrary variable (e.g. `flag = 1;` in the second code snippet in Table 3) or might contain statements like `Runtime r = Runtime.getRuntime();` which are essential for correctness of the implementation but do not match to any query keyword. Our observation is that the irrelevant statements are dissimilar across the candidates, while relevant statements are similar (syntactically identical) in almost all of them. Therefore, we perform a *type-based intersection* of these candidates to extract the relevant statements out of them. The second column of Table 3 shows the intersection of the candidates from the first column. The comments inserted by SNIFF are not shown in Table 3 for convenience. Note that intersection retains the common statements of the two candidate codes, but removes the statements specific to each individual code. This intersection step is another novel contribution of this work and distinguishes SNIFF from other code search engines. Specifically, our intersection allows SNIFF to return a concise and relevant code snippet instead of a set of code chunks containing some irrelevant statements. We will explain our intersection algorithm in section 3.

Finally, there might be multiple code snippets that achieve the same programming purpose. In order to represent all possible relevant code snippets, we perform clustering to group similar snippets together. These clusters must be meaningfully ranked so that the most relevant snippets are displayed at the top. We have observed that the most relevant snippets are also the ones that are

implemented most frequently in our indexed code. Hence, we rank the clusters based on the number of constituent snippets.

3 Our Approach and Algorithm

In this section, we describe in details our approach for generating code snippets from a user query. We begin with formal definitions of user query, code chunk and code snippet.

Definition 1. A **user query** is defined as a sequence $q = (w_1, w_2, \dots, w_n)$, where each w_i is a string of characters. We call each w_i a *keyword*.

An example of a user query is (“execute”, “command”).

Definition 2. A **code chunk** C is defined as an ordered list of statements $s_1; s_2; \dots; s_n$ that occur in contiguous lines in some Java method body. A **code snippet** S is defined as an ordered list of statements $s'_1; s'_2; \dots; s'_m$ that occur in the same order (but not necessarily contiguous) in some Java method body. A code snippet might omit some intermediate statements from a method body.

Note that every code chunk is a code snippet, but not vice-versa. For example, in Table 3, the first column shows the code chunks while the second column shows a code snippet. We now describe the components of SNIFF.

Table 4. Annotated code generated by SNIFF

```
...
FileReader fr = new FileReader(fileName);    // File Reader
                                              // Creates new FileReader given file name read
BufferedReader br = new BufferedReader(fr);  // Buffered Reader
                                              // Creates character-input stream
                                              // uses input buffer specified size
...
```

3.1 Preprocessing: Parsing Open-Source Java Code

We assume that our codebase has a large collection of Java source code and that these Java classes contain sufficient number of examples on how to use various common Java APIs.

Most often, these Java classes, which we will call *client classes*, contain very few user comments. SNIFF annotates this client code by adding its own comments. This annotation is performed in a novel and automatic way. It first collects the Javadoc descriptions and user comments for every class and its methods defined in the standard Java API as well as in the client code. These comments are pruned to remove common word classes like prepositions, conjunctions and articles (called *stopwords*). SNIFF also performs a preliminary natural language processing in form of *stemming* [14] on all the keywords in the comment.

SNIFF then creates a map *MethToComments*: *Sig* $\rightarrow$ *Comment* from each method signature *Sig* to its Javadoc comments *Comment*, where a method signature contains the class name containing the method, the method name, the types of the method's arguments and the return type of the method. Finally, SNIFF performs a simplifying transformation of the client code to convert it into an *intermediate representation*. The grammar for this intermediate form is given in Table 5. The purpose of this transformation is to simplify the indexing and retrieval of the code snippets in the latter stages of the system.

Table 5. Intermediate representation for the client code

<code>stmt ::= var = expr var.field = expr var.method(var*)</code> <code> var = var.method(var*) if (expr) goto label</code> <code>expr ::= constant var var.field var op var</code>

3.2 Preprocessing: Annotating and Indexing the Client Code

After the previously described transformation, SNIFF performs the actual annotation using the *MethToComment* map as follows. SNIFF parses each client Java source file. For each statement that contains a method call `var.method(var*)` with signature, say **sig**, SNIFF adds the comment *MethToComment*(**sig**) at the end of the statement. Java coding convention recommends the use of descriptive method names, where the first character of every internal word is capitalized. (e.g. 'readLine'). Therefore, we also break the method name using these capitalizations to identify the breakpoints and add it as a part of the comment to the statement containing the method call. For example, `readLine` is broken up into `read` and `Line` and these two words are added as comments to any statement that contains the method call `readLine`. Thus, the client code in Table 1 gets converted to the commented code shown in Table 4.

SNIFF then indexes the commented code in a database. Our database schema contains two tables. The first table stores the individual statements in the client source files and the second table stores the tuples (*keyword*, *statementid*). A tuple for the string *keyword* gives the primary key of the statement whose comments contain *keyword*.

3.3 Responding to a Query

Obtaining Code Chunks using Database: SNIFF takes a user query as an input and applies the same stop-word removal and stemming to it as in the previous section. This modified query will be referred to as *q* for the remaining portion of the discussion. *q* is then treated as a *bag of words*, i.e. the ordering between the keywords is ignored. We search for each keyword *w_i* in *q* in the database and retrieve the code chunks that contain all keywords *w_i* in *q*. We enhance these code chunks by adding a few lines of code located before and after the keyword-matching region in the original source file.

Returning the code chunks “as is” is not useful because (with respect to the user query) these code chunks contain both relevant statements as well as irrelevant ones. We would like to return only those statements that are relevant to the user query. A natural way to return the relevant statements would be to only return those statements whose comments contain at least one keyword. However, we observed that some of the statements that do not contain any keyword in their comments are often the useful glue among the statements containing a keyword; therefore such statements cannot be classified as irrelevant. An example of this is in Table 3. The `Runtime` class and its method does not contain the comments relevant to “execute command”, but it is required for actually executing a system command as a `String`. We also observed that these relevant statements, unlike the irrelevant statements, are present in all code chunks. Therefore, to obtain the maximal relevant code snippets, we take an intersection of the code chunks obtained from the previous step.

We next give an algorithm to perform an intersection of any two code snippets. Since each code chunk is also a code snippet, the intersection operation will apply equally well to code chunks. Every statement in the code chunks returned by the database has the form given in Table 5. We define equivalence $\sim$ of two statements recursively as follows.

- $\text{var1} = \text{expr1} \sim \text{var2} = \text{expr2}$ iff $\text{typeOf}(\text{var1}) = \text{typeOf}(\text{var2})$ and $\text{expr1} \sim \text{expr2}$.
- $\text{var1.field1} = \text{expr1} \sim \text{var2.field2} = \text{expr2}$ iff $\text{typeOf}(\text{var1}) = \text{typeOf}(\text{var2})$, $\text{field1} = \text{field2}$, and $\text{expr1} \sim \text{expr2}$.
- $\text{var1.method1}(\text{vars1}*) \sim \text{var2.method2}(\text{vars2}*)$ iff $\text{typeOf}(\text{var1}) = \text{typeOf}(\text{var2})$, $\text{method1} = \text{method2}$, and $\text{typeOf}(\text{vars1}*) = \text{typeOf}(\text{vars2}*)$.
- $\text{if}(\text{expr1}) \text{ goto label1} \sim \text{if}(\text{expr2}) \text{ goto label2}$ iff $\text{expr1} \sim \text{expr2}$.
- $\text{label1} \sim \text{label2}$.
- $\text{var1 op1 var1}' \sim \text{var2 op2 var2}'$ iff $\text{typeOf}(\text{var1}) = \text{typeOf}(\text{var2})$, $\text{typeOf}(\text{var1}') = \text{typeOf}(\text{var2}')$, and $\text{op1} = \text{op2}$.
- $\text{var1} \sim \text{var2}$ iff $\text{typeOf}(\text{var1}) = \text{typeOf}(\text{var2})$.
- $\text{var1.field1} \sim \text{var2.field2}$ iff $\text{typeOf}(\text{var1}) = \text{typeOf}(\text{var2})$ and $\text{field1} = \text{field2}$.

Let $S = s_1; s_2; \dots; s_m$ and $R = r_1; r_2; \dots; r_n$ be any two code snippets. Note that these snippets contain comments added by SNIFF. We define the intersection of S and R , denoted by $S \sqcap R$, as the *longest common subsequence* (LCS) of S and R defined as follows: The LCS of S and R is the longest subsequence $Q = s_{i_1}; s_{i_2}; \dots; s_{i_l}$ of S such that there exists a subsequence $T = r_{j_1}; r_{j_2}; \dots; r_{j_l}$ of R , where $1 \leq i_1 < i_2 < \dots < i_l \leq m$; $1 \leq j_1 < j_2 < \dots < j_l \leq n$ and $s_{i_k} \sim r_{j_k}$ for all k . The LCS of two code snippets can be computed using a modification of a standard dynamic programming algorithm [4]. The complexity of the algorithm is $O(mn)$ where m and n are the sizes of the two code snippets respectively. All the code chunks (or snippets) ever handled by SNIFF are small in size (less than 10 lines of code) and hence the quadratic running time of the algorithm is not an excessive overhead.

Clustering Code Snippets: The code chunks obtained from the main database are grouped together based on their *similarity*. Informally, two code

chunks are *similar* if their intersection is still relevant to the query and does not lose too much information. We next formalize this notion of relevance. We first define *validity* as a measure of similarity of two snippets.

Let $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$ be a set of code snippets and let $q = w_1 w_2 \dots w_n$ be a user query. A code snippet S_i is said to be *q-valid* if its code and comments together contain all keywords w_j from the query q . For example, the code chunks returned from the main database in response to q are *q-valid* by definition. We will henceforth use *valid* in place of *q-valid*, when the query q is understood. Similarity of code snippets is defined below:

Definition 3. Two code snippets S_i and S_j are defined to be **similar** if their intersection $S_i \sqcap S_j$ is **valid**.

Informally, an initial code chunk matched to a query contains some lines of code which are relevant to the query, and others which are not. It should be noted that the relevant lines of code need not contain any of the keywords. Intersecting two code chunks or snippets attempts to remove the irrelevant lines of code, while keeping the ones which are relevant to the query (either through the presence of keywords, or because of issues concerning correct implementation). The intuition behind the intersection operation is that the relevant lines would co-occur in both the chunks while the irrelevant lines would not match. If the common lines found by the intersection operation still contain all the keywords, then the two code chunks or snippets are similar.

However, in some cases, there might be multiple method call sequences which correctly perform the task specified by the query. In that case, two such code chunks or snippets (corresponding to different sequences) would not contain all the keywords in their intersection, as the statements in each of them will be different. Thus, these two code chunks or snippets will be dissimilar.

We cluster the code snippets based on this similarity measure. The idea behind clustering is to group similar code snippets in one cluster. The intersection of all code snippets $\{S_{i_1}, S_{i_2}, \dots, S_{i_r}\}$ in a cluster represents the most relevant code to the query, that can be extracted out of those snippets. Formally, clustering is defined as a function $\mathbb{F} : \mathcal{S} \rightarrow \{1, 2, \dots, p\}$, where $\mathcal{S}$ is a set of code snippets and $p \leq |\mathcal{S}|$, satisfying the condition that if $\mathbb{F}(S_i) = \mathbb{F}(S_j)$, then S_i and S_j are *similar*. All snippets mapping to the same integer are said to be in the same cluster, and are represented by a single snippet in the final results. The clustering procedure is detailed in Algorithm 1. The running time of this procedure is quadratic in number of candidate code chunks. Our observation is that the number of code chunks returned from the database is very small (less than 30 on an average), and hence clustering runs very fast on them.

Ranking Code Snippets: After clustering, SNIFF ranks code snippets before returning them to the user. Ranking should reflect the relevance of the code snippet to the query. We observe that the most relevant code snippet to the query is generally implemented in a large number of client classes and methods, and hence is the most common way of performing the programming task required by the query. We formalize this property of code snippets by using *support* of the

Algorithm 1. Clustering and Ranking algorithm**Input:** $\mathcal{C} = C_1, C_2, \dots, C_N$, the list of code chunks returned for the query**Output:** I , the set of all clusters of code chunks in $\mathcal{C}$.

```

1:  $I \leftarrow \mathcal{C}$ 
2: for all  $C_i \in I$  do
3:    $support(C_i) = 1$ 
4: end for
5: for all  $C_i \in I$  do
6:   for all  $C_j \in I, C_j \neq C_i$  do
7:      $C \leftarrow C_i \sqcap C_j$ 
8:     if  $isValid(C)$  then
9:        $support(C) = support(C_i) + support(C_j)$ 
10:       $I \leftarrow I \cup \{C\} \setminus \{C_i, C_j\}$ 
11:     end if
12:   end for
13: end for
14: return  $sort(I, support)$ 

```

snippet defined as follows: For every code chunk C_i returned from the database, $support(C_i) = 1$. The support of an intersection is defined as the sum of the supports of code snippets that participate in the intersection operation. Thus $support(S_1 \sqcap S_2 \sqcap \dots \sqcap S_k) = \sum_{j=1}^k support(S_j)$. That is, support represents the number of occurrences of the snippet across client source files. Lines 3 and 9 of the Algorithm 1 initialize and update the supports of the clusters respectively. We rank the clusters based on their supports, with the clusters having higher supports receiving higher ranks.

4 Evaluation

We conducted three different experiments using SNIFF to show that SNIFF is effective in solving programmers' queries and to compare it against the existing tools and search engines. In our first experiment, we compared SNIFF against tools such as Prospector [12] and Google Code Search(GCS) [5]. We performed controlled user experiments where a set of programming problems were given to a group of users and their performance (i.e. the time they spent to complete the tasks) using different tools was observed. In the second experiment, we compared SNIFF against online code search engines like GCS, Koders and Krugle. We manually collected programming problems posted on a Java user forum [8] and converted them into natural language queries. We then posed these queries to SNIFF as well as the online search engines and compared the results. We performed a third experiment to show the effectiveness of our intersection and ranking techniques.

4.1 User Study

Our user study was aimed at evaluating the usefulness of SNIFF to developers for real programming tasks which involved reuse of existing APIs. We designed four programming problems and assigned them to a set of users. We had eight participants in our study. Each user was allowed to use SNIFF for two of the

four problems. Of the remaining two, they were allowed to use Prospector for one problem and Google Code Search Engine for the other. We assigned the problems and the tools to be used to solve them randomly to each user. We recorded users' final answers, the time they spent to complete each problem, the queries they issued, and the rank of the snippet that they used in their code.

Each user was given a brief introduction to SNIFF and Prospector. In the introduction we described the tools using a short demo. There was no training phase and none of the users participating in the user study had used SNIFF before. The users were graduate students who had moderate to expert programming skills in Java.

The programming problems were designed to approximate real programming tasks. The users were not given any hints about when to use SNIFF (or any of the other tools/search engines). However, the tasks were designed in such a way that they involved the usage of some APIs which were not very commonly used. The motivation for this scheme was to ensure that the users would be required to search for some APIs that they did not know about. The four programming problems for the user study were as follows:

1. **Problem 1:** In Eclipse, the visual representation of the editor, i.e. the actual window we see on the editor, is called the *active editor*. The task is to retrieve this active editor from the workbench.
2. **Problem 2:** JDOM is a parser for parsing java source files and identifying method declarations/invocations etc. The task is to use this parser to parse a java source and create and AST.
3. **Problem 3:** JDBC is the Java technology to connect to a remote database and run SQL queries on it. The user is given the url of a remote database server. The task is to connect to the database using JDBC.
4. **Problem 4:** I have a generic Viewer object in Eclipse and I want to pop-up a dialog displaying all directories in the workspace. The task is to open such a directory dialog.

The results of the experiments are given in Figure 1. The plot in the figure gives the average time taken by the users on each problem using different tools. The plot shows that on all problems, SNIFF took about 40% less time as compared to Prospector and Google Code Search. In problem 1, the users had difficulty in deducing the desired return type, `IEditorPart`; therefore, they ended up browsing the eclipse API while using Prospector. We observed similar trend in problem 3, where the desired destination class was `java.sql.Connection`. The JDOM API in problem 2 is fairly complicated and Google Code Search failed to return the specific code snippet for parsing the java file. In fact, on all queries, the results returned by Google Code Search required significant manual inspection (most of the time in the associated source code) to arrive at the desired code snippets. SNIFF returned the exact code snippet, although the users needed to play around a little with their queries to arrive at this code snippet. Also, the snippets returned by SNIFF required the least amount of post-processing at the users' end. Most of this post-processing was renaming variables and importing required packages, which was pretty straightforward.

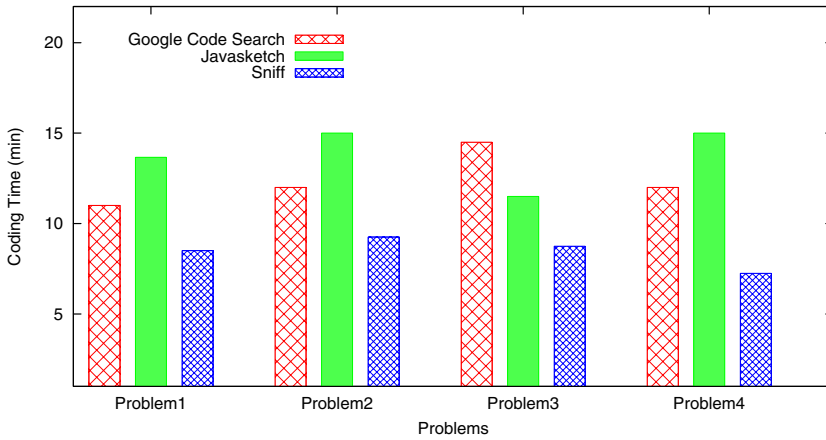


Fig. 1. Average time taken by the users on the problems using each tool

4.2 Comparison with Online Search Engines

We collected random queries from the ones issued by the users in the previous section and issued them to SNIFF as well as online search engines like Google Code Search [5], Koders [10], and Krugle [3]. We also collected some of the most common programming problems from frequently asked questions on a Java user forum [8] and formulated them as natural language queries. On all of these queries, we compared the results of SNIFF and online search engines with the responses posted on the forum. Table 6 shows the rank of the most relevant snippet (as judged by a human programmer based on the forum response) using different tools. All existing search engines display a set of small code snippets in response to the query. These snippets are hyperlinked to the entire source files that contain them, and clicking on the snippets displays the contents of the files with keywords highlighted in different colors. We have manually searched these returned results and report a match if the returned source file contains the desired code snippet. A - in the table means that the desired code snippet was not present in top 10 hits from the corresponding tool. The last three lines for each tool give the percentage of queries where the most relevant code snippet is among top 1, 5 and 10 hits respectively. Our experiments show that SNIFF returned the most relevant snippet as the top ranked snippet for 87.5% of queries. The same numbers for GCS, Koders and Krugle are 25%, 62.5% and 12.5% respectively. On all the queries, SNIFF returned the most relevant code snippet in top 5 results.

4.3 Effectiveness of Intersection and Clustering Techniques in Sniff

In order to show the effectiveness of our intersection and clustering algorithms, we first issued the queries given in Table 6 to SNIFF with intersection and clustering turned on. We then issued the same queries after disabling the intersection and

Table 6. Results from existing code search engines on user queries

Query	Rank of the top snippet that matched the forum response			
	GCS	Koders	Krugle	SNIFF
get active editor window from eclipse workbench	2	2	2	1
parse a java source and create ast	2	3	2	1
connect to a database using jdbc	2	1	–	1
display directory dialog from viewer in eclipse	–	9	2	1
read a line of text from a file	6	1	–	1
return an audio clip from url	1	1	1	1
execute SQL query	1	1	3	2
return current selection from eclipse workbench	5	1	4	1
Relevant snippet is top ranked (%)	25	62.5	12.5	87.5
Relevant snippet is in top 5 hits (%)	75	87.5	75	100
Relevant snippet is in top 10 hits (%)	87.5	100	75	100

clustering. During this phase, we ranked the snippets simply based on their sizes with the smaller snippets getting higher ranks. We compared the ranks and sizes (in LOC) of the most relevant snippets with and without intersection/clustering. We also compared the amount of pruning obtained using intersection.

The results are shown in Table 7. The first column of the table gives the queries. The next two columns give the rank of the most relevant code snippet with and without intersection and clustering (denoted **I/C** and **No I/C**, respectively.) The next two columns give similar observations for the size of the most relevant snippet. The table shows that the rank and size of the returned snippet is better when intersection and clustering are turned on. In more than half of the cases, the most relevant snippets are poorly ranked when intersection is turned off. Intersection also reduces the size of resultant snippets by 34% on an average. Smaller size of snippet usually means less post-processing time on the returned snippets required by the users.

5 Other Related Work

A large fraction of the previous research, including Prospector [12] has focused on user queries of the form $T_{in} \rightarrow T_{out}$, where T_{in} is a source object and T_{out} is a destination object. These tools return code snippets that convert T_{in} to T_{out} , using a sequence of API method calls. PARSEWeb [18] and XSnippet [16] are two more examples in this research direction.

PARSEWeb [18] gathers the relevant code samples from GCS and performs a static analysis over them to answer the queries of type $T_{in} \rightarrow T_{out}$. They also split the query by introducing intermediate object types. The dynamic database (that of Google Code Search) together with the query splitting results in some reported improvements. XSnippet [16] makes use of context information along with a user query for finding relevant snippets. Their object instantiation queries

Table 7. Effect of Intersection and Clustering on Results

Query	Rank		Size(in LOC)	
	I/C	No I/C	I/C	No I/C
get active editor window from eclipse workbench	1	3	3	4
parse a java source and create ast	1	1	2	4
connect to a database using jdbc	1	6	3	6
display directory dialog from viewer in eclipse	1	1	4	7
read a line of text from a file	1	3	3	5
return an audio clip from url	1	2	6	6
execute SQL query	2	2	2	5
return current selection from eclipse workbench	1	2	2	4

can be classified as *type-based* (from type T_{in} to type T_{out}) or *parent-based*, where parenthood is defined by the subclass-superclass relation.

Although there is variation in the expressiveness of queries allowed by these approaches, the basic structure of the queries is still limited to object instantiation and cannot be generalized to free-form natural language queries. SNIFF does not suffer from this limitation since it allows free-form queries.

An alternate approach is the work of Murphy et al [6,7], which locates example code files relevant to the code under development. The work focuses on structural context matching heuristics. However, this approach does not allow the user to explicitly specify a query and hence, proves to be useful only in special cases, when the code under development is very similar to some example in the repository. Often, the developer might not have enough context present in her program to be guided to an actually relevant example.

SPARS-J [13] is a closely related Java class retrieval system that applies a graph-based model to the programs. It returns a ranked set of classes to a free-form user query using a frequency-of-usage based heuristic. However, the SPARS-J technique is not aware of the functionality of a method or API beyond what is suggested by its name. Several user queries in our experiments were based on functionality. SNIFF gathers much more information about the source code from its inline comments or JavaDoc specifications. Moreover, SPARS-J returns relevant classes and it often requires time and expertise to identify the precise snippet inside a complicated class/API. Robillard et al [15] present another graph-based approach to discover the code relevant to change during code modification. They use topological structure of the program graph to propose and rank program elements that are potentially interesting to a developer modifying the source code.

The role of comments as an aid to program understanding has been a well-studied topic [19]. iComment [17] automatically analyzes comments written in natural language to extract implicit program rules and uses them to automatically detect inconsistencies between comments and source code, indicating either bugs or bad comments. However, extracting and using the information contained

in comments can be hard, since comments often convey other information like directions to colleagues (especially in a group development environment) [20].

Besides comments, program invariants is another interesting direction for identifying code snippets. There has also been a lot of work on inferring specifications [1,11]. However, we believe that in the context of code reuse and searching for relevant APIs and their usage patterns, comments are much more useful than formal specification of invariants.

Our intersection approach has some similarities with DECKARD [9]. However, SNIFF performs type-based intersection only at the statement level (since we treat a program as an ordered list of statements), while DECKARD focuses on efficient algorithms for identifying similar subtrees and applying it to tree representations of the source code.

6 Conclusion

We have shown that our approach to locate relevant snippets for a free-form query has a lot of promise. Inserting API comments in the client code at the right places helps in localizing search results. Our current results for free-form queries are already better than several existing code search engines. The intersection performed by SNIFF is also critical in coming up with the relevant code snippet(s), while separating out the statements that are irrelevant to the query.

Integrating and indexing search results from online search engines is an important future work on this problem. An interesting extension of current approach would be to deploy the tool in a cloud or compute cluster. Releasing the tool for a larger class of programmers will provide us more valuable feedback on usefulness of techniques used in this paper.

Acknowledgments

We would like to thank Jacob Burnim, Pallavi Joshi, Chang-Seo Park, Christos Stergiou, Raluca Sauciu, Yamini Kannan, Nicholas Jalbert and Subhransu Maji for their valuable comments on a previous draft of this paper and for participating in the user study. This work is supported in part by the NSF Grants CNS-0720906, CCF-0747390, and CCR-0326577.

References

1. Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. In: POPL 2002, pp. 4–16 (2002)
2. Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30(1–7), 107–117 (1998)
3. Krugle inc, <http://www.krugle.com>
4. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to algorithms. MIT press/ McGraw-Hill (2001)
5. Google code search, <http://google.com/codesearch>

6. Holmes, R., Murphy, G.: Using structural context to recommend source code examples. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 117–125. Springer, Heidelberg (2006)
7. Holmes, R., Walker, R., Murphy, G.: Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering* 32(12), 952–970 (2006)
8. Java frequently asked questions, <http://www.javafaq.com/>
9. Jiang, L., Misserghy, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: ICSE 2007, pp. 96–105 (2007)
10. Koders inc, <http://www.koders.com>
11. Kremenek, T., Twohey, P., Back, G., Ng, A., Engler, D.: From uncertainty to belief: inferring the specification within. In: OSDI 2006, pp. 161–176 (2006)
12. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the api jungle. In: PLDI 2005, pp. 48–61 (2005)
13. Matsushita, M., Inoue, K., Yokomori, R., Yamamoto, T., Kusumoto, S.: Ranking significance of software components based on use relations. *IEEE Trans. Softw. Eng.* 31(3), 213–225 (2005)
14. Porter, M.F.: An algorithm for suffix stripping. In: *Readings in information retrieval*, vol. 14, pp. 130–137 (1980)
15. Robillard, M.P.: Automatic generation of suggestions for program investigation. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 11–20. ACM, New York (2005)
16. Sahavechaphan, N., Claypool, K.: Xsnippet: mining for sample code. In: OOPSLA 2006, pp. 413 – 430 (2006)
17. Tan, L., Yuan, D., Krishna, G., Zhou, Y.: `/*icoment: bugs or bad comments?*/`. In: SOSP 2007, pp. 145–158 (2007)
18. Thummalapenta, S., Xie, T.: PARSEWeb: A programmer assistant for reusing open source code on the web. In: ASE 2007, pp. 204–213 (2007)
19. Woodfield, S., Dunsmore, H., Shen, V.Y.: The effect of modularization and comments on program comprehension. In: ICSE 2002, pp. 215–223 (1981)
20. Ying, A.T.T., Wright, J.L., Abrams, S.: Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In: MSR 2005, pp. 1–5 (2005)

Inquiry and Introspection for Non-deterministic Queries in Mobile Networks

Vasanth Rajamani¹, Christine Julien¹, Jamie Payton²,
and Gruia-Catalin Roman³

¹ Department of Electrical and Computer Engineering
The University of Texas at Austin
`{c.julien,vasanthrajamani}@mail.utexas.edu`

² Department of Computer Science
University of North Carolina, Charlotte
`payton@uncc.edu`

³ Department of Computer Science and Engineering
Washington University in Saint Louis
`roman@wustl.edu`

Abstract. This paper focuses on the information gathering support needs for enterprises that operate over wireless mobile ad hoc networks. While queries are a convenient way to obtain information, the highly dynamic nature of such networks makes it difficult to ensure a precise match between the results returned by a query and the actual state of the enterprise. However, decisions can be made based on the perceived quality of the information retrieved; specialized query support is needed to control and assess the accuracy of the query results. In this paper, we introduce the notion of inquiry mode to allow the user to exercise control over the query processing policy so as to match the level of accuracy to the requirements of the task. In addition, we describe the use of query introspection, a process for assessing the fitness of a particular inquiry mode. Both concepts are formalized, illustrated, and evaluated.

1 Introduction

Information drives the modern world. Everyday decisions depend on the quality of data decision makers have. With the introduction of mobile wireless devices, access to information has been extended to any individual carrying a phone or PDA. This, in turn, led to changes in the very nature of the enterprise structure by empowering mobile users and by facilitating more decentralized decision processes, faster reaction times, and more nimble data acquisition. A still more radical transformation is made possible by the emergence of mobile ad hoc networks (MANETs) that support application domains where a still more fluid organization is required to adapt to rapidly evolving circumstances. Emergency response units, wilderness exploration groups, and battlefield intelligence teams demand quality information gathered from distributed, cooperating sources.

The transformation will have far reaching implications, as enterprises that rely on connectivity to the wired infrastructure are likely to evolve to include

operations over local MANETs with only occasional access to the wired network. Construction sites are a representative example. Today, phones are used on the site to gather information about the status of jobs and to track developing situations. A chemical spill is likely to trigger a large volume of phone conversations to coordinate a response to the emergency and to assess the appropriateness of those actions. Enterprise level systems on the wired network, though well-suited to support logistics and planning, are not nimble enough to support these ad hoc peer-to-peer interactions that emerge unexpectedly. A more effective solution is required to acquire up-to-date data on-demand.

These new enterprises demand flexible and timely access to spatially correlated information about highly dynamic environments. Because the information is distributed, accessing it entails the evaluation of a distributed query over a rapidly evolving network, rendering any atomicity guarantees infeasible much of the time. The spatiotemporal dimension of the information encourages decision makers to pose questions in a manner sensitive to one's mental model of how space is organized and how the system evolves. Furthermore, one is likely to question the soundness of the decision process. Am I asking the right question? Am I looking in the right place? Am I getting an accurate enough answer? By knowing how a query is processed and how volatile the system state is, one can gain important semantic information about the data a query returns. This, combined with an ability to specify how queries are evaluated, can significantly impact the quality of the decision process. For example, a construction site supervisor can specify that a query will be evaluated by sampling across the entire site or by acquiring values from all devices within a smaller local area. Even though the queries may return identical results, a supervisor will interpret the data differently. Finally, changes in query results over time (e.g., rising chemical concentration readings) or the presence of unexpected values may offer insight into the adequacy of the query relative to the decision maker's goals.

This paper explores the semantic dimension of query processing over MANETs. Starting with the premise that the universe of discourse is the global state of a connected mobile ad hoc network that represents a distributed mobile enterprise, we seek to provide decision makers with a new set of query processing tools. These are meant to enable both flexible control over the spatiotemporal dimension of query processing and the ability to assess the fitness of the query processing for the specific task at hand. Our contribution is twofold:

- We propose an inquiry mode as the means to specify how a query is evaluated across a MANET; we identify several inquiry modes that relate to different semantic interpretations of results; and we offer both a general formalization of the concept and specific query processing protocols.
- Complementing the notion of inquiry mode is query introspection, which enables evaluation of the adequacy of an inquiry mode. We propose the use of adequacy metrics that compare query results against expectations or results obtained from previous correlated queries.

Our approach represents an important shift in the way one thinks about query processing. A query is intellectually decoupled from the traditional notions of

database processing and is promoted as a basic tool for exploring the surrounding world. Within this broad context, new user requirements for query processing emerge, which suggest that queries should be sensitive to the spatiotemporal nature of the environment and its evolutionary dynamics. In the remainder of this paper, we first define, formalize and demonstrate inquiry modes. Section 4 then defines and demonstrates query introspection. Section 5 presents our programming model and a case study application that employs it. We discuss related work in Section 6 and conclude in Section 7.

2 Defining Inquiry in Dynamic Networks

Our approach is based loosely on our previous work modeling change in mobile environments [1]. In our model, queries can retrieve information from the network using a variety of techniques, or *inquiry modes*, each of which entails its own costs and benefits. Informally, an inquiry mode specifies the subset of hosts that contribute to resolving the query. Different inquiry modes may generate vastly different results for the same query. In this section, we formalize the specification of queries, their inquiry modes, and their results and provide concrete examples of how real protocols can be expressed using this formalization.

3 Hosts, Configurations, and Configuration Change

A mobile ad hoc network is a closed system of hosts, each represented as a triple (ι, ζ, ν) , where ι is the host's unique identifier, ζ is its context, and ν is its data value. In a simple model, the context can be simply a host's location. In more complicated models, the context may include a list of a host's neighbors, routing tables, and other system or network information. A snapshot of the global abstract state of a mobile ad hoc network, which we call a *configuration*, C , is simply the set of these host tuples, one for every host in the network.

To capture network connectivity, we define a binary logical connectivity relation, $\mathcal{K}$, to express the ability of one host to communicate with a neighboring host. Using the values of the fields of a host triple, we can derive physical and logical connectivity relations. As one example, if the host's context element, ζ , includes the host's location, we can define a physical connectivity relation based on communication range. $\mathcal{K}$ is not necessarily a symmetric relation; in the cases that it is symmetric, $\mathcal{K}$ specifies bi-directional communication links.

The environment evolves as the network topology changes, value assignments occur, and hosts exchange messages. We model network evolution as a state transition system where the state space is the set of possible system configurations, and transitions are *configuration changes*. Specifically, a single configuration change consists of one of the following:

- *neighbor change*: changes in hosts' states impact the connectivity relation.
- *value change*: a single host can change its stored data value.
- *message exchange*: a host can send a message that is received by one or more neighboring nodes.

To refer to the connectivity relation for a particular configuration in this evolution, we assign configurations subscripts (e.g., C_0 , C_1 , etc.) and use $\mathcal{K}_i$ to refer to the connectivity relation for configuration i .

We build on $\mathcal{K}$ to define *reachability* across configurations. The reachability relation, $\mathcal{R}_{(i,j)}$, is a binary relation on host tuples that indicates the potential of one-way multi-hop communication between them that starts no earlier than the i^{th} configuration and completes no later than the j^{th} configuration:

$$\begin{aligned} &\langle \forall k : i \leq k \leq j :: h \in C_k \Rightarrow (h, h) \in \mathcal{R}_{(i,j)} \rangle \\ &\langle \forall h_1, h_2, k : i \leq k \leq j :: (h_1, h_2) \in \mathcal{K}_k \Rightarrow (h_1, h_2) \in \mathcal{R}_{(i,j)} \rangle \\ &\langle \forall h_1, h_2, h_3, k : i \leq k < j :: ((h_1, h_2) \in \mathcal{R}_{(i,k)} \wedge (h_2, h_3) \in \mathcal{K}_{k+1}) \Rightarrow (h_1, h_3) \in \mathcal{R}_{(i,j)} \rangle^1 \end{aligned}$$

First, every host is always reachable from itself. Second, if one host (h_1) is connected to another (h_2) in any configuration between i and j inclusive, then h_2 is reachable from h_1 . Finally, we recursively define reachability.

3.1 Queries, Inquiry Modes, and Query Results

We extend our definition of reachability to define *query reachability*, which, informally, determines whether it was possible to deliver a query to and receive a response from some host h within the sequence of configurations. Given the host who issues the query, $\bar{h}$, query reachability for query q , $\mathcal{R}^q$, is defined as:

$$(\bar{h}, h, i) \in \mathcal{R}^q \Leftrightarrow (\bar{h}, h) \in \mathcal{R}_{(0,i)} \wedge (h, \bar{h}) \in \mathcal{R}_{(i,m)}$$

It is not only necessary that h was reachable from the query issuer during the query, but also that, after h was able to receive the query, $\bar{h}$ was reachable from h , ensuring that it was possible for h 's response to reach the query issuer.

In our model, the inquiry mode is the technique used to process the query over a set of configurations. An inquiry mode is defined by a *forward-function* and a *respond-function*, both of which use a host's state to make a decision about whether to propagate and/or respond to a query. From any host's perspective, an inquiry model is simply the application of two independent functions: $I \triangleq \langle f, r \rangle$. Each of f and r is a boolean function over a host tuple h . In the same atomic step in which a host receives a query, it evaluates both f and r given the particular query and the host's state. Since a message exchange constitutes a configuration change that takes the network from some configuration C_i to C_{i+1} , the two functions are evaluated on the receiving host's tuple in configuration C_{i+1} .

Next, we define a trio of relations over host tuples that allow us to formalize a query's result. Briefly, the *forwards* relation, Φ_q , specifies pairs of senders and receivers of the query and the configuration of reception; the *receptions* relation,

¹ In the three-part notation: $\langle \text{op } \textit{quantified\_variables} : \textit{range} :: \textit{expression} \rangle$, the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a multiset of values to which *op* is applied. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for *op*, e.g., *true* if *op* is $\forall$.

Γ_q , specifies hosts that received the query and the relevant configuration; and the *responses* relation, Ψ_q , specifies whether a host qualified to generate a response.

The forwards relation is a ternary relation over host tuples and configuration numbers. Specifically, $(h_1, h_2, i) \in \Phi_q$ if the host identified by h_1 forwarded the query q in a configuration in which h_1 captured its state, and the host identified by h_2 received the query in a configuration in which h_2 captured its state. In addition, the host that forwarded the query (h_1) must also have satisfied the query's *forward-function*, $q.f$ in the state that it received the query. Formally:

$$(h_1, h_2, i) \in \Phi_q \Rightarrow \langle \exists j, h'_1 : i < j \wedge h'_1.\iota = h_1.\iota :: (h'_1, i) \in \Gamma_q \wedge q.f(h'_1) \wedge (h_1, h_2) \in \mathcal{K}_j \rangle$$

In this formalization, i and j identify configurations; i is the configuration in which the host h'_1 received the query, and j is the configuration in which the forwarding occurred. Forwarding the query caused the transition $C_j \rightarrow C_{j+1}$.

We next define the receptions relation, Γ_q . A host is in the receptions relation if it was the target of a forward or if it is the query issuer:

$$(h, i) \in \Gamma_q \Rightarrow (h = \bar{h} \wedge i = 0) \vee \langle \exists h' :: (h', h, i - 1) \in \Phi_q \rangle$$

This requires that for a host other than the query issuer to receive a query in C_i , the query must have been forwarded by a neighboring host in C_{i-1} .

Finally, the responses relation, Ψ_q , defines the hosts that received the query and generated a response to it. To generate a response, the host must receive the query and satisfy the *respond-function*. Formally, Ψ_q is:

$$h \in \Psi_q \Rightarrow \langle \exists i :: (h, i) \in \Gamma_q \wedge q.r(h) \rangle$$

A query's result, ρ , is a subset of a configuration: it is a collection of host tuples that constitute responses to the query; no host in the network is represented more than once in ρ , though it is possible that a host is not represented at all (e.g., because it was never reachable from the query issuer). The constraints defining a query's result stem from the concepts derived above. First, a result present in the query must come from a host that responded to the query:

$$h \in \rho \Rightarrow h \in \Psi_q \tag{1}$$

Second, any result must have satisfied the aforementioned property of query reachability. This ensures that both forward and reverse paths exist for query propagation and response collection. Formally, given the query issuer, $\bar{h}$:

$$h \in \rho \Rightarrow \langle \exists i, j : i \leq j :: (h, i) \in \Gamma_q \wedge (\bar{h}, h, j) \in \mathcal{R}^q \rangle \tag{2}$$

3.2 Examples of Application Protocols

The ability to specify the inquiry mode with which a query executes gives an application fine-grained control. Consider an application that has some requirement for query quality. Given the availability of forward and respond function

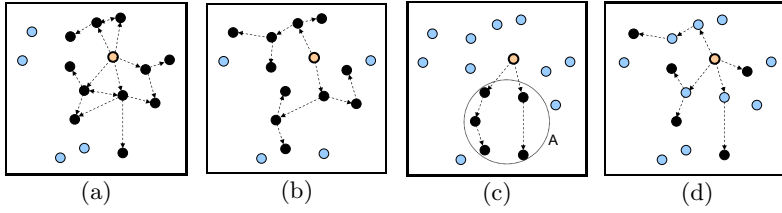


Fig. 1. Potential inquiry modes. Solid lines indicate available connections, dashed lines indicate sent messages. (a) Flooding. Every node within the constraint (2 hops) retransmits the query. (b) Probabilistic. Every node within the constraint (3 hops) that receives the packet retransmits it to 2 random neighbors. (c) Location Based. The query reaches the nodes in region A. (d) Random. The query reaches any 5 nodes.

definitions, the application can construct a variety of possible query processing protocols, honing in on the implementation best suited to the combination of the application’s requirements and the query environment. In this section we show how query processing protocols that are commonly used in mobile ad hoc networks can be easily represented using inquiry modes and their components.

Flooding Inquiry Mode: Flooding based queries are most common in mobile applications [2,3]. The sending node broadcasts the query to all of its one-hop neighbors, who in turn propagate the message to their neighbors. A query is very likely to reach every node in the network. However, this approach can be very expensive in terms of the message overhead [4]. Approaches also exist that constrain the flooding to some region of the network [5]. Because a basic flooding protocol is deterministic, it cannot be parameterized, so no protocol parameters are included in the inquiry mode. However, a constrained flood can be parameterized by specifying the number of hops across which the query is flooded. Fig. 1(b) shows the messages sent by and the nodes responding to a flooding inquiry constrained to nodes within two hops of the query issuer.

Expressing the flooding inquiry mode using our formulation is trivial:

$$\mathcal{I}_{\text{flooding}} = \langle \text{true}, \text{true} \rangle$$

This query reaches all hosts but at a significant communication overhead.

Probabilistic Inquiry Mode: Probabilistic techniques distribute the query with a lower message overhead by reducing the number of nodes involved in query propagation. Fig. 1(c) depicts an example, where each node receiving a query retransmits it to two randomly selected neighbors. A more common variant is to use a probability function to determine whether a particular node should rebroadcast a received message [4]. Additional parameters can be used to determine how many times to retransmit a message [6].

Here, the respond function is identical to flooding’s respond function. The forward function ensures that only a fraction of messages propagate. Every host generates a random number (*rand*) and passes it as an argument to the $f_{\text{probabilistic}}$ function, which is used to determine satisfiability:

$$f_{\text{probabilistic}}(\theta) \triangleq (\theta < p)$$

$$\mathcal{I}_{\text{probabilistic}} = \langle f_{\text{probabilistic}}(\text{rand}), \text{true} \rangle$$

In this inquiry mode, the query reaches only a probabilistically selected set of hosts, which comes with added complexity but reduced overhead.

Location based Inquiry Mode: If location information is available, it can direct queries to particular regions. Fig. 1(d) demonstrates a location based query targeting region A. In this inquiry mode, it is important to be able to compute a logical function that determines whether a given node satisfies the query's location requirements. This is accomplished by writing forward and respond functions that use the cartesian distance to evaluate satisfiability:

$$r_{\text{location}}(x_1, y_1) \triangleq ((x_1 - x_2)^2 + (y_1 - y_2)^2 < \text{maxD})$$

$$\mathcal{I}_{\text{location}} = \langle \text{true}, r_{\text{location}}(h.\zeta.\lambda.x, h.\zeta.\lambda.y) \rangle$$

The parameters come from the host's location (λ) stored in its context ($h.\zeta$). The query targets hosts in a specific location, reducing the communication overhead but requiring significantly increased computation and resource demands.

Random Inquiry Mode: At the far end of the spectrum, a random sampling algorithm may just randomly select a fraction of hosts to reply to the query, as depicted in Fig. 1(e). These protocols can be parameterized by specifying how many or what fraction of nodes should take part in the query.

The forward function is the same as in flooding. However, the respond function needs to ensure that only $k\%$ of the query receptions are replied to:

$$r_{\text{random}}(\theta) \triangleq (\theta < k)$$

$$\mathcal{I}_{\text{random}} = \langle \text{true}, r_{\text{random}}(\text{rand}) \rangle$$

The value of k is used as decision criteria and a randomly generated number, rand , is passed in as a parameter. The random inquiry mode reaches only randomly selected hosts, radically reducing communication complexity. A common technique found in practical protocols is the use of an aggregation strategy. It should be noted that our model only forces decisions to be made immediately; an implementation of the model can accommodate aggregation by delaying the response with proper use of timers to create a routing structure for aggregation.

4 Inquiry Introspection

Using different inquiry modes can yield different results for the same query. The suitability of an inquiry mode is determined by the needs of the querying application and may depend on the dynamics of the environment. We define *query introspection* as the use of information about a query's result to determine if the associated inquiry mode meets the application's needs. In this section, we discuss the use of adequacy metrics, which compare query results against application expectations, to support query introspection.

4.1 An Informal Introduction to Query Introspection

Different inquiry modes provide different sets of tradeoffs as they collect information from a distributed network. The inquiry mode selected depends on an application's needs; how well a particular inquiry mode meets the needs of the application is dependent, in part, on the environment during query execution. For example, a query issued by a construction supervisor may determine the concentration of a dangerous compound on a construction site. If it is important to minimize the query's overhead, the application may use a random inquiry mode to collect concentration readings from randomly selected hosts across the site. However, the random inquiry mode may not provide results that are accurate enough. For a dense network the random inquiry mode may provide a representative result while the same query may not provide an accurate enough view of a sparse network to enable a decision about site safety.

This kind of analysis can be supported through query introspection, using feedback about query execution to determine if an inquiry mode is appropriate. The ideal approach to evaluating an inquiry mode's tradeoffs is to determine how well results reflect the ground truth of the environment during the query's execution, but this is impractical. Instead, query introspection examines query results directly, using a history that may include results for this query and for queries recently executed using the same inquiry mode. Since each host's contribution to a query result contains information about its context, we can consider properties of the execution environment relevant to the application's needs.

Using a history of query results, we can approximate a view of the world that can be used to determine if an inquiry mode is appropriate in the current environment given the application's needs. Query introspection, then, can be achieved by applying an adequacy metric over similar queries' results. We now formalize query introspection and provide examples of adequacy metrics that can be used in the introspection process.

4.2 Formalizing Query Introspection

Query introspection is the process of determining if an inquiry mode is suitable. This decision is often related to a tradeoff between the desired properties of the result and the cost of query execution. A desired property may be that the results are representative samples; another is that the results are relatively stable and do not change rapidly over time. Variability in the network topology, the query's execution context, and randomness introduced by the inquiry mode can impact how well results delivered by a query reflect the desired properties.

To support quantifiable introspection, we apply adequacy metrics to query results. An adequacy metric, d , measures the logical distance between the desired property of a query's execution and the actual properties of the achieved query result. For each distance function, an associated threshold (δ) can be defined by the application to aid in evaluation. This simple construction supports expression of arbitrary adequacy metrics that can enrich decision-making processes.

We identify two categories of adequacy metrics. The first measures the quality of the data captured by a query based on the variability between successive

queries. The second compares an idealized property of the execution environment to the environment's measured state during query execution. This set of metrics is not exhaustive; rather, we intend to illustrate applications' needs and to provide a framework for identifying and expressing adequacy metrics.

Data Capture Quality Metrics. Often, decisions regarding the appropriateness of an inquiry strategy are related to the desired quality of the result. One way to measure the quality of query results is to define an adequacy metric based on a measurement of the changes in the captured data due to network variability during query execution. This type of adequacy metric can be supported by constructing a baseline for comparison using the results of a previously executed query. In general, this kind of distance metric can be expressed as: $d(\mathcal{P}_j(\rho_j), \mathcal{P}_k(\rho_k))$, where ρ_j is the result for the j_{th} query issued in a sequence of queries that employ the same inquiry mode, $\mathcal{P}_j$ maps a desirable property of the result to a numerical value, and $j \leq k$.

Set Difference. Consider a construction site supervisor that uses a probabilistic inquiry mode to return an inventory of available materials; however, the supervisor realizes that when the network is highly dynamic, this type of inquiry mode is not sufficient to achieve a high-quality result that presents an accurate view of the inventory. To support this kind of introspection, the quality of results can be assessed by measuring variability between different collections of results. The set difference metric quantifies the percentage of items returned by a query that are newly available, have been modified, or are no longer reachable in comparison to previously collected results. We can express a distance metric that determines the fraction of new results as:

$$d = \frac{|\rho_k - \rho_j|}{|\rho_k|}$$

where the k^{th} query is the most recently issued in a sequence of queries using the same inquiry mode, and $j < k$. The threshold δ associated with this metric depends on application needs.

The set difference operator can similarly be used to describe how many result elements have departed between the submission of a previous query and the current query. In our construction site scenario, the amount of bricks available on-site is likely to remain steady until a job consuming them begins, so the construction site supervisor may initially use a random sampling inquiry mode to check the inventory of bricks. The supervisor can use the departure distance function to determine when a job on the site that consumes bricks has begun, and can alter the inquiry mode if necessary to more closely track brick use.

We can also define an adequacy metric based on a cumulative view across an entire sequence of returned results. For example, we can define an adequacy metric based on transitive departures, which counts the fraction of hosts that departed between the beginning of the execution of query i and the conclusion of query k , even if the hosts later return. This can be captured as:

$$d = \frac{|\left(\bigcup_{i=1}^k \rho_i\right) - \rho_0|}{|\rho_k|}$$

Transitive additions can be specified similarly. These transitive distance metrics allow a construction site supervisor, for example, to make a decision regarding the use of the inquiry mode to more closely monitor inventory after an influx of supplies due to a delivery or the departure of supplies to another site.

Aggregate distances. Another way to describe the quality of a query's result is to measure the distance between the aggregated numerical values of a previous result and the aggregated values for the current result. Such aggregate distance measures can define adequacy metrics based on trends in the reported results. For example, a construction supervisor may initially use a probabilistic inquiry mode to monitor the total level of hazardous chemicals. If the hazardous chemical level rises at a rate that implies a leak, the probabilistic inquiry mode may no longer be suitable; given the potential danger, an inquiry mode which gives more accurate view of the world is needed regardless of the associated cost. To make this kind of assessment, an aggregate distance metric can compute the distance of an aggregate (e.g., a total) between two queries:

$$d = |\langle \text{sum } p : p \in \rho_{i+1} :: p \rangle - \langle \text{sum } p' : p' \in \rho_i :: p' \rangle|$$

Variations on this metric can be applied to other aggregates such as count, minimum, maximum, and average.

Environmental Property Metrics. The previous class of adequacy metrics are concerned with the quality of a query's result. These metrics defined the distance between the most recent query result and an approximate view of the ground truth. Here, we describe adequacy metrics that evaluate the distance between a query result and a desired property of the execution environment during query processing. These metrics can be defined in terms of the distance between an ideal environment and the conditions under which the query executed, as captured by the context field ζ for each host tuple in the result.

Network Coverage. In some cases, a query's network coverage is important in determining the suitability of its inquiry mode. A query covers an area if the elements of the result form a connected graph that spans the associated geographic region. In regions of the network where the data is dense, an area can be covered by an inquiry strategy that selects a small subset of hosts. This would yield a query result that is representative of the data in the region while reducing the associated overhead. In sparse regions of the network, a query result may cover a region only if all hosts in the region report results. The query issuer can use query introspection to determine if results obtained using a particular inquiry mode provide a reasonable representation of the surrounding area.

While finding the minimum set of nodes that cover a geographic region is an NP-hard problem, it is still possible to provide an adequacy metric based on whether

the results approximately satisfy a network coverage constraint using information about the density of results. We can get a rough estimate of this density through average numbers of neighboring results; a relatively high average results suggests that the results are dense, while a low average suggests that results are sparse. To provide an example based on location, we first define a connectivity relation over host locations that defines the existence of communication links between hosts. A physical connectivity relation that represents a connectivity model with a circular, uniform communication range can be defined using the location variable λ from a host's context ζ :

$$(h_{i_0}, h_{i_1}) \in \mathcal{K} \Leftrightarrow |(h_{i_0}.\zeta.\lambda) - (h_{i_1}.\zeta.\lambda)| \leq b$$

where b refers to a bound on the distance between two hosts to consider them connected. We can roughly determine the density of the query results by averaging across the number of one-hop neighbors, which we compute by applying the connectivity relation to hosts in the query result. This characterization of network density can be expressed as:

$$n = \frac{\langle \text{sum } h_{i_0}, h_{i_1} : h_{i_0} \in \rho_i \wedge h_{i_1} \in \rho_i \wedge (h_{i_0}, h_{i_1}) \in \mathcal{K} : 1 \rangle}{\langle \text{sum } h : h \in \rho_i :: 1 \rangle}$$

The distance metric can be expressed simply as the difference between n and the ideal number of neighbors (i.e., $d = |\text{ideal} - n|$). The application can dictate how the average connectedness measure relates to ideal network density and define a threshold that determines adequacy.

Semantic Discovery. It may also be desirable to determine inquiry strategy suitability based on the presence of a particular value:

$$n = \langle \text{sum } p : p \in \rho_i \wedge p.v = v :: 1 \rangle$$

where v is the desired value. A distance metric can be specified as $d = 1 - n$. For an application that wishes to be alerted of the presence of a single value, the associated threshold is specified as $\delta = 0$.

The value of interest may not be directly related to a query's reported result. Instead, causal relationships between different values may be the basis for adaptation. For example, the discovery of smoke on a construction site implies the presence of fire. If smoke is detected, then a inquiry mode which trades accuracy for overhead should be abandoned in favor of one that provides a higher accuracy in a search for a fire. These causal relationships can be captured in an adequacy metric by collecting and evaluating a metric over causally related values in the context field ζ of responding hosts.

5 Implementing Inquiry Modes and Introspection

We have implemented our model using Java²; we use the public interface presented below to demonstrate how our model can be realized in practice in a real

² The source code and settings used are available at
<http://mpc.ece.utexas.edu/InquiryMode/index.html>

application example. Fig. 2 shows this public interface, which includes a definition of a **Query** and its **InquiryMode**. Implementations of inquiry modes like the ones described in Section 2 extend these abstract classes with concrete functionality. For this discussion, consider a scenario where a construction site supervisor has deployed sensors across the site over which he issues queries to monitor the concentrations of hazardous airborne materials. The manner in which queries are processed should differ depending on the conditions on the site; our model captures this in a changing inquiry mode.

Phase 1. Initially, the supervisor may use a query with these characteristics:

- *Forward Function*: Probabilistic (Parameters: $p = 0.7$)
- *Respond Function*: Random Sampling (Parameters: $k = 0.5$)
- *Introspection*: Aggregate Data Capture quality (Parameters: $\delta = 0.1$)

To execute this query, concrete implementations of the probabilistic forward and random respond functions (such as the one shown in Fig. 3) need to be provided. When a node receives a query, it executes the query's forward and respond functions. If the forward function returns true given the host's current context, the host forwards the query. If the respond function returns true, the node processes the query at the application level. This entails updating the query with its data value(s) and the cost as defined by the introspection type. In this phase, the introspection cost is simply an aggregate on data quality, so no metadata outside the query result is required.

This query is adequate for baseline monitoring. Only a fraction of all the devices are involved in query processing, reducing resource usage. By choosing data quality as the introspection strategy, the supervisor can keep issuing such low cost queries until there is an indication of variance in data quality. In our example, a 10% change in the concentration indicates a chemical leak.

Phase 2. When a leak is detected, more serious monitoring is warranted. The first step is detecting where the leak emanates from.

```
class Query {
    public Query(InquiryMode inquiry, Introspection metadata);
}
class InquiryMode {
    public InquiryMode(ForwardFunction f, RespondFunction s);
}
abstract class ForwardFunction {
    abstract public boolean Execute(Context nodeContext);
}
abstract class RespondFunction {
    abstract public boolean Execute(Context nodeContext);
}
```

Fig. 2. The Inquiry and Introspection API

```

class FowardProbabilistic extends ForwardFunction {
    double p;
    FowardProbabilistic(Conext c) {
        p = c.probThreshold;
    }
    public boolean Execute(Context nodeContext) {
        return (nodeContext.getRandomNumber() < p);
    }
}

```

Fig. 3. Defining a probabilistic forward function

- *Forward Function*: Flooding (Parameters: None)
- *Store Function*: Flooding (Parameters: None)
- *Introspection*: Spatial Coverage (Parameters: 10 units)

By issuing such a query, the supervisor spends more resources by employing a flooding based inquiry mode to detect the location of the leak. By correlating those regions (obtained from the metadata information) with the response values, the application can establish regions of leakage.

Phase 3. Once the leak has been localized, the application can adapt the query once again to focus on the area of the leak:

- *Forward Function*: Location (Parameters: Area enclosing quadrant of site)
- *Store Function*: Location (Parameters: Area enclosing quadrant of site)
- *Introspection*: Data Quality (Parameters: $\delta = 0.2$)

Only devices in the vicinity of the leak respond to the query, and the user can get more detailed data from that region. In addition, this data can be collected more quickly since the network is focusing on a smaller amount of communication.

Results. Table 1 highlights the values for responses, metadata, and cost observed during the execution of this case study; in the table, $P1_2$ refers to the second query issued in Phase 1. When the first query is issued, the average concentration of chemicals is determined. Subsequent queries show a jump from an average that exceeds the threshold of 10% stipulated by the application. The application switches to Phase 2's flooding query, and the introspection strategy determines the query's spatial coverage. The supervisor identifies the location coordinates that have a high value for the amount of chemicals sensed. Once the areas of interest are located, he issues queries to get data only from those locations. At every step, he can evaluate the data against the cost of obtaining it. The cost expressed here is the number of messages to obtain the query result and its associated metadata. When the supervisor is obtaining just the data elements and using a low cost inquiry mode like random sampling, he incurs a lower cost. However, the cost increases to a higher value (175) when a more expensive inquiry mode like flooding is used. However, once this information is used to locate the area of interest, the supervisor can revert to an inquiry mode that

Table 1. Case Study query results by phases

	P1₁	P1₂	P1₃	P2₁	P3₁	P3₂
<i>Data Response</i>	484	504	559	604	609	598
<i>Spatial Coverage</i>	$\emptyset$	$\emptyset$	$\emptyset$	(900, 915), (805, 311), ... (700, 232)	$\emptyset$	$\emptyset$
<i>Cost</i>	19	10	23	175	45	51

restricts the overhead by scoping the query to a particular region of interest. This example demonstrates that introspection can provide great flexibility to the application developer and has the potential to save resources. Introspection thus provides an important mechanism in analyzing feedback, which is critical to developing intelligent adaptive systems.

6 Related Work

Our approach takes a novel perspective on modeling queries in mobile ad hoc networks by defining two new concepts: inquiry modes and query introspection. In this section, we examine related approaches with respect to modeling queries in these dynamic environments and adapting querying techniques.

Several related approaches to modeling dynamic environments rely on process calculi [7,8] or petri nets [9]. The former tend to focus on evolutionary changes (like our configuration changes), but make it difficult to capture the impact of time, space, and other constraints on query processing. The latter focus on low-level aspects of the environment such as packet transmission and energy consumption, lacking constructs to capture query processing behavior. More closely related work on coordination techniques for mobile and disconnected environments defined a concept similar to our reachability [10]: *disconnected routes*, which allow decoupling of mobile communication in space and time. The model, however, focuses on using the availability of motion profiles to plan nodes' interactions over time. Existing work in applying query processing to these dynamic environments focuses primarily on mobile distributed databases and the ability (or inability) of a system to provide traditional strong semantics (e.g., [11]). Our approach explicitly separates dissemination (through our forward function) from result generation (through our respond function). This approach in essence treats the entire network as a global virtual data structure, which is more in line with approaches targeted to database abstractions for sensor networks (e.g., [12]).

Our work also has some similarities to what can be broadly categorized as stream processing systems. Some of this work has explored model-driven query processing [13] where each node constructs a local model of the data available. If the estimated error of the model is below a threshold, a node processes a query over the local data model to avoid consuming resources. A model driven approach is less suitable for mobile environments because of the inherent unpredictability of movement. Our formalization of introspection provides a more

systematic approach to exposing relevant adequacy metrics (both data and network related) to facilitate adaptivity. By evaluating these metrics over values, informed decisions can be made on the trade-off between the cost of executing a particular query against application needs and switch inquiry modes as application requirements change. Similarly, *reflection* is common in mobile computing middleware and models [14]; our work recognizes the importance of reflection to the adaptivity of mobile applications and provides a formal foundation for exposing information about query results to applications through a principled use of introspection.

7 Conclusions

In this paper, we have presented a new perspective on query processing in mobile and pervasive networks. We presented a formalism for the concept of an *inquiry mode* that defines which devices participate in query resolution. In addition, we also formally defined the notion of *introspection* that helps evaluate the tradeoff between the cost of a chosen inquiry mode and its effectiveness by exposing relevant metadata in the form of adequacy measures. We showed how our model can be used to specify a variety of real world inquiry modes and adequacy measures. In addition, we provided a Java implementation that helps realize our model for practical application development and demonstrated its effectiveness.

References

1. Payton, J., Julien, C., Roman, G.C.: Automatic consistency assessment for query results in dynamic environments. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593. Springer, Heidelberg (2007)
2. Johnson, D.B., Maltz, D.A., Broch, J.: Dsr: The dynamic source routing protocol for multi-hop wireless ad hoc networks. *Ad Hoc Networking* 1, 139–172 (2001)
3. Perkins, C., Royer, E.: Ad hoc on-demand distance vector routing. In: Proc. of WMCSA (February 1999)
4. Ni, S.Y., Tseng, Y.C., Chen, Y.S., Sheu, J.P.: The broadcast storm problem in a mobile ad hoc network. In: Proc. of MobiCom, pp. 151–162 (1999)
5. Roman, G.C., Julien, C., Huang, Q.: Network abstractions for context-aware mobile computing. In: Proc. of ICSE, pp. 363–373 (2002)
6. Kyasanur, P., Choudhury, R., Gupta, I.: Smart gossip: An adaptive gossip-based broadcasting service for sensor networks. In: Proc. of MASS (October 2006)
7. Braione, P., Picco, G.P.: On Calculi for Context-Aware Coordination. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 38–54. Springer, Heidelberg (2004)
8. Lopes, L., Martins, F., Silva, M., Barros, J.: A process calculus approach to sensor network programming. In: Proc. of Sensorcomm, pp. 451–456 (2007)
9. Xiong, C., Murata, T., Tsai, J.: Modeling and simulation of routing protocols for mobile ad hoc networks using colored petri nets. In: Proc. of Wkshp. on Formal Methods Applied to Defense Systems, pp. 145–153 (2002)
10. Roman, G.C., Handorean, R., Sen, R.: Tuple space coordination across space and time. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 266–280. Springer, Heidelberg (2006)

11. Dunham, M., Helal, A., Balakrishnan, S.: A mobile transaction model that captures both the data and movement behavior. *ACM-Baltzer Journal on Mobile Networks and Applications* 2(2), 149–161 (1997)
12. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: The design of an acquisitional query processor for sensor networks. In: *Proc. of the 2003 ACM SIGMOD Int'l. Conf. on Management of Data*, pp. 491–502. ACM Press, New York (2003)
13. Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J., Hong, W.: Model-driven data acquisition in sensor networks. In: *Proc. of VLDB* (2004)
14. Capra, L., Blair, G.S., Mascolo, C., Emmerich, W., Grace, P.: Exploiting reflection in mobile computing middleware. *ACM SIGMOBILE Mobile Computing and Communications Review* 6(4), 34–44 (2002)

HOL-TESTGEN

An Interactive Test-Case Generation Framework

Achim D. Brucker¹ and Burkhart Wolff²

¹ SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
`achim.brucker@sap.com`

² Université Paris-Sud, Parc Club Orsay Université, 91893 Orsay Cedex, France
`wolff@lri.fr`

Abstract. We present HOL-TESTGEN, an extensible test environment for specification-based testing build upon the proof assistant Isabelle. HOL-TESTGEN leverages the semi-automated generation of test theorems (a form of partitioning the test input space), and their refinement to concrete test-data, as well as the automatic generation of a test driver for the execution and test result verification.

HOL-TESTGEN can also be understood as a unifying technical and conceptual framework for presenting and investigating the variety of unit test and sequence test techniques in a logically consistent way.

Keywords: symbolic test-case generations, black box testing, white box testing, theorem proving, interactive testing.

1 Introduction

HOL-TESTGEN (<http://www.brucker.ch/projects/hol-testgen/>) is an *interactive*, i. e., semi-automated, test tool for specification based tests built upon Isabelle/HOL. HOL-TESTGEN allows one to write test specifications in higher-order logic (HOL), (semi-) automatically partition the input space, resulting in abstract test-cases, automatically select concrete test-data, automatically generate test scripts for testing arbitrary implementations.

2 The HOL-TESTGEN Architecture and Workflow

In this section, we briefly review the main concepts and outline the standard workflow (see Figure 1) of HOL-TESTGEN [1–3]. The latter is divided into five phases: first, the *test theory* containing the basic datatypes and key predicates of the problem-domain has to be written. Since the test theory can be written in classical higher-order logic (HOL), i. e., a functional programming language extended by logical quantifiers, our approach is extremely flexible. Second, the test-engineer has to write the *test specification*, i. e., the concrete property the system under test is tested for. Third comes the generation of *test-cases* along with a *test theorem*, forth the generation of *test-data* (TD), and fifth the *test*

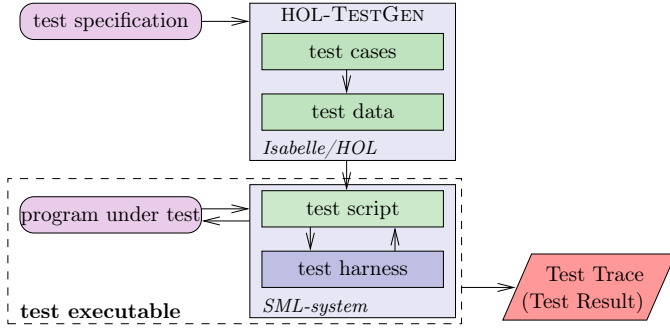


Fig. 1. Overview of the Standard Workflow of HOL-TESTGEN

execution (result verification) phase involving runs of the “real code” of the program under test. Once a test theory is completed, an integrated documentation (i.e., a formal test plan) with all definitions and results can be generated.

The properties of the *program under test* are specified in HOL in the *test specification* (TS). A test-specification, typically, will have the form $\text{pre } x \rightarrow \text{post } x$ ($\text{PUT } x$), where **pre** and **post** represent pre and post conditions of the program under test PUT , which is just a variable in the test-specification. Instead of just a partition of the input spaces, our system will decompose the test specification in the test-case generation phase into a semantically equivalent *test theorem* which has the form:

$$\llbracket \text{TD}_1; \dots; \text{TD}_n; \text{THYP } H_1; \dots; \text{THYP } H_m \rrbracket \implies \text{TS}$$

where THYP is a constant used to mark the *test hypotheses* that are underlying this test. At present, HOL-TESTGEN uses only uniformity and regularity Hypothesis; for example, a uniformity hypothesis means informally “if the program conforms to one instance of a case to TS, it conforms to all instances of this case to TS.” Thus, a test theorem has the following meaning: *If the program under test passes the tests for all TD_i successfully, and if it satisfies all test hypothesis, it conforms to the test specification.* In this sense, a test theorem bridges the gap between test and verification. The TD_i are just formulae so far, containing variables and arbitrary predicates of the test theory as well as the free variable referring to the system under test. In the data-selection phase, which is implemented by a constraint-resolution based on Isabelle’s proof procedures, ground instances for these variables were constructed. Both the test-case generation and the test-data-selection phase can be improved by adding lemmas (derived within Isabelle), or all sorts of logical massage can be realized by Isabelle on the test specification, the test theorem, the test-data, etc. This is how advanced users can improve the power of the deduction process dramatically.

The test theory containing test specifications, configurations of the test-data and test script generation, possibly extended by proofs for rules that support the overall process, is written in an extension of the Isar language [6]. It can be

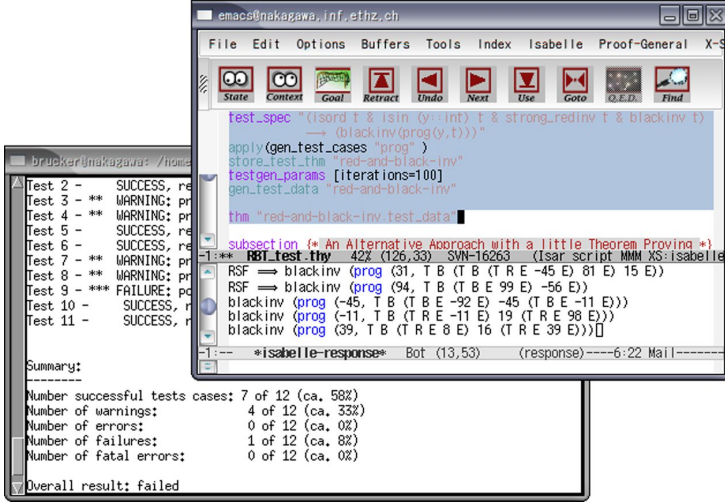


Fig. 2. A HOL-TESTGEN Session Using Proof General

processed in batch mode, but also using the Proof General interface interactively, see Figure 2. This interface allows for interactively stepping through a test theory (in the upper sub-window) and the sub-window below shows the corresponding system state. A system state may be a proof state in a test theorem development, or the result of inspections of generated test-data or a list of test hypothesis.

After test-data generation, HOL-TESTGEN can produce a *test script* driving the test using the provided *test harness*. The test script together with the test harness stimulate the code for the program under test built into the *test executable*. Executing the *test executable* runs the test and results in a *test trace* showing possible errors in the implementation (see lower window in Figure 2).

3 Case Studies

HOL-TESTGEN was used successfully in several case studies, among them:

Unit testing of red-black trees: In this case study [2], we generated test-cases for recursive data-structures. In particular, we generated test-cases for red-black trees testing the red-black properties (i. e., both the insertion and deletion operation preserve these properties). We also generated test-data and test scripts for this scenario and used them for testing the red-black tree implementation of the SML/NJ library. Our work revealed a major bug in this implementation which has not been detected during the last 12 years.

Unit testing of packet filters: In this case study [4], we modeled stateless packet filters (firewalls) and their security policy in HOL. Based on this specification, we generated test-cases for testing that a real firewall implements a specific security policy. Furthermore, we exploited the framework

aspect of HOL-TESTGEN and developed a domain-specific test case generator: HOL-TESTGEN/FW. HOL-TESTGEN/FW provides both domain specific test-case and test-data generation heuristics and domain-specific extensions of the theorem prover, e. g., supporting the simplification of firewall policies.

Sequence testing of application level firewalls: In this case study [3], we applied HOL-TESTGEN to different sequence-testing scenarios. In particular, we modeled stateful communication protocols (e. g., ftp and voice-over-ip) and used these models as basis for the test-case generation. Overall, this provides a method for testing the compliance of an application level, stateful firewall to a give security policy.

In all these applications, we made the experience that combining theorem proving techniques and testing techniques can improve the overall quality of the generated test-cases and test-data.

4 Conclusion

We provide a test environment for specification-based (also called model-based) unit and sequence testing. Moreover, our test environment bridges the gap between formal verification and testing techniques, i. e., testing, in a logically consistent way. The system has been used in several substantial case studies [2–4] and for test-theoretical work [5].

References

- [1] Brucker, A.D., Wolff, B.: HOL-TESTGEN 1.0.0 user guide. Technical Report 482, ETH Zurich (April 2005)
- [2] Brucker, A.D., Wolff, B.: Symbolic test case generation for primitive recursive functions. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 16–32. Springer, Heidelberg (2005)
- [3] Brucker, A.D., Wolff, B.: Test-sequence generation with HOL-TESTGEN with an application to firewall testing. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 149–168. Springer, Heidelberg (2007)
- [4] Brucker, A.D., Brügger, L., Wolff, B.: Model-based firewall conformance testing. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 103–118. Springer, Heidelberg (2008)
- [5] Brucker, A.D., Brügger, L., Wolff, B.: Verifying test-hypotheses: An experiment in test and proof. *Electronic Notes in Theoretical Computer Science* 220(1), 15–27 (2008); proceedings of the Fourth Workshop on Model Based Testing (MBT) (2008) ISSN 1571-0661, doi(10.1016/j.entcs, 11.003)
- [6] Wenzel, M.M.: Isabelle/Isar – a versatile environment for human-readable formal proof documents. PhD thesis, TU München, München (February 2002)

CADS*: Computer-Aided Development of Self-* Systems

Radu Calinescu and Marta Kwiatkowska

Computing Laboratory, University of Oxford, UK
{Radu.Calinescu,Marta.Kwiatkowska}@comlab.ox.ac.uk

Abstract. We present the prototype tool CADS* for the computer-aided development of an important class of self-* systems, namely systems whose components can be modelled as Markov chains. Given a Markov chain representation of the IT components to be included into a self-* system, CADS* automates or aids (a) the development of the artifacts necessary to build the self-* system; and (b) their integration into a fully-operational self-* solution. This is achieved through a combination of formal software development techniques including model transformation, model-driven code generation and dynamic software reconfiguration.

1 Introduction

The ever growing complexity of today's IT systems has led to unsustainable increases in their management and operation costs. Software architects and developers aim to alleviate this problem by building *self-** (or *autonomic*) *systems*, i.e., systems that self-configure, self-optimize, self-protect and self-heal based on a set of high-level, user-specified objectives [4,8].

The architecture of a self-* system is depicted in Fig. 1. Given a set of user-specified system objectives (or *policies*), an *autonomic manager* monitors the system components through *sensors*, uses its *knowledge* (i.e., *model* of the system) to analyse their state and to plan changes in their configurable parameters, and implements these changes through *effectors*. In recent work, we introduced a general-purpose implementation of an autonomic manager as a reconfigurable *policy engine* [1], and we described how quantitative analysis methods from the area of probabilistic model checking can be used to implement user-specified policies in a self-* system [2,3].

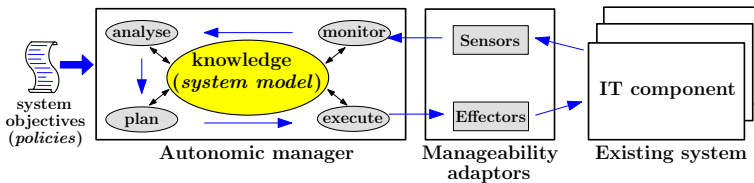


Fig. 1. High-level architecture of a self-* system

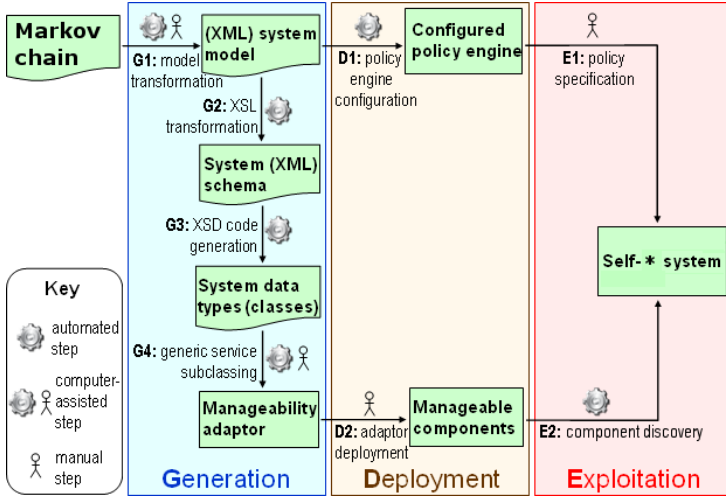


Fig. 2. The CADS* development process

In this paper, we introduce CADS*, a tool for the computer-aided development of self-* systems whose components can be modelled as Markov chains. The tool takes as input a continuous- or discrete-time Markov chain (CTMC or DTMC) describing the behaviour of the IT components to be included in the self-* system. This Markov chain is expressed in the high-level modelling language used by the probabilistic model checker PRISM [5], and may be available already from the verification of the IT components.¹ Alternatively, the Markov chain can be built as described in [6,5].

Starting from the Markov chain mentioned above, CADS* reduces the effort and expertise required to develop self-* systems by automating or guiding: (a) the generation of the system model used to set up the reconfigurable policy engine from [1]; (b) the generation of the manageability adaptors from Fig. 1; and (c) the configuration of the policy engine for the planned self-* solution.

2 CADS* Development Process

The three-stage development process implemented by CADS* is shown in Fig. 2, and presented below.

Generation stage. This stage starts with the developer uploading into CADS* a Markov chain describing the behaviour of the targeted IT components, and expressed in the PRISM modelling language [5]. In a first step (labelled **G1** in Fig. 2), CADS* employs a model transformation to derive an XML-encoded model of the system to be managed, as described by the mapping

$$modelTransformation : MarkovChain \rightarrow SystemModel, \quad (1)$$

¹ PRISM models for a wide range of system components are available from [9].

where *MarkovChain* and *SystemModel* represent the set of Markov chains accepted by PRISM and the set of models used to configure the policy engine from [1]. Step **G1** is computer-assisted, i.e., the developer is required to allocate the system parameters that CADS* identifies in the Markov chain to IT components, and to partition them into read-only *state parameters* and read-write *configuration parameters*.

In step **G2**, CADS* uses an XSL transformation to automatically extract an XML schema specification for the targeted IT components from the result of (1):

$$schemaGen : SystemModel \rightarrow XmlSchema. \quad (2)$$

In step **G3**, the tool runs an instance of the XML Schema Definition tool [7] to generate the set of data types associated with the XML schema:

$$dataTypeGen : XmlSchema \rightarrow 2^{DataType}. \quad (3)$$

The result is a set of .NET classes. Finally, we implemented a model-driven code generation module that CADS* uses in step **G4** to automate the generation of web service stubs for the manageability adaptors in Fig. 1:

$$adaptorGen : XmlSchema \rightarrow 2^{ManageabilityAdaptor}. \quad (4)$$

These stubs subclass a generic abstract web service *ManagedResource*<*T*> that implements the bulk of the sensor and effector functionality associated with the manageability adaptor for an IT component (or *resource*). At the end of this computer-assisted step, CADS* requires that the developer implements manually a couple of simple, component-specific methods that are declared **abstract** in *ManagedResource*<*T*>—the work involved is described in [1].

Deployment stage. In step **D1**, the developer provides the URL of a running instance of the policy engine from [1], and CADS* calls the appropriate web method to supply the model from (1) to this policy engine. In step **D2**, the manageability adaptors from step **G4** are deployed manually, and configured to access the IT components to which autonomic capabilities are being added.

Exploitation stage. In step **E1**, user-specified policies are forwarded by CADS* to the policy engine configured in step **D1**. These policies are specified in the policy expression language described in [2,1], by using a combination of arithmetic, logic, relational and string operators, and optimisation functions such as MIN, MAX and GOAL to construct well-defined policies as expressions of the system parameters identified in step **G1**. Finally, the policy engine applies these policies to the IT components exposed by the manageability adaptors it discovers automatically in step **E2**.

3 Tool Validation

In order to assess the effectiveness of CADS*, we used it to re-implement two self-* systems that we had previously developed manually in [2,3]. The first system was a self-configuring/self-protecting system whose objective was to maintain user-specified levels of availability for a set of data-centre clusters. This

objective was achieved by automatically adapting the number of servers allocated to each cluster to changing cluster workloads, priorities and target availabilities. The second system was a self-optimising system involving the dynamic power management of a disk drive, and had as objective the optimisation of user-specified trade-offs between the performance and the power usage of a disk drive exposed to variable workloads.

In both cases, we started from existing PRISM CTMCs from [9], and we successfully devised operational prototypes of the planned self-* system in approximately a tenth of the time taken to develop an equivalent solution manually (i.e., under one day compared to over a week). This gain was primarily due to CADS* reducing significantly the potential for developer error through automating or aiding the development of the self-* system artifacts, and their integration into a fully operational solution.

4 Conclusion

We introduced the prototype computer-aided development tool CADS*, and briefly described how its use in two case studies sped up the development of self-* systems compared to implementing equivalent systems manually. Future work includes augmenting CADS* with the ability to aid users in the specification of valid, non-conflicting system objectives, and to validate the tool further by exposing it to developers with limited expertise in self-* system development.

Acknowledgement. This work was partly supported by the UK Engineering and Physical Sciences Research Council grant EP/F001096/1.

References

1. Calinescu, R.: Implementation of a generic autonomic framework. In: Proc. 4th Intl. Conf. Autonomic and Autonomous Systems, pp. 124–129 (2008)
2. Calinescu, R.: General-purpose autonomic computing. In: Denko, M., et al. (eds.) *Autonomic Computing and Networking*. Springer, Heidelberg (2009)
3. Calinescu, R., Kwiatkowska, M.: Software engineering techniques for the development of systems. In: Proc. 15th Monterey Workshop on Foundations of Computer Software, pp. 86–93 (2008)
4. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *IEEE Computer Journal* 36(1), 41–50 (2003)
5. Kwiatkowska, M., et al.: Quantitative analysis with the probabilistic model checker PRISM. *Electronic Notes in Theoretical Computer Science* 153(2), 5–31 (2005)
6. Kwiatkowska, M., et al.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation*, pp. 220–270. Springer, Heidelberg (2007)
7. Microsoft Corporation. Xml schema definition tool (xsd.exe) (2007), [http://msdn2.microsoft.com/en-us/library/x6c1kb0s\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/x6c1kb0s(VS.80).aspx)
8. Parashar, M., Hariri, S.: *Autonomic Computing: Concepts, Infrastructure & Applications*. CRC Press, Boca Raton (2006)
9. PRISM Case Studies, <http://www.prismmodelchecker.org/casestudies>

HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis*

Qichang Chen¹, Liqiang Wang^{1,**}, Zijiang Yang², and Scott D. Stoller³

¹ Dept. of Computer Science, University of Wyoming, WY, USA
`{qchen2,wang}@cs.uwyo.edu`

² Dept. of Computer Science, Western Michigan University, MI, USA
`zijiang.yang@wmich.edu`

³ Computer Science Dept., Stony Brook University, NY, USA
`stoller@cs.stonybrook.edu`

Abstract. The reality of multi-core hardware has made concurrent programs pervasive. Unfortunately, writing correct concurrent programs is difficult. Atomicity violation, which is caused by concurrently executing code unexpectedly violating the atomicity of a code segment, is one of the most common concurrency errors. However, atomicity violations are hard to find using traditional testing and debugging techniques.

This paper presents a hybrid approach that integrates static and dynamic analyses to attack this problem. We first perform static analysis to obtain summaries of synchronizations and accesses to shared variables. The static summaries are then instantiated with runtime values during dynamic executions to speculatively approximate the behaviors of branches that are not taken. Compared to dynamic analysis, the hybrid approach is able to detect atomicity violations in unexecuted parts of the code. Compared to static analysis, the hybrid approach produces fewer false alarms. We implemented this hybrid analysis in a tool called **HAVE** that detects atomicity violations in multi-threaded Java programs. Experiments on several benchmarks and real-world applications demonstrate promising results.

1 Introduction

Today, multi-core hardware has become ubiquitous, which puts us at a fundamental turning point in software development. In order for software applications to benefit from the continued exponential throughput advances in new processors, the applications will need to be well-written multi-threaded programs. However, writing correct multi-threaded programs is difficult, because concurrency can introduce subtle errors that do not exist in sequential programs, if concurrent accesses to shared data are not properly synchronized.

* The work was supported in part by Wyoming NASA Space Grant Consortium, NASA Grant NNG05G165H, Wyoming NASA EPSCoR, NASA Grant NCC5-578, NSF CCF-0811287, CNS-0831298, CNS-0627447, CCF-0613913, and CNS-0509230.

** Corresponding author.

Two of the most common concurrency errors are data races and atomicity violations. A data race occurs when two concurrent threads perform conflicting accesses (*i.e.*, accesses to the same shared variable and at least one access is a write) and the threads use no explicit mechanism to prevent the accesses from being simultaneous. In Program 1 of Figure 1, conflicting accesses to the shared variable `bal` can happen simultaneously without any protecting lock, hence a data race occurs. An atomicity violation occurs when an interleaved execution of a set of code blocks (expected to be atomic) by multiple threads is not equivalent to any serial execution of the same code blocks. Program 2 in Figure 1 eliminates the data race in Program 1 by adding a lock `o`. However, Program 2 is still incorrect if the `deposit` method is required to be atomic. An atomicity violation occurs in Program 2 when the two synchronization blocks in thread 2 execute between the two synchronization blocks in thread 1.

Program 1		Program 2	
Thread 1	Thread 2	Thread 1	Thread 2
<code>deposit(int val){</code>	<code>deposit(int val){</code>	<code>deposit(int val){</code>	<code>deposit(int val){</code>
<code>int tmp = bal;</code>	<code>int tmp = bal;</code>	<code>synchronized(o){</code>	<code>synchronized(o){</code>
<code>tmp = tmp + val;</code>	<code>tmp = tmp + val;</code>	<code>int tmp = bal;</code>	<code>int tmp = bal;</code>
<code>bal = tmp;</code>	<code>bal = tmp;</code>	<code>tmp = tmp + val;</code>	<code>tmp = tmp + val;</code>
<code>}</code>	<code>}</code>	<code>}</code>	<code>}</code>
		<code>synchronized(o){</code>	<code>synchronized(o){</code>
		<code>bal = tmp;</code>	<code>bal = tmp;</code>
		<code>}</code>	<code>}</code>
		<code>}</code>	<code>}</code>

Fig. 1. Examples in Java demonstrating data races and atomicity violations

Most existing approaches to detect atomicity violations are either purely dynamic (*e.g.* [6,21,20,19]) or purely static (*e.g.* [9,7]). The strength of static analysis is that it can consider all possible behaviors of a program. However, it may produce false positives (*i.e.*, false alarms), because some aspects of a program's behavior, such as alias relationships, values of array indices, and happens-before relationships, are very difficult to analyze statically. Moreover, many static analyses, such as the type system for atomicity in [9], require either manual annotation of the program or rewriting of the program into a special language. Dynamic analysis observes and analyzes the actual behaviors of a program by executing it. Generally, dynamic analysis is unsound compared to static analysis, because it does not analyze unobserved behaviors of programs. On the positive side, it generally produces much fewer false positives. Furthermore, dynamic analysis generally does not require manual annotation of code that is often required in static analysis; this is a significant practical advantage.

In order to exploit the complementary benefits of static and dynamic analyses, we propose a hybrid approach to detect atomicity violations. In our approach, we perform a conservative intraprocedural static analysis to generate a summary for each method in the program. Our runtime system tracks and records the values of reference variables during execution. When we observe an unexecuted branch during dynamic analysis, the static summary of that unexplored branch

is retrieved and instantiated using the recorded values. Thus, the instantiated summary speculatively approximates what would have happened if the branch had been executed.

We implemented the hybrid approach in a tool called *Hybrid Atomicity Violation Explorer (HAVE)* for detecting atomicity violations in multi-threaded Java programs and evaluated it on several benchmarks and real-world applications. The experiments show that the hybrid approach reports fewer false positives than the previous static approaches [9,1], and fewer false negatives (*i.e.*, missed errors) than the previous dynamic approaches [6,20,19].

The rest of this paper is organized as follows. Section 2 formally defines atomicity violations. Section 3 presents the architecture of our tool HAVE. Section 4 introduces the conflict-edge algorithm. Section 5 describes some optimizations. Section 6 presents the experimental results. Section 7 reviews the related work. Section 8 discusses the conclusions and future work.

2 Atomicity Violations

An execution $\sigma = \langle s_1, \dots, s_n \rangle$ is a sequence of accesses to shared variables, lock acquire, lock release, thread **start**, thread **join**, and **barrier** synchronization operations.

A *transactional unit* (or *transaction*) is an execution of a code block expected to be atomic. A *non-transactional unit* is an execution of a code block not expected to be atomic. For an event or transaction x , let $th(x)$ be the thread that performed x . As in [19], we assume that transaction boundaries are chosen so that thread start and join operations and barrier operations occur at transaction boundaries, not in the middle of transactions. Thus, thread and barrier operations induce a partial order on transactions: given an execution σ , and two transactional or non-transactional units u_1 and u_2 , u_1 *happens-before* u_2 , denoted $u_1 <_H u_2$, if (1) $th(u_1) = th(u_2)$ and u_1 is executed before u_2 , or (2) $th(u_1) \neq th(u_2)$ and (2a) $th(u_1)$ starts thread $th(u_2)$ after executing u_1 or (2b) $th(u_2)$ joins on $th(u_1)$ before executing u_2 , or (3) $\exists u_i : (u_1 <_H u_i) \wedge (u_i <_H u_2)$. From monitoring an execution, we extract a set T of transactions, a set A of non-transactional units, and a happens-before relation $<_H$.

Given $\langle T, A, <_H \rangle$, a *trace* of $\langle T, A, <_H \rangle$ is an interleaving of events from units in $T \cup A$ that is consistent with the happens-before relation $<_H$ (*i.e.*, if $u_1 <_H u_2$, then all events in u_1 precede all events in u_2) and respects locking (*i.e.*, for every matching pair of acquire and release operations that belong to the same thread, no acquire or release of the same lock by other threads happens between them).

Traces π_1 and π_2 for $\langle T, A, <_H \rangle$ are *equivalent* if (1) they contain the same events, and (2) for each pair of conflicting accesses, the two accesses appear in the same order in both traces.

A trace of $\langle T, A, <_H \rangle$ is *serial* if the events of each transaction in T form a contiguous subsequence of the trace. $\langle T, A, <_H \rangle$ is *atomic* if every trace of $\langle T, A, <_H \rangle$ has an equivalent serial trace of $\langle T, A, <_H \rangle$.

For example, consider Program 2 in Figure 1. Suppose the method `deposit` is expected to be atomic. The program has only two serial executions, $[t_1.R(bal) \ t_1.W(bal) \ t_2.R(bal) \ t_2.W(bal)]$ and $[t_2.R(bal) \ t_2.W(bal) \ t_1.R(bal) \ t_1.W(bal)]$, where $t.A(x)$ denotes that thread t performs action A on variable x . The interleaved execution $[t_1.R(bal) \ t_2.R(bal) \ t_2.W(bal) \ t_1.W(bal)]$ is not equivalent to any serial trace, hence, the execution of method `deposit` is not atomic.

This notion of atomicity is also called *conflict atomicity* [19]. In [19], we also explored another notion of atomicity, called *view atomicity*. We do not consider view atomicity in this paper because checking it is more expensive and gives the same results as checking conflict atomicity in our experiments [19].

In this paper, we assume that the program does not have *potential for deadlock* (*i.e.*, some trace of the program may end in deadlock). This assumption is needed because a trace that ends in deadlock with some thread in the middle of a transaction is not equivalent to any serial trace. Potential for deadlock can be checked using our approach in [2].

3 Integrating Dynamic and Static Analyses

This section gives an overview of our hybrid approach to check atomicity violations. Figure 2 shows the architecture of our tool, HAVE, which consists of five components.

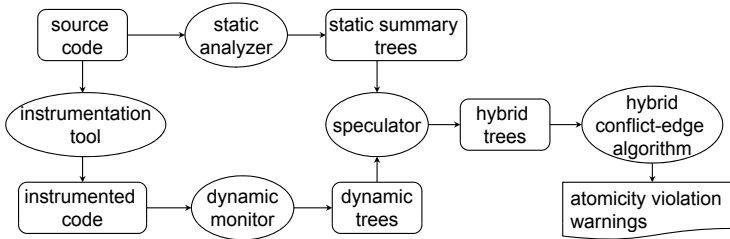


Fig. 2. The architecture of the tool HAVE

1. A static analyzer, which parses the source code to generate *static summary trees* (SSTs).
2. An instrumentation tool, which inserts event interception code.
3. A dynamic monitor, which intercepts events and builds *dynamic trees* during execution.
4. A speculator, which generates speculations for the unexecuted branches from SSTs and combines them with dynamic trees to form *hybrid trees*.
5. A detector, which analyzes the hybrid trees for atomicity violations using the hybrid conflict-edge algorithm described in Section 4.

3.1 The Static Analyzer

The static analyzer parses source code to construct static summary trees (SSTs). Each SST corresponds to a method in a certain class. Specifically, a static tree may contain nodes representing: (1) read/write to non-**final** and non-**volatile** fields; or (2) entrance and exit of synchronized blocks, including synchronized methods (which represent lock acquire and release operations); or (3) control-flow structures, namely, **if**, **for/while**, and **switch/case**; or (4) assignments to reference variables, which are used to speculate reference changes for the unexecuted code blocks. SSTs do not contain interprocedural information, *i.e.*, method calls are ignored. Unlike the dynamic monitor, the static analyzer ignores thread **start**, **join** and barrier synchronizations. Accesses to array elements are ignored in this paper due to the difficulty of statically resolving the indices of array elements. Figure 3 shows an example of a code block and its SST.

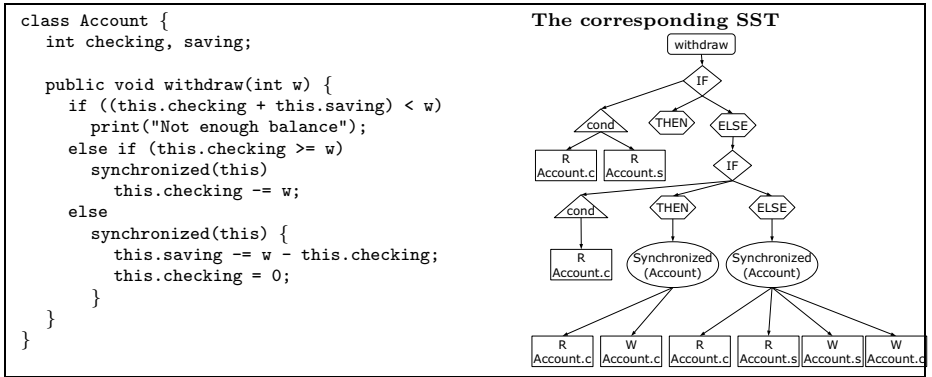


Fig. 3. An example of a static summary tree (SST), where `Account.c` and `Account.s` denote `Account.checking` and `Account.saving`, and “R” or “W” denotes that the node is a read or write, respectively.

3.2 Instrumentation

The instrumentation component instruments source code in order to intercept specific events during execution. The intercepted events include program control-flow structures, reads and writes to non-**final** and non-**volatile** fields, synchronization (including lock **acquire** and **release**, barrier operation, thread **start** and **join**), assignments to reference variables, and transaction boundaries.

Similar to [19], executions of the following code fragments are considered as transactions by default because their executions are often expected to be atomic by programmers: non-private methods, synchronized private methods, and synchronized blocks inside non-synchronized private methods. With exceptions, the executions of the `main()` method in which the program starts and the executions of `run()` methods of classes that implement `Runnable` are not considered as

transactions, because these executions represent the entire executions of threads and are often not expected to be atomic. Moreover, **start**, **join** and barrier operations are treated as boundaries, *i.e.*, they separate the preceding events and following events into different units, and are not contained in any unit. We adopt this heuristic because execution fragments containing these operations are typically not atomic and hence are not expected to be transactions. The events not in transactions form non-transactional units. Note that for nested transactions, we check atomicity only for the outmost transactions, since they contain the inner transactions. The defaults can be overridden using a configuration file.

3.3 The Dynamic Monitor and Speculator

When an instrumented program runs, the dynamic monitor receives events issued by the instrumented code. The events of each unit (including transactional and non-transactional units) are stored in a structure called a *hybrid tree*, which consists of events observed in the execution and speculations based on static summary trees. Each leaf node represents a read or write to a shared variable and contains the runtime identifier for the shared variable. For example, `R(320.checking)` denotes a read to the field “**checking**” of an object identified by its hashcode 320. Each non-leaf node except for the root represents a lock-based synchronization block or control-flow structure (*e.g.* **if/then/else**, **for/while** loop, **switch/case**). Each synchronization node contains the runtime identifier for the current lock (*i.e.*, synchronization object). The root node simply identifies the unit.

For each unexecuted branch in the unit, we instantiate the corresponding part of the method’s SST by simulating its execution using the runtime context at the associated branch point, and add the resulting concrete events (*e.g.* synchronization nodes, reads and writes) to the hybrid trees. We instantiate symbolic names in the SST by querying binding tables. A binding table is maintained for each object; it stores the mappings between symbolic names and runtime values of all reference fields and local reference variables under the context of the object. A binding table is maintained for each class with static reference fields. Binding tables are updated when assignments to reference variables are executed. During speculative execution, assignments to reference variables in SSTs trigger updates on temporary copies of binding tables, instead of the original ones. Since there might be unresolved symbolic names left during speculation, the speculation may be not as accurate as its runtime equivalent observed in the dynamic analysis if it can be executed. This speculative technique may lead to false positives. Our experiments show that such kind of false positives are very rare in practice.

Our speculative execution also constructs subtrees corresponding to speculative iterations of loop bodies. According to Theorem 3 in Section 4.3, if all iterations of a loop perform the same accesses, then at most two iterations are needed to detect atomicity violations. We use this as a heuristic, without attempting to verify the hypothesis of the theorem. Specifically, when the control flow reaches a loop, if the execution contains no iterations of the loop at that

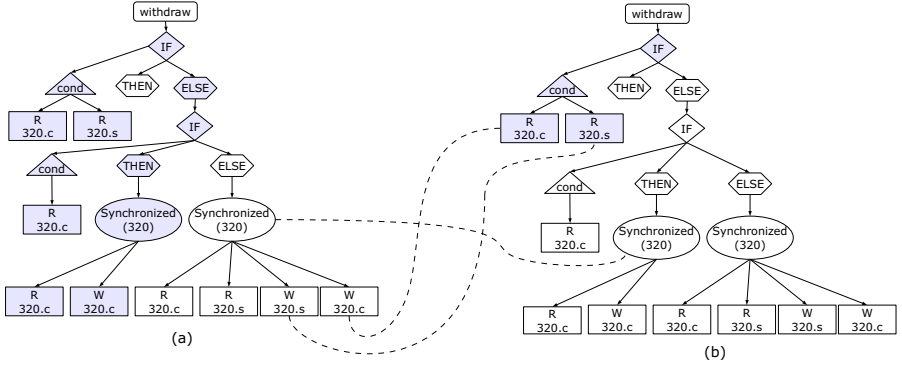


Fig. 4. An example of hybrid trees. Tree (a) corresponds to an execution of `withdraw` in Figure 3 when the `else if` is executed; tree (b) corresponds to the scenario when the first `then` is executed. The grey nodes are generated from the real executions; all the other nodes are speculated. The dotted lines denote conflict-edges introduced in Section 4. Only partial conflict-edges are marked out.

point, we add two speculative iterations; if the execution contains only one iteration of the loop at that point, we add one speculative iteration.

Two examples of hybrid trees are shown in Figure 4. Tree (a) and Tree (b) are generated by two threads of an execution that call the method `withdraw()` in Figure 3. The hashcode of the instance of `Account` is assumed to be 320.

4 The Conflict-Edge Algorithm

This section presents the conflict-edge algorithm that detects atomicity violations based on hybrid trees. The algorithm adds edges, called *conflict-edges*, between hybrid trees, to connect two conflicting nodes (which is discussed in details in Section 4.1). The algorithm then generates all valid pairs of conflict-edges; informally, “valid” means that all nodes involved in the pair can coexist in some execution. The algorithm detects and reports atomicity violations by analyzing each valid conflict-edge pair. Note that the conflict-edge algorithm does not merely look for violations of atomicity in the observed execution, but also determines whether atomicity violations exist in feasible permutations of the observed execution.

4.1 Building Conflict-Edges between Hybrid Trees

Two nodes n_1 and n_2 *conflict* if (1) they are in different hybrid trees, and (2) they represent accesses to the same variable and at least one of them is a write, and (3) thread `start`, `join`, and barrier operations do not induce a happens-before relation on them (*i.e.*, do not prevent them from occurring simultaneously). Let

$held(n_i)$ denote the set of locks held when n_i is executed, which is determined by the synchronization nodes that are ancestors of n_i in the tree.

For each pair (n_1, n_2) of conflicting nodes, if $held(n_1) \cap held(n_2) = \emptyset$, we add a conflict-edge between n_1 and n_2 ; otherwise, we add a conflict-edge between the highest ancestors of n_1 and n_2 that are synchronization nodes for the same lock. The highest synchronization nodes represent the outmost common lock held during the executions of n_1 and n_2 . The conflict-edge reflects the granularity at which the code blocks containing conflicting accesses can be interleaved. For example, Figure 4 shows partial conflict-edges between the two hybrid trees.

4.2 Detecting Atomicity Violations

A hybrid tree t represents a set $\llbracket t \rrbracket$ of possible (transactional or non-transactional) execution units, corresponding to different choices of the branches of the **if**, **switch**, and loop statements that appear in it. For simplicity, our speculative analysis assumes that each branch could be taken, independently of other choices; in other words, the conditions guarding the branches are ignored. Given a set $T = \{t_1, \dots, t_n\}$ of hybrid trees representing transactions, a set $A = \{a_1, \dots, a_m\}$ of hybrid trees representing non-transactional units, and a happens-before relation $<_H$ on these trees, $\langle T, A, <_H \rangle$ is *atomic* if, for all $t'_1 \in \llbracket t_1 \rrbracket, \dots, t'_n \in \llbracket t_n \rrbracket, a'_1 \in \llbracket a_1 \rrbracket, \dots, a'_m \in \llbracket a_m \rrbracket, \langle \{t'_1, \dots, t'_n\}, \{a'_1, \dots, a'_m\}, <'_H \rangle$ is atomic, where two execution units are related by $<'_H$ iff the hybrid trees they were generated from are related by $<_H$.

Conflict-edges e and e' are *incompatible* if one end node of e and one end node of e' appear in mutually exclusive branches of a hybrid tree, such as **then** and **else** branches of the same **if** statement, or different cases of a **switch** statement; otherwise, conflict-edges e and e' are *compatible*.

Conflict-edge e is an *ancestor* of conflict-edge e' in hybrid tree t if an endpoint of e is an ancestor of an endpoint of e' in t . A pair (e, e') of conflict-edges is *valid* for hybrid tree t , if (1) e and e' are compatible, (2) e is not an ancestor of e' in t , and vice versa, and (3) e and e' are incident on different nodes of t . In the rest of the paper, all pairs mentioned are valid by default if without explicit indication.

We have the following theorem to determine atomicity for a transactional hybrid tree. Let F_σ be a hybrid forest generated from an execution σ . Given a transactional hybrid tree t contained in F_σ , let $F_\sigma \setminus \{t\}$ be the set of all the other units.

Theorem 1. *Suppose a hybrid forest F_σ has no potential for deadlock. If t has no valid pair in $F_\pi, \langle \{t\}, F_\sigma \setminus \{t\}, <_H \rangle$ is atomic.*

Proof Sketch: If t does not have valid pairs in F_π , for every trace of $\langle \{t\}, F_\sigma \setminus \{t\}, <_H \rangle$, there are only two possible cases: (1) t has at most one node with an incident conflict-edge; or (2) t has a set S of nodes with incident conflict-edges, and for all $n_1, n_2 \in S$, n_1 is an ancestor or descendant of n_2 . In the first case, for each non-serial trace, we can construct an equivalent serial trace by commuting all events in t to the position of that node. In the second case, we can construct


```

CheckAtomicityViolations() {
   $AVScenarios := \emptyset$ ;
  for each transactional hybrid tree  $t$  do
    for each valid conflict-edge pair  $(e, e')$  of  $t$  do
      if only two hybrid trees including  $t$  are connected by  $e$  and  $e'$  then
        /* find an atomicity violation scenario */
         $AVScenarios := AVScenarios \cup \{(e, e')\}$ ;
      else
        if  $\exists$  a valid cycle  $c$  of conflict-edges involving  $(e, e')$  then
           $AVScenario := AVScenario \cup \{c\}$ ;
}

```

Fig. 5. The conflict-edge algorithm to detect atomicity violations

an equivalent serial trace by commuting all events in t to the position of the lowest node in S . Hence, $\langle \{t\}, F_\sigma \setminus \{t\}, <_H \rangle$ is atomic. $\square$

Given a valid pair of conflict-edges (e, e') , a sequence of conflict-edges involving e and e' may form into a cycle, where two conflict-edges connected to the same tree are considered linked. A cycle involving (e, e') is *valid* if (1) neither e nor e' is an ancestor of the other; and (2) all conflict-edges on the cycle are compatible; and (3) for each node n on the cycle, $held(n) \cap held(n_e^t) = \emptyset \wedge held(n) \cap held(n_{e'}^t) = \emptyset$, where n_e^t and $n_{e'}^t$ are the end nodes of e and e' in t , respectively; and (4) all involved transactional and non-transactional units are concurrent (i.e., the happens-before relation enforces no order on them). We have the following theorems to check atomicity violations.

Theorem 2. *Suppose a hybrid forest F_σ has no potential for deadlock. If a valid pair of conflict-edges (e, e') on a transactional hybrid tree t is involved in a valid cycle of F_σ , then t has an atomicity violation with the scenario indicated by the cycle.*

Proof Sketch: Suppose the valid cycle consists of conflict-edges $e_0, e_1, \dots, e_n$, where $e_0 = e$ and $e_n = e'$. Let u_i and u_{i+1} be the execution units containing the endpoints of e_i , for $i = 0..n$. Note that $u_0 = t$ and $u_{n+1} = t$. The conditions in the definition of valid cycle imply that $u_0, \dots, u_n$ are distinct and there exist $u'_0 \in \llbracket u_0 \rrbracket, \dots, u'_n \in \llbracket u_n \rrbracket$ and an interleaving σ' for $\langle T', A', <'_H \rangle$ that contains events in the order $\langle endpoint(e_0, t), endpoint(e_0, u_1), endpoint(e_1, u_1), endpoint(e_1, u_2), \dots, endpoint(e_n, u_n), endpoint(e_n, t) \rangle$ (i.e., all of the other nodes on the cycle occur between the two nodes of t on the cycle), where T' and A' contain the transactional and non-transactional units, respectively, in $\{u'_0, \dots, u'_n\}$, and $u'_i <'_H u'_j$ iff $u_i <_H u_j$. Because the two endpoints of a conflict-edge represent conflicting accesses to a shared variable, all executions equivalent to σ must preserve the order of $t, u_1, \dots, u_n, t$. Thus, there is no serial execution (in particular, no execution in which t occurs serially) equivalent to σ , so the cycle indicates an atomicity violation. $\square$

Corollary 1. *Suppose a hybrid forest has no potential for deadlock. If a valid pair of conflict-edges on a transactional hybrid tree t is incident to only two*

transactions (including t), then t has an atomicity violation with the scenario indicated by the pair.

Proof Sketch: The valid pair forms into a cycle. Thus, the conclusion is simply implied by Theorem 2. $\square$

For each hybrid tree t , we detect atomicity violations by checking valid pairs of conflict-edges as shown by `CheckAtomicityViolations()` in Figure 5. Given a valid pair (e, e') of t , if e and e' involve only two hybrid trees, this pair implies an atomicity violation by Corollary 1. If e and e' involve three hybrid trees (recall that e and e' are already incident to t), we check atomicity violations based on Theorem 2. Our current implementation does not use Theorem 1, because our system looks for potential atomicity violations; it does not try to verify atomicity. Corollary 1 is applied first because it is cheaper.

For example, in Figure 4, an atomicity violation is revealed by a valid pair $(\langle \text{a.W}(320.\text{c}), \text{b.R}(320.\text{c}) \rangle, \langle \text{a.W}(320.\text{s}), \text{b.R}(320.\text{s}) \rangle)$. The atomicity violation cannot be discovered by a purely dynamic approach because the valid pair connects a speculative branch in tree **a** with an executed branch in tree **b**.

Let S (mnemonic for Size of trees) denote the maximum number of nodes in any hybrid tree. Note that the number of trees is $|T \cup A|$. The worst-case time complexity of constructing conflict-edges is $O((|T \cup A| \times S)^2)$. Let n_c denote the maximum number of conflict-edges incident on any hybrid tree. Usually n_c is much less than $|T \cup A| \times S^2$. Theorem 2 requires finding a valid cycle, which is $O((|T \cup A| \times n_c)^2)$. The total number of valid pairs is $O(T \times n_c^2)$. Hence, the worst-case time complexity of checking atomicity violations is $O(|T| \times |T \cup A|^2 \times n_c^4)$.

4.3 Unwrap Loops

The following theorem shows that executing a loop twice is sufficient to find atomicity violations, if all iterations perform the same accesses.

Consider a loop such that every iteration contains the same sequence of access events. Let σ_2 denote an execution in which, at some point, a thread performs exactly two iterations of the loop. Let t_2 be the corresponding transaction containing the two iterations. Let σ_m be an execution that differs from σ_2 only in that, at the same point, the thread performs more than two iterations. Let t_m be the corresponding transaction containing the m iterations. Suppose t_2 and t_m differ only on the number of iterations.

Theorem 3. $\langle t_2, A, \langle_H \rangle$ is not atomic iff $\langle t_m, A, \langle_H \rangle$ is not atomic.

Proof Sketch: “ $\Rightarrow$ ”: it is obvious.

“ $\Leftarrow$ ”: We prove the contrapositive, *i.e.*, if $\langle t_2, A, \langle_H \rangle$ is atomic, then $\langle t_m, A, \langle_H \rangle$ is atomic. Given any trace π_m of $\langle t_m, A, \langle_H \rangle$, it must have a corresponding trace π_2 of $\langle t_2, A, \langle_H \rangle$, where π_m and π_2 differ only on the number of iterations for the loop. Because there must exist a way to swap π_2 into an equivalent serial trace, the events in π_m can be swapped in the same way. Specifically, if there is an

event in π_m to prevent from swapping, an event in π_2 with the same properties (*i.e.*, accesses the same variable, holds the same lock, and observes the same happens-before relation) must exist to prevent π_2 from being serializable. Hence π_m has an equivalent serial trace, *i.e.*, $\langle t_m, A, <_H \rangle$ is atomic. $\square$

5 Optimization: Dynamic Sharing Analysis

To reduce the runtime overhead of monitoring, we restrict monitoring to shared variables. Before an object becomes shared (*i.e.*, escapes from the thread that created it), all events involving it can be ignored. We designed and implemented dynamic sharing analysis to accurately determine the sharing property of each variable. This analysis extends our previous dynamic escape analysis [20] and introduces an additional execution on the same input before the atomicity analysis.

The first execution is used to determine whether each field of every class ever becomes shared during the entire run. Note that we do not construct and analyze hybrid trees during this execution. Each field of a class is processed independently, since some fields might be always accessed by a single thread even if the owner object is shared by multiple threads. Specifically, for each field, if that field in some instance has ever been accessed by multiple threads, the field of the corresponding class is marked as **shared**; otherwise, it is considered **unshared**.

During the second execution with the same input, we keep track of when an object (instead of field) becomes shared while constructing hybrid trees and analyzing atomicity violations. Fields classified as unshared from the first execution are not monitored. When an object becomes shared, all its monitored fields are marked as shared. To indicate whether an object has escaped from its creating thread, we add a boolean instance field to every class with the initial value **false**. We use Java reflection mechanism to dynamically update the field. An object o becomes shared in the following scenarios: (1) o is stored in a static field or a field of a shared object; (2) o is an instance of a thread and the thread is started; (3) o is referenced by a field of another object o' , and o' becomes shared (this leads to cascading sharing); (4) o is passed as an argument to a native method that may cause it to be shared.

The dynamic sharing analysis is based on an assumption that given the same input, the sharings of a variable are the same during different executions, which is true in our experiment of Section 6. The dynamic sharing analysis has improved performance significantly. For example, it reduces the overall runtime by 40% on the benchmarks **Tsp** and **Jigsaw** compared to the executions without the dynamic sharing analysis.

Another optimization is that, for access nodes with the same parent node, we preserve only the first two accesses in the same type (read or write) to each shared variable, because the first two accesses can represent all discarded accesses for checking atomicity. This is justified by Theorem 7.1 in [19].

6 Experiments

We tested our tool on the following programs: **Elevator**, **Tsp**, **Sor**, and **Hedc** from [15], **Jigsaw 2.2.6** from [11], **Apache tomcat 6.0.16**, and **Vector**, **Stack**, and **Hashtable** from JDK 1.4.2.

We performed the experiments on a machine with 1.8 GHz Intel dual-core CPU, 2GiB memory, Windows XP SP3, and Sun JDK 1.6.

Figure 6 compares the running time and results of our hybrid algorithm with the purely dynamic commit node algorithm for conflict-atomicity in [19]. “Base” is the original program’s running time without instrumentation. “Dummy” is the instrumented program’s running time without analyzing atomicity violations (*i.e.*, analysis is not performed after intercepting the events). “Purely Dynamic” is the instrumented program’s running time using the purely dynamic commit node algorithm in [19]. “Hybrid” is the running time of our hybrid algorithm. “Code Coverage” is the coverage of statements in the current execution, which is obtained using an Eclipse plugin EclEmma.

For **Tsp**, HAVE discovers more potential atomicity violations because of speculation. For example, we found that an atomicity violation involves a read on the static field **TspSolver.MinTourLen** in the speculative branch in the method **split_tour** and two writes on the same field in the executed code of the method **set_best** called by the method **recursive_solve**.

For **Jigsaw**, HAVE also reveals more atomicity violations than the purely dynamic approach. HAVE reports that the non-atomic method **perform** in **httpd.sjava** has multiple atomicity violations regarding several fields such as the instance field **LRUNode.next** and the instance field **ResourceStoreImpl.resources**. The previous purely dynamic approach missed this because some field accesses occur in speculatively executed branches.

Program	LOC	Threads	Base	Dummy	Purely Dynamic			Hybrid			Code Coverage
					Time	nAV	NA methods	Time	nAV	NA methods	
Elevator	339	3	0.1	0.2	0.5	18	0-2-0	1	18	0-2-0	89.2%
Tsp	519	3	0.4	3.5	14	28	2-0-0	66.9	54	2-0-0	79.7%
Sor	8253	3	0.8	1.2	1.6	0	0-0-0	3.1	0	0-0-0	74.9%
Hedc	4267	3	0.3	0.4	0.5	7	1-0-0	0.6	7	1-0-0	35.1%
Jigsaw	100846	68	1.4	1.7	2.7	3	1-0-0	118.3	24	2-0-0	8.1%
Tomcat	168297	3	3.3	4.1	7.7	10	0-2-0	38.5	18	1-2-0	13.7%
Vector1.4	383	2	0.1	0.2	0.6	10	4-4-0	0.8	10	4-4-0	69.2%
Stack1.4	418	2	0.1	0.2	0.7	10	3-4-0	0.8	10	3-4-0	85.7%
HashTable1.4	597	2	0.2	0.3	0.6	4	0-4-0	0.9	4	0-4-0	47.5%

Fig. 6. Comparison of the purely dynamic commit node algorithm and the hybrid conflict-edge algorithm in performance and accuracy. The column “nAV” denotes the number of atomicity violations, which are counted based on the places in source code where the events involved in atomicity violations appear. The column “NA-methods” denotes the number of non-atomic methods with the categories being **bug** - **benign** - **false positive**. All times are measured in seconds.

For Tomcat, the static field `StringCache.accessCount` in the method `toString (ByteChunk bc)` of `StringCache.java` has the potential for atomicity violation when at least two threads find `StringCache.bcCache != null` and speculate the `else` branch. The same risk exists for the static field `StringCache.hitCount` in the same method, if both threads fail the condition test before it. We classify this atomicity violation as a bug, because it may cause the statistics to be inaccurate, even though this inaccuracy does not cause other incorrect behavior.

7 Related Work

The most closely related work is our commit-node algorithm in [19], which is purely dynamic. The main contribution of this paper is to extend it to a hybrid algorithm that combines static and dynamic analyses. This paper also presents a new optimization to the algorithm.

Dynamic algorithms to detect atomicity violations can be classified into two categories, based on whether they aim to detect *potential* atomicity violations (*i.e.*, whether any feasible permutation of an observed trace is unserializable), or *actual* atomicity violations (*i.e.*, whether an observed trace is unserializable). The algorithms to detect potential atomicity violations include this paper, Wang and Stoller’s reduction-based, block-based algorithms, commit-node algorithms, [17,20,19], and Flanagan and Freund’s reduction-based algorithm [6], which is similar to Wang and Stoller’s reduction-based algorithm. Xu *et al.* infer computation units (subcomputations that the programmer might expect to be atomic) based on data and control dependencies and report an atomicity violation when an unserializable write by another thread is interleaved in a computation unit [21]. Lu *et al.*’s AVIO system learns access interleaving invariants as indications of programmers’ likely expectations about atomicity and reports an atomicity violation when an observed interleaving violates an access interleaving invariant [12]. Flanagan *et al.* developed a sound and complete atomicity violation detector based on analysis of exact dependencies between operations [8]. Farzan and Madhusudan developed a space-efficient algorithm for detecting atomicity violations [4]. Park and Sen propose a randomized dynamic analysis technique that greatly increases the probability that a special class of potential atomicity violations will manifest as actual atomicity violations [13].

Static analyses have been developed to infer or verify atomicity of code segments, *e.g.*, [16,9,7,18]. Static analysis gives stronger guarantees, because it considers all possible behaviors of a program, but is typically more restrictive or reports more false alarms than dynamic analysis. Model checking can also be used to check atomicity [5,10,4]. Model checking also provides strong guarantees but is feasible only for programs with relatively small state spaces.

Static and dynamic analyses can be combined in various ways for atomicity checking. Agarwal, Sasturkar, Wang, and Stoller used static analysis to reduce the overhead of the reduction-based algorithm [14] and the block-based algorithm [1]. JPPredictor uses static analysis to improve the accuracy of the dependency

relation used in dynamic checking for potential concurrency errors, including atomicity violations [3]. Those techniques, in contrast to ours, do not use speculative execution.

8 Conclusions and Future Work

This paper describes a new approach to enhance dynamic analysis with results from static analysis to make the dynamic analysis more effective at finding subtle atomicity violations, by augmenting the dynamic analysis to consider some of the behavior of unexecuted branches in the program. This is significant because software testing rarely achieves full code coverage in practice.

In our experiments, our hybrid conflict-edge algorithm scales almost as well as our previous dynamic algorithm [19]. Comparing our results in Figure 6 with results for those benchmarks in other papers [19,20,9,6,4,8], our system detects all the atomicity violations detected by the purely dynamic algorithms described in those other papers and, for some benchmarks, detects additional atomicity violations.

Directions for future work include extending the static analysis to be inter-procedural, taking the predicates guarding branches into account, incorporating more sophisticated approaches to identify transaction boundaries, and using a testcase generator to generate inputs that lead to execution of speculative events involved in atomicity violations to verify that they are not false alarms.

References

1. Agarwal, R., Sasturkar, A., Wang, L., Stoller, S.D.: Optimized run-time race detection and atomicity checking using partial discovered types. In: Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE) (November 2005)
2. Agarwal, R., Wang, L., Stoller, S.D.: Detecting potential deadlocks with static analysis and runtime monitoring. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 191–207. Springer, Heidelberg (2006)
3. Chen, F., Serbanuta, T.F., Rosu, G.: jPredictor: a predictive runtime analysis tool for Java. In: Proc. 30th International Conference on Software Engineering (ICSE), pp. 221–230. ACM, New York (2008)
4. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008)
5. Flanagan, C.: Verifying commit-atomicity using model-checking. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 252–266. Springer, Heidelberg (2004)
6. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multi-threaded programs. In: Proc. ACM Symposium on Principles of Programming Languages (POPL), pp. 256–267 (2004)
7. Flanagan, C., Freund, S.N., Qadeer, S.: Exploiting purity for atomicity 31(4) (April 2005)

8. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI). ACM, New York (2008)
9. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2003)
10. Hatcliff, J., Robby, Dwyer, M.B.: Verifying atomicity specifications for concurrent object-oriented software using model-checking. In: Steffen, B., Levi, G. (eds.) VM-CAI 2004. LNCS, vol. 2937, pp. 175–190. Springer, Heidelberg (2004)
11. Jigsaw, version 2.2.4, <http://www.w3c.org>
12. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2006)
13. Park, C.-S., Sen, K.: Randomized active atomicity violation detection in concurrent programs. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (FSE), pp. 135–145. ACM, New York (2008)
14. Sasturkar, A., Agarwal, R., Wang, L., Stoller, S.D.: Automated type-based analysis of data races and atomicity. In: Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP) (June 2005)
15. von Praun, C., Gross, T.R.: Object race detection. SIGPLAN Notices 36(11), 70–82 (2001)
16. von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. Journal of Object Technology 3(6) (June 2004)
17. Wang, L., Stoller, S.D.: Run-time analysis for atomicity. In: Third Workshop on Runtime Verification (RV 2003), vol. 89(2) (2003)
18. Wang, L., Stoller, S.D.: Static analysis of atomicity for programs with non-blocking synchronization. In: Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP) (June 2005)
19. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM Press, New York (2006)
20. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multi-threaded programs. Transactions on Software Engineering 32(2), 93–110 (2006)
21. Xu, M., Bodik, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2005)

Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection

Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi,
and Tien N. Nguyen

Electrical and Computer Engineering Department, Iowa State University, USA

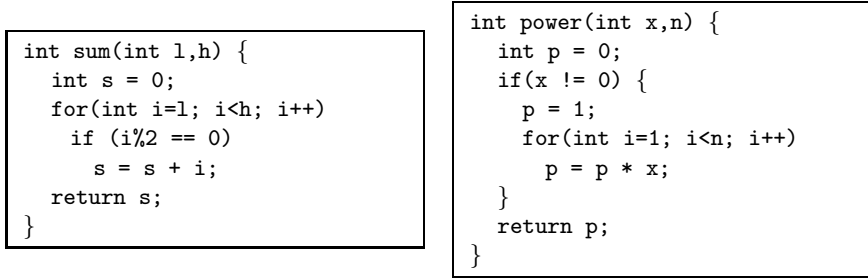
Abstract. Structure-oriented approaches in clone detection have become popular in both code-based and model-based clone detection. However, existing methods for capturing structural information in software artifacts are either too computationally expensive to be efficient or too light-weight to be accurate in clone detection. In this paper, we present Exas, an accurate and efficient structural characteristic feature extraction approach that better approximates and captures the structure within the fragments of artifacts. Exas structural features are the sequences of labels and numbers built from nodes, edges, and paths of various lengths of a graph-based representation. A fragment is characterized by a structural characteristic vector of the occurrence counts of those features. We have applied Exas in building two clone detection tools for source code and models. Our analytic study and empirical evaluation on open-source software show that Exas and its algorithm for computing the characteristic vectors are highly accurate and efficient in clone detection.

1 Introduction

The habit of copy-and-paste in programming is one of the sources for similar fragments of code in many software systems. Such fragments are called *clones*. It is found that cloning also occurs in model-based software development (MBD). Clones create difficulties for software maintenance, for example, changes to a cloned fragment must be carried out in several other places in the codebase.

Many approaches have been proposed to detect clones in traditional software as well as in model-based software. Among them, *structure-oriented* approaches are the most popular and successful in both code-based and model-based clone detection [11,25]. In structure-oriented approaches, software artifacts including source code and models are represented as tree-based and/or graph-based data structures. For example, abstract syntax trees (ASTs) or parse trees are used for source code and attributed, directed graphs are used for program dependence graphs (PDGs), call graphs, and models in MBD.

In those approaches, the common detection process include (1) modeling the artifacts and their fragments (i.e. potential clone parts) in the form of a structure-oriented representation, (2) extracting features of each fragment from its representation, (3) computing the similarity between fragments using a similarity



```

int sum(int l,h) {
    int s = 0;
    for(int i=1; i<h; i++)
        if (i%2 == 0)
            s = s + i;
    return s;
}

int power(int x,n) {
    int p = 0;
    if(x != 0) {
        p = 1;
        for(int i=1; i<n; i++)
            p = p * x;
    }
    return p;
}

```

Fig. 1. Different fragments with similar occurrence-count vectors of single node types

measure on the features, and (4) grouping similar fragments into clone groups. The phases (3) and (4) involve the comparison of the features extracted from the structure-oriented representation of fragments in phase (2). The key feature in those structure-oriented approaches is the internal structure of a (code or model) fragment. However, the methods for comparing tree-based or graph-based structures, such as tree editing distance measurement [1] or graph isomorphism [3], are computationally expensive for scalable and efficient clone detection.

To avoid that high complexity, several light-weight approaches for clone detection have been proposed [4,5,6,7,8]. However, the accuracy is not satisfactory. Methods for extracting structural features in artifacts are still simple. One of the state-of-the-art light-weight approaches, Deckard [8], uses a vector-based approach to extract the structural information in an AST or a parse tree. It proposes to use the occurrence counts of each q -level complete binary subtree in such a tree as characteristic vectors. However, for $q > 1$, this method does *not* work for *graph-based* representations. For $q = 1$ (the case that Deckard tool is implemented), a fragment is characterized by a vector in which each element is the occurrence counts of single AST node types. However, the occurrence counts of only single node types is insufficient to capture well structural information.

The illustrated example in Figure 1 shows two code fragments. They quite differ from each other because they have very different *nesting structures* of program constructs (e.g. a “for” encloses an “if” and vice versa). In addition, in two methods, the *sequential structures* (i.e. the orders) of statements are also different, despite that each contains a declaration, a “for”, an “if”, and a “return” statement. Unfortunately, the above two code fragments have very similar Deckard representation vectors since they have similar numbers of occurrences of single AST node types, e.g., method and variable declarations, “if” and “for” statements, expressions, literals, simple names, etc. Thus, two fragments would be incorrectly detected by Deckard’s vector-based approach.

In brief, existing methods for capturing structural information in software artifacts for clone detection are either too computationally expensive to be efficient or too light-weight to be accurate. Importantly, no light-weight approach has been proposed for graph-based structure. In this paper, we present a novel approach, Exas, for better approximating the structure as a feature of tree-based and graph-based representations in software artifacts. At the same time, Exas

achieves high levels of both accuracy and efficiency. In Exas, features are the sequences of labels and numbers built from nodes, edges, and paths of various lengths in a generic graph-based representation. A *structural* characteristic vector for a fragment contains the occurrence counts of all kinds of its features.

Our analytical study shows that in Exas, for the graph-based representation, isomorphic (sub)graphs have the same structural characteristic vectors. For the graph-based representation, the distance of Exas vectors of two (sub)graphs is bounded by their graph editing distance. That is, if two (sub)graphs are considered clones (i.e. having a small editing distance), their vector distance is small as well. This result also holds for trees and tree editing distance.

We also developed two clone detection tools using Exas: one for code (tree-based representation) and one for Simulink models (graph-based representation). Our empirical study shows that with Exas, the clone detection result is 3-12% more accurate than that of Deckard, the state-of-the-art vector-based approach, with less than a few seconds more in running time. Additionally, the result for model clones is also highly precise. The study shows that our vector computation algorithm is more time-efficient than the canonical labeling, the state-of-the-art method for graph isomorphism. The contributions of this paper include:

1. *Exas*: A novel, accurate and efficient characteristic feature extraction approach for clone detection in structure-based software artifacts. It provides a good approximation of a structure in a program or a model,
2. *An efficient algorithm* to compute Exas vectors,
3. *Two applications of Exas* in two clone detection tools for code and models,
4. *Analytical and empirical studies* of the accuracy and efficiency of Exas.

Section 2 presents Exas approach and an analytic study. Section 3 describes an efficient algorithm to compute Exas vectors. Section 4 discusses the application of Exas in two clone detection tools for code and models. Section 5 presents our empirical evaluation. Related work is in Section 6. Conclusions appear last.

2 Exas Approach

2.1 Structure-Oriented Representation

In our structure-oriented representation approach, a software artifact is modeled as a *labeled, directed graph* (tree is a special case of graph), denoted as $G = (V, E, L)$. V is the set of nodes in which a node represents an element within an artifact. E is the set of edges in which each edge between two nodes models their relationship. L is a function that maps each node/edge to a label that describes its attributes. For example, for ASTs, node types could be used as nodes' labels. For Simulink models, the label of a node could be the type of its corresponding block. Other attributes could also be encoded within labels. In existing clone detection approaches, labels for edges are rarely explored. However, for general applicability, Exas supports the labels for both nodes and edges.

The purpose of clone detection is to find cloned parts in software artifacts. Potential cloned parts in a software artifact are called *fragments*. In our approach,

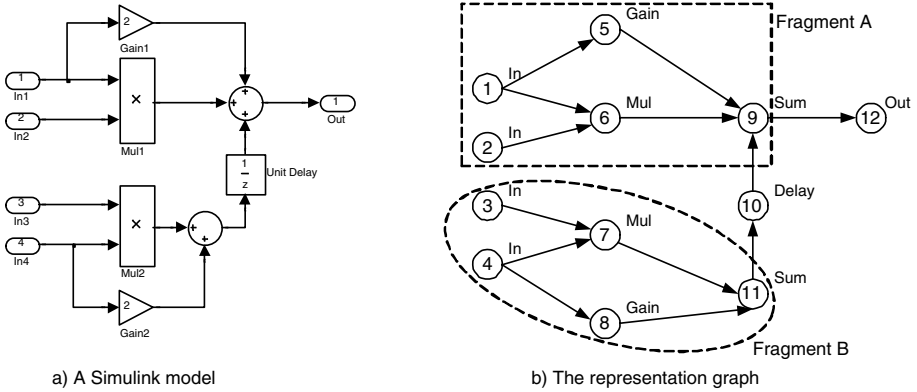


Fig. 2. An example: numbers are the indexes of nodes, texts are their labels

a fragment within a tree-based software artifact is considered as a subtree of the representation tree. For a graph-based software artifact, a fragment is considered as a weakly connected sub-graph in the corresponding representation graph.

Figure 2 shows an illustrated example of a Simulink model, its representation graph and two cloned fragments A and B.

2.2 Structural Feature Selection

Exas focuses on two kinds of *patterns* of structural information of the graph, called (p, q) -node and n -path.

A (p, q) -**node** is a node having p incoming and q outgoing edges. The values of p and q associated to a certain node might be different in different examined fragments. For example, node 9 in Figure 2 is a $(3, 1)$ -node if entire graph is currently considered as a fragment, but is a $(2, 0)$ -node if fragment A is examined.

An n -**path** is a directed path of n nodes, i.e. a sequence of n nodes in which any two consecutive nodes are connected by a directed edge in the graph. A special case is 1-path which contains only one node.

Structural feature of a (p, q) -node is the label of the node along with two numbers p and q . For example, node 6 in fragment A is $(2, 1)$ -node and gives the feature `mul-2-1`. *Structural feature* of an n -path is a sequence of labels of nodes and edges in the path. For example, the 3-path 1-5-9 gives the feature `in-gain-sum`. Table 1 lists all patterns and features extracted from A and B. It shows that both fragments have the same feature set and the same number of each feature. Later, we will show that it holds for all isomorphic fragments.

2.3 Characteristic Vectors

An efficient way to express the property “*having the same or similar features*” is the use of vectors. The characteristic vector of a fragment is the occurrence-count vector of its features. That is, each position in the vector is indexed for

Table 1. Example of Patterns and Features: numbers are the indexes of nodes and texts are features

Pattern	Features of fragment A					Features of fragment B				
1-path	1 in	2 in	5 gain	6 mul	9 sum	4 in	3 in	8 gain	7 mul	11 sum
2-path	1-5 in-gain	1-6 in-mul	2-6 in-mul	6-9 mul-sum	5-9 gain-sum	4-8 in-gain	4-7 in-mul	3-7 in-mul	7-11 mul-sum	8-11 gain-sum
3-path	1-5-9 in-gain-sum		1-6-9 in-mul-sum		2-6-9 in-mul-sum	4-8-11 in-gain-sum		4-7-11 in-mul-sum		3-7-11 in-mul-sum
(p,q)-node	1 in-0-2		2 in-0-1		5 gain-1-1	4 in-0-2		3 in-0-1		8 gain-1-1
(p,q)-node (continued)	6 mul-2-1		9 sum-2-0			7 mul-2-1		11 sum-2-0		

Table 2. Example of Feature Indexing. Based on the occurrence counts of features in fragment A, the vector for A is (2,1,1,1,1,2,1,1,1,2,1,1,1,1,1).

Feature	Index	Counts	Feature	Index	Counts	Feature	Index	Counts	Feature	Index	Counts
in	1	2	in-gain	5	1	in-gain-sum	9	1	gain-1-1	13	1
gain	2	1	in-mul	6	2	in-mul-sum	10	2	mul-2-1	14	1
mul	3	1	gain-sum	7	1	in-0-1	11	1	sum-2-0	15	1
sum	4	1	mul-sum	8	1	in-0-2	12	1			

a feature and the value at that position is the number of occurrences of that feature in the fragment. Table 2 shows the indexes of the features, which are global across all vectors, and their occurrence counts in fragment A.

Two fragments having the same feature sets and occurrence counts will have the same vectors and vice versa. The vector similarity can be measured by an appreciably chosen vector distance such as 1-norm distance.

Note that 1-paths are equivalent to 1-level binary subtrees used in Deckard tool. Therefore, Deckard vector of a fragment is a part of the Exas vector for that fragment. For example, Deckard vector for fragment A would be (2,1,1,1). In other words, Exas uses more features. It implies that the vector distance between Deckard vectors of two fragments is no larger than that of Exas vectors. It also implies that Exas vector distance has better discriminative characteristic, i.e. is more accurate in measuring the fragments' similarity. These are also true when applying for tree structures. For example, in the illustrated example (Section 1), Exas vectors are able to distinguish two code fragments because Exas can better approximate the nesting and sequential structures of program elements.

2.4 Analytical Study

Given two (sub)graphs G and G' . Let V and V' be their vectors, respectively, d be the maximum degree of all nodes of all (sub)graphs, and N be the maximum size of all n -paths.

Lemma 1. *The number of n -paths containing a node is at most $P = \sum_{n=1}^N n \cdot d^{n-1}$ and that of n -paths containing an edge is at most $Q = \sum_{n=2}^N n \cdot d^{n-2}$.*

Due to space limit, we do not provide the proof for Lemma 1 since it is easy to be verified.

For brevity, we call n -paths and (p, q) -nodes *instances*. Let S and S' be the sets of instances of G and G' , respectively. If G is edited to be G' , S' is updated accordingly from S by removing some instances and/or inserting others. Let us call those removed and inserted instances as “*affected instances*”.

Lemma 2. *k graph editing operations affect at most $(2P + 4)k$ instances.*

Proof. We consider four types of graph editing operations: removing an edge, inserting an edge, relabeling a node, and relabeling an edge.

Removing (inserting) an edge removes (inserts) all n -paths containing it and replaces two (p, q) -nodes at its two ends with two new (p, q) -nodes. This replacement affects four instances. Thus, the total number of affected instances is at most $Q + 4$. Relabeling a node replaces its corresponding (p, q) -node and all n -paths containing it with new instances, thus, affects at most $2 + 2P$ instances. Similarly, relabeling an edge affects at most $2Q$ instances since no (p, q) -node is affected. In all cases, the total number of affected instances is at most $2P + 4$. Therefore, k editing operations affect at most $(2P + 4)k$ instances.

Lemma 3. *If there are M affected instances, $\|V - V'\|_1 \leq M$.*

Proof. If an instance is removed (inserted), the occurrence counts of its feature reduce (increase) by one. Since there are M affected instances, V' is obtained from V by the total of M units of such increment and/or decrement. Since $\|V - V'\|_1$ is the total differences of occurrence counts between V and V' , it is at most M .

Two above lemmas imply the following theorem.

Theorem 1. *If graph edit distance of G and G' is k , $\|V - V'\|_1 \leq (2P + 4)k$.*

We could consider two isomorphic graphs as having the editing distance of zero. Therefore, applying Theorem 1, we have the following corollary.

Corollary 1. *If G and G' are isomorphic, they have the same vector, i.e. $V = V'$.*

The results can be applied directly to (sub)trees. However, tree editing distance can be defined in a different set of operations. The following results are for the case in which G and G' are (sub)trees and tree editing operations include relabeling, inserting, and deleting a node.

Lemma 4. *The number of n -paths containing a node in a (sub)tree is at most $R = \sum_{n=1}^N \sum_{i=1}^n d^{i-1}$.*

Lemma 5. *k tree editing operations affect at most $(2R + 3)k$ instances.*

Proof. Relabeling a node affects at most $2R + 2$ instances (see the proof of Lemma 2). Removing a node u , i.e. connecting all its children to its parent v , removes all n -paths containing u , inserts some n -paths containing v , replaces (p, q) -node at v with a new one, and removes (p, q) -node at u . Thus, the total number of affected instances is $2R + 3$. Similar argument is applied for the case of inserting

a node u . Therefore, a single tree editing operation affects at most $(2R + 3)$ instances, thus, k operations affect at most $(2R + 3)k$ instances.

From Lemmas 4 and 5, we have the following theorem.

Theorem 2. *If tree editing distance of G and G' is k , $\|V - V'\|_1 \leq (2R + 3)k$.*

2.5 Implications in Clone Detection

The aforementioned important properties of Exas characteristic vectors imply that they are very useful in the problems involving graph isomorphism or tree/graph similarity, especially in structure-oriented clone detection.

State-of-the-art graph-based clone detection approaches [11,32] require graph-based cloned fragments to be isomorphic. With **Corollary 1**, instead of checking isomorphism of two (sub)graphs, we could compare their Exas characteristic vectors to find cloned (sub)graphs. That corollary guarantees that all clone pairs will be detected. However, it is not a sufficient condition for absolute precision, i.e. two (sub)graphs with the same vectors might be non-isomorphic since nodes and edges which cannot be mapped between two (sub)graphs can make up n -paths or (p, q) -nodes with the same feature. Other criteria should be used along with Exas to increase the precision of detected results.

Theorem 1 shows that our approach is also useful for the problems involving graph editing distances such as similarly matched clone detection in graph-based representations or graph similarity measurement.

Theorem 2 is useful for clone detection approaches based on tree editing distance, i.e. two tree-based fragments are considered clones if their editing distance is smaller than a chosen threshold k . For a set of fragments, we can always find a common value R for any two fragments. Then, the vector distance of any two cloned fragments will be less than $(2R + 3)k$. In other words, the 1-norm distance of Exas characteristic vectors could be used as a necessary condition: to be a cloned pair, two fragments must have the distance of their vectors smaller than a chosen threshold $\delta = (2R + 3)k$. Of course, a small vector distance does not imply a small tree editing distance, i.e. this condition is not sufficient.

In our empirical evaluation (Section 5), the precision of only Exas characteristic vectors is evaluated for both tree-based and graph-based clone detection.

3 Vector Computing Algorithm

In this section, we describe an efficient algorithm to calculate Exas vectors from the structure-oriented representation. The key idea is that *the characteristic vector of a fragment is calculated from the vectors of its sub-fragments*. Of course, a node is the smallest fragment and its vector is calculated directly.

3.1 Key Computation Operation: *incrVector*

The key operation in our algorithm, *incrVector()*, is the computation of the vector for a fragment $g = f + e$ (i.e. g is built from f by extending f with an

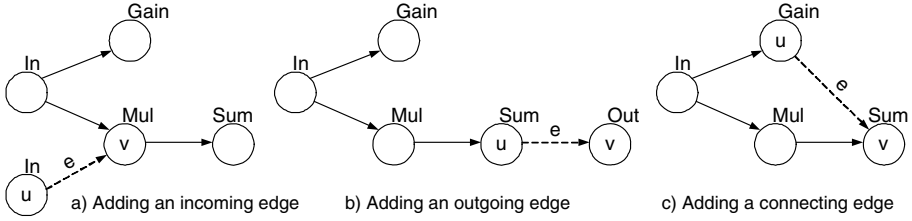


Fig. 3. Three cases of adding an edge to a fragment

edge e), given e and f (along with its vector) as inputs. In brief, the vector of g is derived from that of f by updating it with the occurrences of all new features of g created by the addition of the edge e into f .

Since we consider only weakly connected components as fragments, at least a node of e must belong to f . Let $e = (u, v)$. There are three following cases:

Case 1: incoming-edge, i.e. $u \notin f$ and $v \in f$. In this case, u is a newly added node. New features are created from the 1-path u , the 2-path $u - v$, the new $(0, 1)$ -node at u . The (x, y) -node at v is replaced by the new $(x + 1, y)$ -node because of the new incoming edge. All new n -paths of f will have the first node of u and the second node of v . Therefore, they are generated by adding u to the first of all $(n - 1)$ -paths starting from v . These $(n - 1)$ -paths can be achieved by a depth-first search (DFS) *within fragment* g from node v to the depth of $n - 2$.

Case 2: outgoing-edge, i.e. $u \in f$ and $v \notin f$. The situation is similar. However, new n -paths are generated from $(n - 1)$ -path ending at u , i.e. DFS needs to expand in backward direction. Furthermore, (x, y) -node at v is replaced by a new $(x, y + 1)$ -node.

Case 3: connecting-edge, i.e. both u and v were already in f . In this case, new n -paths are generated by the combination of any i -path ending at u (DFS in backward direction) and an j -path starting from v (DFS in forward direction), for all $i + j = n$. Both (x, y) -nodes at u and v are replaced by new (x', y') -nodes.

Time Complexity and Improvement. Assume that d is the maximum degree of the nodes and N is the maximum length of n -paths of interest. The number of n -paths searched by DFS is $O(d^{N-2})$ in all three cases (in the 3rd case, two DFSs from u and v to level $n - 2$ are sufficient to find all those x -paths and y -paths). This seems to be exponential. However, instead of extracting features from n -paths of all sizes, we just extract features from *short* n -paths, i.e. n -paths having at most N nodes. This gains much efficiency and reduces little precision. In our experiments, in most subject systems, $N = 4$ gives the precision of almost 100% (Section 5). Moreover, in practice, representation graphs are generally not very dense, i.e. d is small. Thus, $O(N \cdot d^{N-2})$ is indeed not very time-consuming.

3.2 Vector Computation for All Fragments in a Graph

Using *incrVector* operation, Exas calculates the vector of any individual fragment by starting from one of its nodes, adding one of its edges, then computing the

vector, and so on. Thus, time complexity of computing vector for a fragment is $O(m.N.d^{N-2})$, with m as the fragment's size, i.e. the number of edges.

For the clone detection problem, the goal is to calculate the vectors for all potential cloned fragments in a graph. Generating all of its sub-graphs and then calculating their vectors as for individual fragments will not take advantage of *incrVector* operation. A more efficient approach is to generate the fragments with the increase in size by adding edge-by-edge and then to calculate the vector for the larger fragment from the vectors of the smaller ones.

However, the number of sub-graphs of a graph is exponential to its size. To increase efficiency, if graph isomorphism is used as a clone condition, we can take advantage of the following fact: to be a clone, a fragment should contain a smaller cloned fragment, i.e. two large isomorphic graphs should contain two smaller isomorphic sub-graphs.

Let C_k be the set of all cloned fragments of size k (i.e. with k edges). Observe that: (1) every fragment of size k can be generated from a fragment of size $k-1$ by adding a relevant edge; and (2) if two fragments of size k are a clone pair (isomorphic), there exists two cloned fragments of size $k-1$ within them, i.e. every clone pair of C_k can be generated from a clone pair of C_{k-1} .

Those facts imply that C_k can be generated from C_{k-1} by following steps: (1) extending all cloned fragments in C_{k-1} by one edge to have a candidate set D_k , (2) calculating vectors for all fragments in D_k by the *incrVector* operation, (3) grouping D_k into clone groups by characteristic vectors (i.e. all fragments in a group must have the same characteristic vectors), and (4) adding only the cloned fragments in D_k into C_k . By gradually generating $C_0, C_1, C_2, \dots, C_k, \dots$, we can find all cloned fragments precisely and completely. Note that, this strategy reduces significantly time complexity for fragment generation and vector computation for sparse graphs. In worst case (such as for a complete graph), time complexity is still exponential. More details can be found in another paper [12].

Vector Calculation for All Fragments in a Tree. For tree-based representations, a fragment is represented by a subtree. Since each node is the root of a subtree, i.e. each fragment corresponds to a node, the generation process is not needed. To compute the vectors for all subtrees in a tree, Exas traverses it in post-order. When a root p of a subtree $T(p)$ is traversed, the vector for $T(p)$ will be calculated as follows. Assume that $c_1, c_2, \dots, c_k$ are the children of p , connecting from p by edges $e_1, e_2, \dots, e_k$. Because of the post-order traversal, the vectors of the sub-trees $T(c_1), T(c_2), \dots, T(c_k)$ are already calculated. Adding edge e_i to subtree $T(c_i)$ using *incrVector* operation gives the vector V_i for each branch. Then, the vector of $T(p)$ is derived from all vectors $V_1, V_2, \dots, V_k$. By this strategy, time for computing vectors for all fragments of a tree is just $O(m.N.d^{N-2})$.

3.3 Vector Indexing and Storing

The potential number of features is huge. For example, if the number of different labels of nodes is L_v and that of edges is L_e , the total number of potential features generated from all n -paths of size no longer than N is $\sum_{n=1}^N L_v^n L_e^{n-1}$. However,

in practice, the actual features encountering in certain graph modeling for a software artifact is much smaller because there are not all combinations of nodes and edges that make sense with respect to the semantics of elements in artifacts.

Our experiment confirmed this fact. We conducted an experiment with WCD-MALIB, a real-world model-based system with 388 nodes and $L_v = 107$ labels ($L_e = 0$). With $N = 4$, the actual features encountered in the whole model is only 381, although the number of all possible features is more than 107^4 .

More importantly, most fragments do not contain all features, especially small fragments. Thus, the *characteristic vectors are often sparse*. To efficiently process sparse characteristic vectors, a hashmap H is used to map between features and their indexes (i.e. their positions in vectors for storing their occurrence counts). H is global and used for all vectors. During vector calculation, if a feature has never been encountered before, it will be added into H with a next (increasing) available index. The vector of a fragment is also stored as a hashmap that maps the index of each feature into its corresponding counting value in the vector.

4 Applications of Exas

To demonstrate the usefulness of our structural feature extraction approach in clone detection, we implemented two tools. The first one, ClemanX, is for tree-based software artifacts. The second one, GemScan, is for graph-based artifacts.

4.1 ClemanX

ClemanX is a clone group management tool based on Cleman framework [9]. One key task of ClemanX is to detect clone groups of code fragments of a software project. Each source file managed by ClemanX is parsed and represented in ClemanX as an AST. The label of each node of the tree is its AST node type such as *class*, *method declaration*, *block*, *for statement*, etc. No label for edges is used. Each fragment is represented by a sub-tree of the AST. ClemanX uses Exas to extract the structural features in fragments as described in Section 3.

ClemanX detects not only exact-matched but also similar-matched code clones. Therefore, clone condition is defined based on the 1-norm distance of characteristic vectors: *two fragments are considered clones if their relative similarity $r = 1 - \frac{2\|v_1 - v_2\|_1}{\|v_1\|_1 + \|v_2\|_1}$ is greater than a chosen threshold* where v_1 and v_2 are their vectors. Relative similarity allows larger clones to have greater differences.

4.2 GemScan

GemScan is a clone detection tool for Simulink models [10]. The transformation of Simulink models into graphs in GemScan is carried out in the same manner as in CloneDetective [11]. Basically, it consists of three tasks: (1) *parsing* the model into a directed graph in which a node represents a block and an edge represents a signal connection, (2) *flattening* the subsystems, and (3) *labeling* nodes/edges with the labels depending on their attributes. The output is a *labeled, directed graph* where the set of nodes V represents Simulink blocks, the set

of directed edges E represents the signal lines, and the labeling function L assigns labels to nodes and edges. Cloned fragments of a model are *weakly connected, non-overlapping, isomorphic sub-graphs* of the representation graph. Fragment generation and vector calculation in GemScan are described in Section 3.

5 Empirical Evaluation

Our experiment mainly focuses on the *accuracy* and *efficiency* of our structural feature extraction approach. We also evaluate the trade-off between those qualities and the limit length of extracted features.

To use ClemanX and GemScan for evaluating Exas, they are configured to use only Exas structural features as a sole criteria for detection. The efficiency is evaluated by the time of fragment generation and vector computation. The accuracy is evaluated by checking the precision of the detected clones, i.e. the ratio between the number of correct clone groups and the total number of clone groups. A group is considered to be correctly detected if all pairs of member fragments in that groups are clones. Clone groups detected by GemScan are checked by *canonical labeling* method, the state-of-the-art technique for checking graph isomorphism. Clone results of ClemanX are checked manually because it detects similar code clones. All experiments are conducted on a computer with AMD Athlon 64 X2 Dual Core 5200+ 2.70GHz, 1.50GB RAM, and Windows XP.

5.1 Clone Detection on Graph-Based Artifacts

GemScan was run on three Simulink systems with different maximum sizes N of used n -paths. For example, $N = 4$ means that only n -paths with at most four nodes are processed. The number of generated fragments, clone groups, used features, and time for fragment generation and vector calculation ($FTime$ in seconds) were reported. We also wanted to compare our approach with canonical labeling, one of the fastest techniques for checking graph isomorphism. Therefore, the canonical labeling module was run to get $Ctime$, time for generating canonical labels of all generated fragments. Those generated canonical labels were then also used to check the correctness of GemScan's detected clone groups.

Table 3 shows the result. $MSize$ is the maximum size of generated fragments. The result shows that our approach is very fast (less than a second) and *thousands times faster* than the canonical labeling method. However, that level of efficiency does not sacrifice much precision. Especially when $N = 4$, the precision reaches 97-100% in almost all cases. Moreover, the precision is increased as the maximum size N of n -paths increases, i.e. extracting longer features achieves higher precision. This implies that Exas is accurate and efficient. Our experiment to evaluate Exas in similar-matched graph-based clone detection is in [12].

5.2 Clone Detection on Tree-Based Artifacts

ClemanX was run on six open-source projects of different sizes from medium to very large ones. Time was measured as before. Table 4 shows the result of

Table 3. Feature extraction time and Precision of GemScan

System	N	Fragment	Feature	MSize	FTime	CTime	Group	Correct	Precision
WCDMALIB	1	11597	139	9	0.84	3135	361	352	98%
388 nodes, 410 edges	2	11415	314	9	1.00	3148	355	355	100%
107 labels	4	11391	382	9	1.10	3151	355	355	100%
Simulink_labs	1	4551	58	13	0.52	1017	739	720	97%
452 nodes, 415 edges	2	4492	181	13	0.64	1080	741	722	97%
39 labels	4	4492	334	13	0.85	1107	729	729	100%
Multiuav	1	7439	78	34	0.91	2000	542	490	90%
471 nodes, 573 edges	2	7316	238	34	0.92	2012	520	496	95%
52 labels	4	7276	465	34	1.00	1966	514	501	97%

Table 4. Feature extraction time of ClemanX

Log4J 1.2.14 (41 kLOC)				jEdit 4.2 (141 kLOC)				Axis 1.4 (227 kLOC)			
N	kFrag.	Feature	FTime	N	kFrag.	Feature	FTime	N	kFrag.	Feature	FTime
1	97	59	2.5	1	379	59	6.1	1	717	59	10.0
2	97	128	2.6	2	379	139	6.4	2	717	134	10.1
4	97	823	2.8	4	379	1388	7.0	4	717	1195	10.4
jFreeChart1.0.6(270kLOC)				JDK6 (3972 kLOC)				Eclipse3.2 (8318 kLOC)			
N	kFrag.	Feature	FTime	N	kFrag.	Feature	FTime	N	kFrag.	Feature	FTime
1	689	59	9.5	1	10,114	59	131	1	28,848	59	337
2	689	133	9.8	2	10,114	145	135	2	28,848	145	351
4	689	1007	10.1	4	10,114	2176	141	4	28,848	2302	361

running ClemanX on those systems. Its columns are similar to those of Table 3. The number of fragments (*kFrag.*) is shown in thousand units, e.g. the number of fragments in Eclipse3.2 is about 28,848,000. Those numbers (also equal to the number of AST nodes) do not change as N is varied because all fragments (i.e. all subtrees in an AST) are processed, regardless of the size of used features.

The result shows that ClemanX performed feature extraction very fast. It can scale up to very large projects with millions lines of code. For example, for Eclipse3.2 with more than 8 millions LOCs and 28 millions fragments, it took only about 6 minutes. Table 4 shows that Exas is very efficient and scalable.

ClemanX detects similar-matched code clone. Because the similarity of code clones is subjective, we have to check the precision of ClemanX manually. The checking was done on 100 randomly selected groups from the clone report for jFreeChart 1.0.6 on each experiment of N . Table 5 shows the result for two different choices for the threshold of similarity (TGroup is the total number of detected clone groups, CGroup is the number of checked groups). As we can see, the precision is very high. In addition, the longer the features are, the higher the precision is. The precision of 100% can be also achieved with $N = 4$.

Remind that for $N = 1$, the feature extraction of our approach is equivalent to Deckard (with $q = 1$). Running on the example in Section 1 and examining closely the fragments from the clone reports, we found that many fragments are wrongly reported as clones when $N = 1$, but are correctly reported as

Table 5. Precision of ClemanX by manual checking

threshold = 0.95					threshold = 0.90				
N	TGroup	CGroup	Correct	Precision	N	TGroup	CGroup	Correct	Precision
1	547	100	97	97%	1	2042	100	88	88%
2	528	100	99	99%	2	2188	100	95	95%
4	532	100	100	100%	4	1904	100	100	100%

```

public List getCategoriesForAxis(CategoryAxis axis){
    List result=new ArrayList();
    int axisIndex=this.domainAxes.indexOf(axis);
    List datasets=datasetsMappedToDomainAxis(axisIndex);
    Iterator iterator=datasets.iterator();
    while (iterator.hasNext()) {
        CategoryDataset dataset=(CategoryDataset)iterator.next();
        for (int i=0; i < dataset.getColumnCount(); i++) {
            Comparable category=dataset.getColumnKey(i);
            if (!result.contains(category))
                result.add(category);
        }
    }
    return result;
}

```

```

public List getCategories(){
    List result=new java.util.ArrayList();
    if (this.subplots != null) {
        Iterator iterator=this.subplots.iterator();
        while (iterator.hasNext()) {
            CategoryPlot plot=(CategoryPlot)iterator.next();
            List more=plot.getCategories();
            Iterator moreIterator=more.iterator();
            while (moreIterator.hasNext()) {
                Comparable category=(Comparable)moreIterator.next();
                if (!result.contains(category))
                    result.add(category);
            }
        }
    }
    return Collections.unmodifiableList(result);
}

```

Fig. 4. A real example where $N=1$ could not capture nesting and sequential structures

non-clones when $N = 2$ or $N = 4$. Figure 4 shows such two fragments in a real case study with different nesting and sequential structures of program elements. In those real examples, our approach is more accurate than Deckard approach. The feature extraction time for $N = 4$ is only a couple tens of seconds longer than that of Deckard (i.e. when $N = 1$).

6 Related Work

Code clone detection approaches can be classified based on the representation of extracted features [2,25]. Text-based approaches [27,28] consider two code fragments as clones if their constituent texts match. Token-based approaches [5,6,29] view a code fragment as a sequence of program tokens. Similar or exactly matched sequences of tokens signify clones. Basit *et al.* [24] use suffix array on the token sequence. No structural features are extracted in text-based and token-based approaches. Tree-based approaches [1,4,7,30,31] represent a code fragment as a subtree in either an AST or a parse tree. Subtrees with similar extracted features are detected as clones. Kontogiannis *et al.* [1] use tree editing distance. Baxter *et al.* [4] compute the similarity between two subtrees in an AST by the ratio between the number of shared nodes and the total. No other structural information is used. In DMS [30], graph transformation and rewriting rules were applied. Wahler *et al.* [31] represent a Java program in XML and detect clones using frequent itemsets mining technique. Evans *et al.* [7] count the number of AST nodes, characters, tokens, LOCs in a fragment. Koschke *et al.* [26] use suffix trees on AST. Fluri *et al.* [23] propose a tree differencing algorithm for ASTs via matching nodes with a minimum edit script.

Deckard [8], a state-of-the-art tree-based approach, extracts characteristic vectors from parse trees by counting q -level binary subtree patterns. Its approach with $q = 1$ is equivalent to Exas with only 1-paths. When $q > 1$, the approach does not work for graphs. To detect semantic clones from Program Dependence Graph (PDG), it maps a subgraph into a forest of subtrees of the program's ASTs, and uses that technique for vector computation. Yang *et al.* also represent a tree by a set of q -level binary subtrees [14]. They transform a general tree to a binary tree and build a profile of all binary subtrees having the depth of q . This approach of q -level binary branches cannot be extended to support graphs.

For graph-based representations, existing approaches are too heavy. Komondor and Horwitz [32] detect code clones in PDGs using subgraph isomorphism. To support similar clones in PDGs, program slicing is applied. In Datrix [33], to compare control or data flow graphs, a variety of software metrics are used including the number of arcs, loops, nodes, exits in a function, independent paths, etc. The state-of-the-art tool for clone detection in graph-based models is CloneDetective [11]. In CloneDetective, the structural feature extraction for clone detection is graph isomorphism. CloneDetective avoids graph isomorphism computation via its heuristic approach. Liu *et al.* [16] detect clones in a UML sequence diagram by representing it as an array of elements, and structural feature is a suffix tree. There has been much research on the methods for similar/exact matching of subgraphs [15]. However, similar to canonical labeling [3] for graph isomorphism, those approaches are too heavy to apply for clone detection.

There are also many approaches to support model evolution including the detection of differences between models [17,18,19], the merging of different models or different versions [19,20], and the management of consistency model changes [21]. The approaches in [17,18] represent a UML diagram as a tree. Those methods share a common strategy in which they try to match nodes from one graph to another via the matching of node labels and only the *local* connectivity of a node to its neighboring nodes. For example, the numbers of parents and/or children nodes of a node are considered. It is equivalent in spirit to our (p, q) -node pattern. Bergmann *et al.* [22] propose an approach for incremental graph-based pattern matching via a model transformation language. Our idea of using p -paths is inspired from the use of q -grams in Information Retrieval [13]. A q -gram refers to a sequence of continuous characters or words in text processing. q -grams are widely used to represent strings and effective in approximate string matching [13].

7 Conclusions

Structure-oriented approaches in clone detection have become popular. However, existing methods for capturing structural information in structure-oriented representation of software artifacts are either too computationally expensive to be efficient or too light-weight to be accurate in clone detection.

In this paper, we introduce Exas, a light-weight structural feature extraction approach that can approximate well the structure of tree-based and graph-based

software fragments. In Exas, the characteristic features are extracted from the patterns of elements of the trees and graphs. The fragments are characterized by their counting vectors of those features. We also provided efficient strategies for fragment generation and algorithms for computing the vectors. We implemented two tools to show the applications and the usefulness of our approach.

Our analytical and empirical studies show that our structural characteristic features are accurate. The detection result is highly precise while it does not lose completeness. Importantly, our approach can be used as an approximation solution for other problems that involve graph isomorphism or tree/graph similarity.

Acknowledgment. This project was funded in part by a grant from the Vietnam Education Foundation (VEF) for the second author. The opinions, findings, and conclusions stated herein are those of the authors and do not necessarily reflect those of VEF. The fifth author is partially supported by the NSF award #0737029 and the Litton Industries Professorship.

References

1. Kontogiannis, K.A., Demori, R., Merlo, E., Galler, M., Bernstein, M.: Pattern matching for clone and concept detection. *Reverse Engineering*, 77–108 (1996)
2. Roy, C., Cordy, J.: Towards a mutation-based automatic framework for evaluating code clone detection tools. In: *C3S2E 2008*, pp. 137–140. ACM, New York (2008)
3. Read, R., Corneil, D.: The graph isomorph disease. *Journal of Graph Theory* 1, 339–363 (1977)
4. Baxter, I.D., Yahin, A., Moura, L., Sant’Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: *ICSM 1998*, p. 368. IEEE CS, Los Alamitos (1998)
5. Li, Z., Lu, S., Myagmar, S.: CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.* 32(3), 176–192 (2006)
6. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilingualistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* 28(7), 654–670 (2002)
7. Evans, W.S., Fraser, C.W., Ma, F.: Clone detection via structural abstraction. In: *WCRE 2007: Working Conference on Reverse Engineering*, pp. 150–159. IEEE CS, Los Alamitos (2007)
8. Jiang, L., Misherghi, G., Su, Z., Glondou, S.: Deckard: scalable and accurate tree-based detection of code clones. In: *ICSE 2007*, pp. 96–105. IEEE CS, Los Alamitos (2007)
9. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Cleman: Comprehensive clone group evolution management. In: *ASE 2008*. IEEE CS, Los Alamitos (2008)
10. The MathWorks Inc. *SIMULINK Model-Based and System-Based Design* (2002)
11. Deissenboeck, F., Hummel, B., Jürgens, E., Schätz, B., Wagner, S., Girard, J.F., Teuchert, S.: Clone detection in automotive model-based development. In: *ICSE 2008*, pp. 603–612. ACM, New York (2008)
12. Pham, N.H., Nguyen, H.A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N.: Complete and Accurate Clone Detection in Graph-based Models. In: *ICSE 2009, International Conference on Software Engineering*. IEEE CS, Los Alamitos (2009)
13. Ukkonen, E.: Approximate string matching with q-grams and maximal matches. Albert-Ludwigs University at Freiburg. Technical report (1991)

14. Yang, R., Kalnis, P., Tung, A.K.H.: Similarity evaluation on tree-structured data. In: SIGMOD 2005: International conference on Management of data (2005)
15. Kuramochi, M., Karypis, G.: Finding frequent patterns in a large sparse graph*. *Data Mining and Knowledge Discovery* 11(3), 243–271 (2005)
16. Liu, H., Ma, Z., Zhang, L., Shao, W.: Detecting duplications in sequence diagrams based on suffix trees. In: APSEC 2006, pp. 269–276. IEEE CS, Los Alamitos (2006)
17. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. *SIGSOFT Softw. Eng. Notes* 28(5), 227–236 (2003)
18. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: ASE 2005, pp. 54–65. ACM, New York (2005)
19. Mehra, A., Grundy, J., Hosking, J.: A generic approach to supporting diagram differencing and merging for collaborative design. In: ASE 2005, pp. 204–213. ACM, New York (2005)
20. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of statecharts specifications. In: ICSE 2007, pp. 54–64. IEEE CS, Los Alamitos (2007)
21. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE 2007, pp. 164–173. ACM, New York (2007)
22. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the viatra model transformation system. In: GRaMoT 2008: Proc. of international workshop on graph and model transformations, pp. 25–32. ACM, New York (2008)
23. Fluri, B., Wuersch, M., Pinzger, M., Gall, H.: Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.* 33(11), 725–743 (2007)
24. Basit, H., Jarzabek, S.: Efficient token based clone detection with flexible tokenization. In: FSE 2007, pp. 513–516. ACM, New York (2007)
25. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.* 33(9), 577–591 (2007)
26. Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: WCRE 2006, pp. 253–262. IEEE CS, Los Alamitos (2006)
27. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences* 26(1), 28–42 (1996)
28. Johnson, J.H.: Identifying redundancy in source code using fingerprints. In: CASCON 1993, pp. 171–183. IBM Press (1993)
29. Baker, B.S.: Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.* 26(5), 1343–1362 (1997)
30. Baxter, I.D., Pidgeon, C., Mehlich, M.: DMS®: Program transformations for practical scalable software evolution. In: ICSE 2004, pp. 625–634. IEEE CS, Los Alamitos (2004)
31. Wahler, V., Seipel, D., Gudenberg, J.W., Fischer, G.: Clone detection in source code by frequent itemset techniques. In: SCAM 2004, pp. 128–135. IEEE CS, Los Alamitos (2004)
32. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 40–56. Springer, Heidelberg (2001)
33. Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics. In: ICSM 1996, p. 244. IEEE CS, Los Alamitos (1996)

Enhanced Property Specification and Verification in BLAST*

Ondřej Šerý

Charles University in Prague
Malostranské náměstí 25
118 00 Prague 1
Czech Republic
ondrej.sery@dsrg.mff.cuni.cz
<http://dsrg.mff.cuni.cz>

Abstract. Model checking tools based on the iterative refinement of predicate abstraction (e.g., SLAM and BLAST) often feature a specification language for expressing complex behavior rules. The source code under verification is instrumented by artificial variables and statements in order to transform the problem of checking such a rule into the problem of program location reachability. This way, the source code gets bloated and additional predicates have to be discovered and tracked during the verification. We suggest that a significant performance improvement can be achieved by tracking state of the behavior rules aside from the source code instead of instrumenting them. We have implemented an extension to BLAST, which accepts a specification language (a simplified version of *behavior protocols*), and checks its validity without modifying the input source code. An experiment with two Linux kernel drivers confirms the performance gain using the extension.

1 Introduction

For the last few years, the explicit state and predicate abstraction based model checking techniques have developed rather independently to each other. Success stories of the predicate abstraction based tools, like SLAM [2] and BLAST [14], earned a lot of both research and industry attention. Basically, these tools create a very coarse existential abstraction (over-approximation) of a system, try to find an error trace (if there is none, the system is safe), decide whether the error trace is a real one (i.e., the system is erroneous) or not, in which case the existential abstraction is refined and the cycle repeats. The technique has very good performance on single-threaded programs even those containing high level of data nondeterminism. On the other hand, these tools typically feature a very limited support for multi-threading, complex data types (e.g., floats and arrays), reasoning about heap objects, and perform poorly on certain inputs (e.g., containing `for` cycles).

* This work was partially supported by the Grant Agency of the Czech Republic project 201/08/0266.

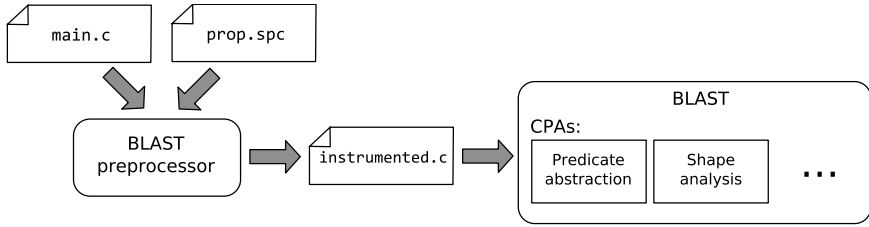


Fig. 1. Architecture of the BLAST model checker

In contrast, the explicit state model checkers, like Java PathFinder [20] and SPIN [15], which are based on explicit representation and exploration of the state space, perform complementarily in many cases. They are typically equipped with optimizations for dealing with multi-threaded programs (e.g., partial order reduction, transactions), and complex data types as well as heap objects are represented explicitly without much trouble. A major obstacle, however, is data nondeterminism, for which the predicate abstraction based tools excel.

In general, this suggests not only that for some inputs one or the other technique is preferred but also that relying on one technique only might not be enough; that mixing these two techniques in a single tool is a promising idea. Quite recently, approaches to mixing explicit state and abstraction based model checking have been published [13,17,8].

In this paper, we apply the idea on the BLAST model checker. We propose an extension for tracking the state of a behavior specification during verification explicitly rather than encoding it into the source program and then using the purely abstraction based verification (as done in BLAST). The original process is depicted on Fig. 1. Before the actual verification, the input source code is instrumented so that the problem of checking a rule is converted into the problem of program location reachability. The resulting program is bloated by a code whose only purpose is to identify the error states. To analyze the instrumented code, additional costly theorem prover calls are necessary. In contrast, our extension tracks the state of the behavior specification explicitly without any modification of the source program and with no unnecessary theorem proving overhead.

BLAST has a specification language [5] for stating the behavior rules. Although the language is very powerful (almost arbitrary C statements can be used) and well suited for simple rules, we argue that it is not very user-friendly for specifying more complex rules concerning function call sequencing and nesting. In such a case, a user has to manually encode the rule into an additional state variable(s) and ensure proper state transitioning, which is very impractical and error prone.

1.1 Goals and Structure of the Paper

The goal of this paper is twofold, (i) to extend BLAST's algorithm for verification of behavior rules by explicit state representation of the rule without modification of the input source code and any additional theorem proving overhead, and (ii)

to allow for specification of behavior rules restricting sequencing and nesting of function calls in a lightweight easy-to-use formalism.

The rest of the paper is structured in the following way. First, we summarize the BLAST's concept of configurable program analysis (Sect. 2), which we employ in our technique on both the formal and the implementation level. Then (Sect. 3), we present the simplified formalism of behavior protocols to be used for behavior specification (Sect. 3.1) along with necessary extensions to the configurable program analysis concept (Sect. 3.2) and a note about the prototype extension of the BLAST model checker (Sect. 3.3). Experimental evaluation of the proposed technique and discussion in the context of the related work (Sect. 5, 4) are followed by list of directions for future research (Sect. 6) and concluding remarks (Sect. 7).

2 Configurable Program Analysis

For the sake of completeness, we summarize the concept of *Configurable Program Analysis* (CPA) as published by the BLAST authors in [7]. The CPA concept stems from *abstract interpretation* [12] and was originally introduced to support a uniform view on model checking and static analysis. Nevertheless, later in Section 3, we will use CPA with advance as a means for plugging the explicit state space of behavior specification into BLAST.

The basic idea is to have multiple CPAs for tracking different kinds of information (e.g., predicates, heap shape) about the program under analysis. Each CPA tracks the information in either a path sensitive or insensitive way. By combining the different CPAs, various configurations of the resulting analysis can be achieved.

Definition 1 (Configurable Program Analysis). *A configurable program analysis is a four-tuple $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$, where D is an abstract domain, $\rightsquigarrow$ is a transfer function, merge is an operator for merging states, and stop is a termination check.*

Informally, D represents the state space of a CPA. It consists of a set of concrete states C , a semi-lattice (with a preorder $\sqsubseteq$ and a join operator $\sqcup$) of abstract states E , and a concretization function relating the abstract and concrete states. For explicit state CPAs featuring no abstraction, the semi-lattice is the trivial flat lattice over the set of concrete states ($E = C \cup \{\top, \perp\}$). The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ contains transitions among the abstract states of D , where G is a set of labels, which contains statements of the program under analysis.

The two operators merge and stop play their role during the state space traversal. The termination check $\text{stop} : E \times 2^E \rightarrow \mathbb{B}$ is used to decide whether a newly discovered state (first parameter) is covered by the already explored states (second parameter). If so, the new state is not analyzed any further. For purposes of this paper, the operator $\text{merge} : E \times E \rightarrow E$ is not of high importance. An intuitive idea that merge is used to merge information from a newly discovered state (first parameter) to each already visited state (second parameter), i.e., merging

Algorithm: *traverseCPA*($\mathbb{D}, e_0$)

Input: a configurable program analysis $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$, an initial abstract state $e_0 \in E$, let E denote the set of abstract states of D

Output: a set of reachable abstract states

Variables: a set *reached* of elements of E , a set *waitlist* of elements of E

```

waitlist := { $e_0$ }
reached := { $e_0$ }
while waitlist  $\neq \emptyset$  do
  pop  $e$  from waitlist
  for each  $e'$  with  $e \rightsquigarrow e'$  do
    for each  $e'' \in$  reached do
      // Combine with already visited abstract states.
       $e_{\text{new}} := \text{merge}(e', e'')$ 
      if  $e_{\text{new}} \neq e''$  then
        waitlist := (waitlist  $\cup \{e_{\text{new}}\}$ )  $\setminus \{e''\}$ 
        reached := (reached  $\cup \{e_{\text{new}}\}$ )  $\setminus \{e''\}$ 
      if  $\neg \text{stop}(e', \text{reached})$  then
        waitlist := waitlist  $\cup \{e'\}$ 
        reached := reached  $\cup \{e'\}$ 
return reached

```

Fig. 2. Algorithm for CPA state space traversal taken from [7]

information from different execution paths, would suffice. A special case of the operators is $\text{stop}^{\text{sep}}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ and $\text{merge}^{\text{sep}}(e, e') = e'$. These correspond to a model checking CPA without merging of information from different execution paths.

In [7], the authors describe a number of CPAs for predicate abstraction, shape analysis, and pointer analysis. They also present a way how to combine more CPAs into a single composite CPA and an algorithm for state space traversal of a CPA. We recapitulate the algorithm in Fig. 2. For further details on CPA, the reader is kindly referred to the original paper.

3 Checking Behavior

Having the CPA concept explained, we first show what kind of behavior specification we are interested in and how it can be encoded using the CPA concept.

3.1 Behavior Specification

The behavior specification used in this paper stems from the formalism of *behavior protocols* [18,1], which comes from the software component world. In a syntax close to regular expressions, a behavior protocol specifies a behavior as a set of finite traces of method calls that are allowed to occur on the component's interfaces. The reason for using behavior protocols (besides our long experience with the formalism) lies in their relative simplicity, which makes them easy to use even for a nonprofessional.

In this paper, a slightly modified definition of behavior protocols, tailored for specification of behavior rules of C code, is used. The basic building block is not a method call on an interface of a component, but a C function call. The simplified syntax and semantics of behavior protocols is as follows.

Definition 2 (Syntax). *A behavior protocol over an alphabet Σ is an expression obtained by a finite number of applications of the following rule. Let a and b be behavior protocols, and $func \in \Sigma$, then all expressions of the form: $NULL$, $func\uparrow$, $func\downarrow$, (a) , a^* , $a; b$, $a + b$, $a \mid b$, $func$, $func\{a\}$ are also behavior protocols.*

The semantics of a behavior protocol is then a set of allowed traces of events $func\uparrow$ and $func\downarrow$, where $func\uparrow$ denotes a function call and $func\downarrow$ denotes return from the call. Distinguishing between the two events allows for precise specification of function nesting.

Definition 3 (Semantics). *The set of traces specified by a behavior protocol p , and denoted as $\mathbb{L}(p)$, is inductively defined as follows:*

Protocol Description Semantics		
$NULL$	Empty prot.	$\mathbb{L}(NULL) = \{\lambda\}$
$func\uparrow$	Func. call	$\mathbb{L}(func\uparrow) = \{func\uparrow\}$
$func\downarrow$	Return	$\mathbb{L}(func\downarrow) = \{func\downarrow\}$
(a)	Parentheses	$\mathbb{L}((a)) = \mathbb{L}(a)$
a^*	Repetition	$\mathbb{L}(a^*) = \{u^n \mid u \in \mathbb{L}(a) \wedge n \in \mathbb{N}_0\}$
$a; b$	Sequence	$\mathbb{L}(a; b) = \{u.v \mid u \in \mathbb{L}(a) \wedge v \in \mathbb{L}(b)\}$
$a + b$	Alternative	$\mathbb{L}(a + b) = \mathbb{L}(a) \cup \mathbb{L}(b)$
$a \mid b$	Parallelism	$\mathbb{L}(a \mid b) = \{u \mid u \text{ is interleaving of } v \in \mathbb{L}(a), w \in \mathbb{L}(b)\}$
$func$	Abbreviation	$\mathbb{L}(f) = \mathbb{L}(func\uparrow; func\downarrow)$
$func\{a\}$	Abbreviation	$\mathbb{L}(f\{a\}) = \mathbb{L}(func\uparrow; a; func\downarrow)$

As an example of a behavior protocol, consider the following usage rule of the SDL graphic library:

`SDL_Init; (SDL_PushEvent + SDL_WaitEvent)* ; SDL_Quit`

The rule states that a call to *SDL_Init* should precede any manipulation with event queues (finite number of calls to *SDL_PushEvent* and *SDL_WaitEvent*) and that the *SDL_Quit* cleanup function should be called afterwards.

Naturally, the set of traces specified by a behavior protocol can be represented by the means of a finite automaton. The transformation follows the standard algorithm of transformation of a regular expression into an automaton [16], straightforwardly extended by the parallel operator $\mid$.

Definition 4. *Let p be a behavior protocol over an alphabet Σ , then $A_p = (S_{A_p}, \Sigma_{\uparrow\downarrow}, \rightarrow_{A_p}, initial_{A_p}, F_{A_p})$, where $\Sigma_{\uparrow\downarrow} = \{func\uparrow, func\downarrow \mid func \in \Sigma\}$, denotes the minimal deterministic finite automaton over the alphabet $\Sigma_{\uparrow\downarrow}$ accepting traces specified by p .*

3.2 Behavior CPA

Decoupling the behavior specification from code into a separate CPA requires one change to the concept of CPA. During state space traversal, different CPAs can add different kinds of information to the states being traversed. In principle, this allows CPAs to affect the shape and the size of the state space to be traversed. However, there is no mechanism that would allow CPA to identify erroneous states. Only the states involving a program location, which is labeled as erroneous (i.e., passed as an input to BLAST) are considered erroneous. Information tracked by individual CPAs is not used for error state detection.

This is because BLAST was originally designed to decide reachability of program locations (marked as erroneous). In theory, this is sufficient for deciding any safety property, as the problem can be always transformed into decision of program location reachability on an accordingly modified program. In practice, however, the necessity to transform all properties into reachability of a program location is prohibitive. It seems more natural to allow the individual CPAs to identify error states based on the information a particular CPA tracks rather than requiring modification of the original source code and thus affecting (obfuscating) input shared by all the CPAs.

For example, to check absence of null pointer dereference errors in a program using BLAST, one can insert an if statement asserting that $p \neq \text{NULL}$ before dereferencing any pointer p . In case of a null pointer, the statement would lead to an error program location. BLAST is then executed to check that the error location is unreachable and thus no null pointer dereference error can occur (this technique was used and documented by others in [6]). In contrast, we argue that such a change of the original program affects all CPAs (mainly the predicate abstraction CPA) and that such an error could be detected by the shape or pointer analysis CPA, which tracks the information concerning heap.

Note that we discuss only “identification” of the possible error states. Once there is a candidate error trace, all the CPAs can contribute to prove the trace infeasible by their means and, if it is a spurious one, get refined to disallow the trace in the future. The point here is that there is no need to bother all the CPAs by the information necessary for identification of the possible errors related to only some of them.

This becomes even more pronounced in the case of behavior specification, whose purpose is only to observe an execution of a program (without altering it in any sense) and to signal any violation. As mentioned above, this problem can also be transformed into the program location reachability by encoding the behavior specification into the program itself (as is done for the BLAST specification language). However, then all CPAs are affected by the additional code, whose purpose lies only in identification of the error states. Namely, the predicate abstraction CPA will be cluttered by artificial predicates. This is costly, because finding and managing additional predicates means additional theorem prover calls.

Therefore, we extended the concept of CPA to allow identification of error states of two kinds. First, a *standard error state* ε_{reach} , is a state which should

never be reached in a correct program; i.e. a program is considered incorrect, if there is a prefix of a concrete path in the program reaching the error state. Second, an *error final state* ε_{final} , is a state which represents an error only for the final state of the model. In other words, a program is considered incorrect, if there is a finite concrete path ending in an error final state. As an example, the null pointer dereference error is a standard error (i.e., it should never happen), while finishing without deallocating all resources is an error final state (i.e., it is alright to have allocated memory during execution but not at its end).

Definition 5 (Configurable program analysis – revisited). A configurable program analysis is a five-tuple $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop}, \text{error})$, where D is an abstract domain, $\rightsquigarrow$ is a transfer function, merge is an operator for merging states, stop is a termination check, and error is an error identifying relation.

Given that E is the set of abstract states of D , then $\text{error} \subseteq E \times \{\varepsilon_{reach}, \varepsilon_{final}\}$ annotates the abstract states which are considered erroneous. An abstract state e implies a whole system error state if $\text{error}(e, \varepsilon_{reach})$ holds. If $\text{error}(e, \varepsilon_{final})$ holds, the whole system error state is implied only if the system state is final; i.e., the system can terminate in its current state.

With the CPA definition extended by error state signalization, we can define a CPA which tracks a single behavior protocol and signal its violation as an error state to the model checker. In turn, the model checker can attempt to avoid the error state by refining abstractions captured by all the CPAs.

Definition 6 (Behavior CPA). Let p be a behavior protocol over an alphabet Σ , then the behavior CPA with respect to p is denoted as $\mathbb{B}(p) = (D_{\mathbb{B}(p)}, \rightsquigarrow_{\mathbb{B}(p)}, \text{merge}_{\mathbb{B}(p)}, \text{stop}_{\mathbb{B}(p)}, \text{error}_{\mathbb{B}(p)})$. $D_{\mathbb{B}(p)}$ is based on the flat lattice over states of the automaton A_p (i.e., $S_{A_p} \cup \{\top, \perp\}$). The transfer relation $\rightsquigarrow_{\mathbb{B}(p)}$ follows the transition function of A_p , extended by a self-transition ($s \xrightarrow{g} s$) for every state s and a control-flow edge g which does not represent any event tracked by the protocol p . More precisely: $s \xrightarrow{g} s'$ iff any of the following holds:

- (i) $s \xrightarrow{g}_{A_p} s'$
- (ii) $s = s'$ and $g \notin \Sigma_{\uparrow\downarrow}$
- (iii) $s' = \perp$ and $g \in \Sigma_{\uparrow\downarrow}$ and $\neg \exists s'' \in S_{A_p} : s \xrightarrow{g}_{A_p} s''$
- (iv) $s = s' = \perp$

The operators $\text{merge}_{\mathbb{B}(p)}$ and $\text{stop}_{\mathbb{B}(p)}$ are chosen to be the simple model checking variants $\text{merge}_{\mathbb{B}(p)} = \text{merge}^{sep}$ and $\text{stop}_{\mathbb{B}(p)} = \text{stop}^{sep}$. Last, the error identifying relation is chosen so that $\text{error}_{\mathbb{B}(p)}(s, \varepsilon_{final})$ iff $s \notin F_{A_p}$ and $\text{error}_{\mathbb{B}(p)}(s, \varepsilon_{reach})$ iff $s = \perp$.

Behavior CPA is straightforwardly derived from the deterministic automaton A_p representing the given behavior protocol p . Those states that do not correspond to a final state of the automaton A_p are identified as error final states. In such states, the behavior protocol expects further activity and does not allow termination of the program yet. Whenever there is an activity not allowed by the protocol (see (iii) of Def. 6), the next state is chosen to be $\perp$, for which $\text{error}_{\mathbb{B}(p)}(\perp, \varepsilon_{reach})$ holds and is therefore a standard error state.

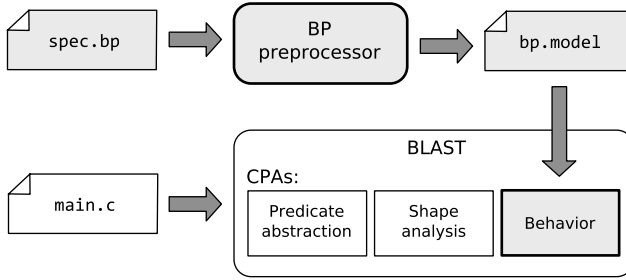


Fig. 3. Architecture of the BLAST extension

3.3 Tool Support

The concept of behavior CPA was implemented as a prototype extension of the BLAST 2.4 release¹. The resulting architecture is depicted in Fig. 3.

The gray emphasized boxes represent the parts of the tool chain newly added to BLAST as a part of this effort. *BP preprocessor* is a simple command line tool for preprocessing the behavior specification. It parses the specification and transforms it into the minimal deterministic automaton. This tool was based on the core library of *dchecker*, the distributed model checker for behavior protocols [19], and is written in Java. As well as the rest of BLAST, the implementation of behavior CPA is written in OCaml and uses the CPA interface as an extension point. The CPA interface itself was modified to allow identification of error states by individual CPAs.

Unfortunately, libraries, which BLAST uses for theorem proving and constraint solving, are available only as Linux binaries. Even though the rest of the implementation is platform independent, the prototype implementation runs also only under Linux, due to these dependencies.

4 Evaluation

Easier usage of behavior protocols for rule specification is, of course, a subjective matter. However, we show the performance improvements in the following experiment. BLAST was used to analyze two Linux 2.6.24 kernel driver files `drivers/char/esp.c` and `drivers/net/znet.c` with and without our extension. There were two behavior rules (i) correct spinlock locking and unlocking, and (ii) correct sequencing of DMA manipulating function calls used in these files (namely the functions `claim_dma_lock`, `release_dma_lock`, `enable_dma`, `disable_dma`, `clear_dma_ff`, `set_dma_mode`, `set_dma_addr`, `set_dma_count`, and `get_dma_residue`). The rules (i) and (ii) specified using both BLAST specification language and behavior protocols are available in the Appendix.

¹ Source code of the prototype implementation along with test files from Sect. 4 are available for download at <http://dsrg.mff.cuni.cz/~sery/blast/>

Table 1. Verification of the spinlock rule

<i>File</i>	drivers/char/esp.c			drivers/net/znet.c		
<i>Test</i>	<i>no hint</i>	<i>hinted</i>	<i>bp</i>	<i>no hint</i>	<i>hinted</i>	<i>bp</i>
<i>Preparation</i>	0.17s	0.17s	0.30s	0.17s	0.17s	0.25s
<i>Verification</i>	3.35s	0.77s	0.26s	1.42s	0.29s	0.14s
<i>Sum</i>	3.52s	0.94s	0.56s	1.59s	0.46s	0.39s

Table 2. Verification of the DMA rule

<i>File</i>	drivers/char/esp.c			drivers/net/znet.c		
<i>Test</i>	<i>no hint</i>	<i>hinted</i>	<i>bp</i>	<i>no hint</i>	<i>hinted</i>	<i>bp</i>
<i>Preparation</i>	0.17s	0.17s	0.30s	0.17s	0.17s	0.30s
<i>Verification</i>	7.07s	1.07s	0.27s	2.55s	0.42s	0.14s
<i>Sum</i>	7.24s	1.24s	0.57s	2.72s	0.59s	0.44s

The verification times² are then summarized in Table 1 and 2 for the rules (i) and (ii), respectively. On each source file and for each rule, three different configurations were executed (columns *no hint*, *hinted*, *bp*). Columns *no hint* and *hinted* contain times for the rule specified in the BLAST specification language. In the *hinted* column, BLAST got the list of predicates suitable for checking the specified rule as a part of the input, while in the *no hint* column, it had to discover the necessary predicates by itself. The *bp* column contains times for the rule specified using behavior protocols and verified using the proposed extension. The difference between *no hint* and *hint* is due to the need for discovery of necessary predicates. However, the predicates can be generated from the specification beforehand. The difference between *hint* and *bp* is due to the need for tracking the additional predicates and coping with the inflated input. The *preparation* row contains times necessary for running the C preprocessor and either code instrumentation or behavior protocol preprocessing. Note also that behavior protocol preprocessing has to be done only once for each rule, while code instrumentation has to be performed after every code modification.

5 Related Work

There is quite a number of software model checkers based on the counter example guided abstraction refinement [11] (CEGAR) and predicate abstraction [3]. Of those, probably the most famous are SLAM [2], BLAST [14], and SATABS [10]. All the three tools analyze programs written in the C programming language.

SLAM is the oldest of the three tools. It features a straightforward implementation of the abstraction refinement loop. In every iteration, a uniform program abstraction is constructed from scratch, which is costly. BLAST enhances the basic idea by constructing the abstraction in memory and refining only the nec-

² All the test were run on a Linux 2.6.24, Pentium 4, 3.0GHz machine.

essary portions of it (this is referred to as *lazy abstraction* [14]). This feature significantly improves performance. Like SLAM, the SATABS tool is a straightforward implementation of the abstraction refinement loop. Unlike SLAM, SATABS uses a SAT solver instead of a theorem prover. This allows for precise reasoning about integers as bit-vectors, including arithmetic overflows.

Although the tools can only decide reachability of a program location, other interesting properties can be transformed into this problem by instrumentation of the input program's source code. For this purpose, both SLAM and BLAST use a special purpose specification language (SLIC [4] and the BLAST specification language [5], respectively). However, as already discussed in Sect. 3.2, the instrumentation results in artificial predicates to be discovered and managed, which implies unnecessary theorem proving overhead.

In contrast, our solution exploits the specific nature of the behavior specification and tracks it explicitly in a separate CPA domain without necessity to alter the input source code. This way, no additional theorem prover calls are necessary. Moreover, we argue that using behavior protocols (which are close to regular expressions known to a majority of software developers) for specifying rules restraining method sequencing and nesting is more convenient than using SLIC or BLAST specification language, where such a rule has to be encoded by hand. For more complex rules, this effectively means transformation into a corresponding automaton and its representation using special state variable.

As both SLIC and BLAST specification language permit using almost arbitrary C code, the expressive power is stronger than the expressive power of behavior protocols used in our work. Therefore, we intend to extend the formalism to cover a bigger set of the real-life rules (see Sect. 6).

There are other works that combine explicit state and abstraction based techniques. In [13], the authors propose an algorithm, SYNERGY, which uses concrete execution in cooperation with predicate abstraction. An abstract counter example is used to guide the concrete execution, while the concrete execution traces are used when refining the abstraction. Another technique is presented in [17], where the explicit state space is traversed but abstraction is employed when deciding whether a current state has been already visited. The resulting under-approximation is then iteratively refined. In [8], a technique using explicit representation of some program variables, while predicate abstraction for other, with possibility of precision adjustment, is proposed and an implementation is done in BLAST.

6 Future Work

One of potential directions for future research is extending the power of the formalism used for specification of behavior rules. Regular language is sufficient for conveniently expressing rules concerning correct function call sequencing and nesting. However, other real-life rules the developers are interested in are related to dynamically created and destroyed program entities (e.g., files and locks). In

other words, developers are often interested in correct sequencing and nesting of function calls that refer to the same instances of these entities.

The idea is similar to *tracematches* [9], which are used to specify incorrect behavior patterns that may relate to individual entities. In contrast, behavior protocols are used for positive specification (e.g., specification of expected behavior) not negative (e.g., specification of forbidden behavior), as we believe that the positive specification is less prone to omissions. Signaling a false error is safer than missing a real one. In order to implement checking capability for such *entity protocols* into the BLAST model checker, the Behavior CPA would have to track the explicit state of an entity protocol separately for every instance of an entity.

7 Conclusion

We believe that combining abstraction and explicit state based model checking is a promising direction for further work in software verification. We have made another step in this direction by extending a predicate abstraction tool BLAST by an explicit state representation of behavior rules specified in a simplified version of the behavior protocols formalism. Thanks to the extension, behavior rules can be verified more efficiently as was shown on an experiment.

A less significant but noteworthy contribution of this paper is the presentation of a novel use case for configurable program analysis, which was originally unanticipated by its authors. We have also proposed changes to this concept in order to allow individual CPAs to identify erroneous states during the verification.

References

1. Adamek, J., Plasil, F.: Component composition errors and update atomicity: static analysis. *Journal of Software Maintenance and Evolution* 17(5), 363–377 (2005)
2. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.* 40(4), 73–85 (2006)
3. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. *SIGPLAN Not.* 36(5), 203–213 (2001)
4. Ball, T., Rajamani, S.K.: Slic: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research (January 2002)
5. Beyer, D., Chlipala, A., Henzinger, T., Jhala, R., Majumdar, R.: The BLAST query language for software verification. In: Giacobazzi, R. (ed.) *SAS 2004*. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
6. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Checking memory safety with blast. In: Cerioli, M. (ed.) *FASE 2005*. LNCS, vol. 3442, pp. 2–18. Springer, Heidelberg (2005)
7. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) *CAV 2007*. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)

8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008). IEEE Computer Society Press, Los Alamitos (2008)
9. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative run-time verification with tracematches. In: Sokolsky, O., Taşiran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 22–37. Springer, Heidelberg (2007)
10. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
11. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 238–252. ACM, New York (1977)
13. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT 2006/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 117–127. ACM, New York (2006)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. SIGPLAN Not. 37(1), 58–70 (2002)
15. Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2003)
16. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley, Reading (2000)
17. Pasareanu, C.S., Pelánek, R., Visser, W.: Predicate abstraction with under-approximation refinement. Logical Methods in Computer Science 3(1) (2007)
18. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Transactions on Software Engineering 28(11), 1056–1076 (2002)
19. Poch, T.: Distributed behavior protocol checker. Master’s thesis, Charles University in Prague, Czech Republic (2006)
20. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering 10(2), 203–232 (2003)

Appendix

The spinlock locking rule prescribes proper alternation of calls to the functions `spin_lock_irqsave` and `spin_unlock_irqrestore`. Listing of this rule specified as behavior protocol follows:

```
( spin_lock_irqsave; spin_unlock_irqrestore )*
```

Specified in the BLAST specification language:

```
global int lockStatus = 0;
event {
```

```

    pattern { spin_lock_irqsave($?, $?); }
    guard { lockStatus == 0 }
    action { lockStatus = 1; }
}
event {
    pattern { spin_unlock_irqrestore($?, $?); }
    guard { lockStatus == 1 }
    action { lockStatus = 0; }
}

```

The DMA rule prescribes specific ordering of calls to the DMA helper functions. For example, other helper functions should be called after the function `claim_dma_lock` and before `release_dma_lock`. According to the source code comments, the `set_dma_XXX` and `get_dma_XXX` functions expect a preceding call to `clear_dma_ff`. All these function are to be called with the specific DMA channel disabled. Specification of the rule using behavior protocol follows:

```

(
  claim_dma_lock;
  (
    (
      disable_dma;
      (
        clear_dma_ff;
        (
          set_dma_mode +
          set_dma_addr +
          set_dma_count +
          get_dma_residue
        ) *
      ) + NULL
    )
  )
  +
  enable_dma
)*;
release_dma_lock
)*

```

The DMA rule in the BLAST specification language:

```

global int dmaStatus = 0;
event {
    pattern { $? = claim_dma_lock(); }
    guard { dmaStatus == 0 }
    action { dmaStatus = 1; }
}
event {
    pattern { disable_dma($?); }
}

```

```
    guard { dmaStatus == 1 }
    action { dmaStatus = 2; }
}
event {
    pattern { enable_dma($?); }
    guard { dmaStatus > 1 }
    action { dmaStatus = 1; }
}
event {
    pattern { clear_dma_ff($?); }
    guard { dmaStatus == 2 }
    action { dmaStatus = 3; }
}
event {
    pattern { set_dma_mode($?, $?); }
    guard { dmaStatus == 3 }
}
event {
    pattern { set_dma_addr($?, $?); }
    guard { dmaStatus == 3 }
}
event {
    pattern { set_dma_count($?, $?); }
    guard { dmaStatus == 3 }
}
event {
    pattern { $? = get_dma_residue($?); }
    guard { dmaStatus == 3 }
}
event {
    pattern { release_dma_lock($?); }
    guard { dmaStatus > 0 }
    action { dmaStatus = 0; }
}
```

Finding Loop Invariants for Programs over Arrays Using a Theorem Prover^{*}

Laura Kovács¹ and Andrei Voronkov²

¹ EPFL

² University of Manchester

Abstract. We present a new method for automatic generation of loop invariants for programs containing arrays. Unlike all previously known methods, our method allows one to generate first-order invariants containing alternations of quantifiers. The method is based on the automatic analysis of the so-called *update predicates* of loops. An update predicate for an array A expresses updates made to A . We observe that many properties of update predicates can be extracted automatically from the loop description and loop properties obtained by other methods such as a simple analysis of counters occurring in the loop, recurrence solving and quantifier elimination over loop variables. We run the theorem prover Vampire on some examples and show that non-trivial loop invariants can be generated.

1 Introduction

Invariants with quantifiers are important for verification and static analysis of programs over arrays due to the unbounded nature of array structures. Such invariants can express relationships among array elements and properties involving arrays and scalar variables of the loop, and thus significantly ease the verification task. Automated discovery of array invariants therefore became a challenging topic, see e.g. [9,20,10,12,3,11,22,13]. Approaches presented in these papers combine inductive reasoning with predicate abstraction, constraint solving and interpolation-based techniques and normally require user guidance in providing necessary templates, assertions or predicates.

In this paper we present a framework for automatically inferring array invariants without any user guidance and without using a priori defined boolean templates or predicates. Moreover, unlike all previously known methods, our method allows one to generate loop invariants containing *quantifier alternations*.

The method is based on the following idea.

1. Given a loop over array and scalar variables, we first try to extract from it various information that can be expressed by first-order formulas. This can be information about scalar variables occurring in the loops, such as precise values of these variables in terms of the loop counter, monotonicity properties of these variables

^{*} This research was partly done in the frame of the Transnational Access Programme at RISC, Johannes Kepler University Linz, supported by the European Commission Framework 6 Programme for Integrated Infrastructures Initiatives under the project SCIENCE (contract No 026133). The first author was supported by the Swiss NSF.

considered as functions of the loop counter and polynomial relations among these variables. For extracting this information we deploy techniques from symbolic computation, such as recurrence solving and quantifier elimination, as presented in [18,14], to perform inductive reasoning over scalar variables.

2. Using the derived loop properties, we then automatically discover first-order properties of the so-called *update predicates* for array variables used in the loop and monotonicity properties for scalar variables. The update predicates describe the positions at which arrays are updated, iterations at which the updates occur and the update values. The first-order information extracted from the loop description can use auxiliary symbols, such as symbols denoting update predicates or loop counters.
3. After having collected the first-order information, we run a saturation theorem prover to eliminate the auxiliary symbols and obtain loop invariants expressed as first-order formulas. When the invariants obtained in this way contain skolem functions, we de-skolemise them into formulas with quantifier alternations.

The main features of the technique presented here are the following.

1. We require no user guidance such as a postcondition or a collection of predicates from which an invariant can be built: all we have is a loop description.
2. We are able to generate automatically complex invariants involving quantifier alternations.

All experiments described in this paper were carried out using two systems: *Aligator* — the package for invariant generation described in [18,14], and the first-order theorem prover *Vampire* [25].

This paper is organised as follows. Section 2 motivates our work with an example. Section 3 presents our program model together with some basic principles of saturation theorem proving. The notion of update predicates is introduced in Section 4 together with properties involving such predicates. Section 5 describes how properties of update predicates and scalar variables are extracted from the loop description. Section 6 presents our method of invariant generation and Section 7 discusses some experiments with the theorem prover *Vampire*. Section 8 focuses on related work. Section 9 concludes the paper with some ideas for future work.

2 Example

In this section we give an example illustrating what kind of loop invariant we would like to generate.

We will use the program of Figure 1 as our running example throughout the paper. The program fills an array B with the non-negative values of a source array A , and an array C with the negative values of A . It is not hard to derive that after n iterations of this loop (assuming $n \leq k$) the value of a is equal to the value of the loop counter n .

```

a := 0; b := 0; c := 0;
while (a ≤ k) do
  if A[a] ≥ 0
    then B[b] := A[a]; b := b + 1;
    else C[c] := A[a]; c := c + 1;
  a := a + 1;
end do

```

Fig. 1. Array partitioning [3]

For example, this property can be derived by the methods of [6,23] or by the recurrence solving part of *Aligator* [14,18]. Moreover, *Aligator* is able to find the linear invariant relation $a = b + c$.

Using light-weight analysis, it is also not hard to see that the values of the variables b and c may not decrease during the loop execution, therefore $c \geq 0$ and $b \geq 0$ are loop invariants. This property can also be extracted by *Aligator* using more complex reasoning involving quantifier elimination techniques [14]. However, such a light-weight analysis would not give us much information about arrays A, B, C and their relationships, apart from the fact that the value of A does not change since A is not updated. For example, one may want to derive the following properties of the loop (n denotes the loop counter).

1. Each of $B[0], \dots, B[b-1]$ is non-negative and equal to one of $A[0], \dots, A[n-1]$.
2. Each of $C[0], \dots, C[c-1]$ is negative and equal to one of $A[0], \dots, A[n-1]$.
3. Each non-negative value in $A[0], \dots, A[n-1]$ is equal to one of $B[0], \dots, B[b-1]$.
4. Each negative value in $A[0], \dots, A[n-1]$ is equal to one of $C[0], \dots, C[c-1]$.
5. For every $p \geq b$, the value of $B[p]$ is equal to its initial value.
6. For every $p \geq c$, the value of $C[p]$ is equal to its initial value.

These properties in fact describe much of the intended function of the loop and can be used to verify properties of programs manipulating arrays in which this loop is embedded. However, the first four of these invariants cannot be obtained by other methods of invariant generation since, when formulated in first-order logic, they require quantifier alternations.

In this paper we introduce a new method that can be used to derive such loop properties automatically using a first-order theorem prover. For example, all of the invariants given above were automatically generated by the theorem prover Vampire.

3 Preliminaries

In this section, we describe our program model and give a brief introduction into saturation theorem proving.

Array and scalar variables. We assume that programs contain *array variables*, denoted by capital-case letters $A, B, C, \dots$, and *scalar variables*, denoted by lower-case letters $a, b, c, \dots$. All notations may have indices. The lower-case letter n will be reserved for *the loop counter*.

Program $\mathcal{P}$. Consider a program $\mathcal{P}$ consisting of a single loop whose body contains assignments, sequencing and conditionals. In the sequel we assume that $\mathcal{P}$ is fixed and give all definitions relative to it. Denote by Var the set of all variables occurring in $\mathcal{P}$, and by Arr the set of all array variables occurring in it.

Expressions. We will use a *language $Expr$ of expressions*. We assume that $Expr$ contains constants (including all integer constants), variables in $Var \cup Arr$, logical variables, some interpreted function symbols, including the standard arithmetical function

symbols $+$, $-$, $\cdot$, and interpreted predicate symbols, including the standard arithmetical predicate symbols $\geq$, $\leq$. We assume that expressions are well-typed with respect to a set of sorts and ι is a sort of integers. *Types* are defined as follows: every sort is a type and types can be built from other types using type constructors $\times$ and $\rightarrow$. We assume that each scalar variable has a sort and each array variable has a type $\iota \rightarrow \tau$, where τ is a sort. If A is an array variable and e an expression, we will write $A[e]$ instead of $A(e)$ to mean the element of A at the position e .

Semantics of Expressions. We assume that every sort has an associated non-empty domain and that the domain associated with ι is the set of integers. Furthermore we assume that interpreted function and predicate symbols of the language are interpreted by functions and relations of appropriate sorts. For example, we assume that $\geq$ is interpreted as the standard inequality on integers.

The semantics of the language *Expr* is defined using the notion of *state*. A state maps each scalar variable of a sort τ into a value in the domain associated with τ , and each array variable A of a type $\iota \rightarrow \tau$ into a function from integers to the domain associated with τ . Note that (for the sake of simplicity) we do not consider arrays as partial functions and do not analyse array bounds. Given a state σ , we can define the value of any expression in this state in the standard way, see e.g. [21].

Semantics of programs. We can define the semantics of programs with assignment, sequencing and conditionals in the standard way, see e.g. [21]. A program of this kind can be considered as a mapping from states to states. A *computation* of a program is a sequence of states.

Extended expressions $v^{(i)}$. Remember that we are dealing with a program $\mathcal{P}$ consisting of a single loop. Suppose that a computation of $\mathcal{P}$ starts at some *initial state* σ_0 . If we ignore the loop condition, then after i iterations of the loop the computation will reach a state σ_i . Let us now extend the notion of expression to capture the state σ_i of program execution obtained after i iterations of the main loop. To this end, we first *fix a program $\mathcal{P}$ and some initial state σ_0* so that the definition is parametrised by this initial state and the program. Let σ_i be the state obtained after i iterations of the computation of $\mathcal{P}$ starting at σ_0 .

For every integer expression i and loop variable v of a type τ , we define a new expression $v^{(i)}$ of the type τ . The value of this expression is defined to be the value of v at the state σ_i . We say that a formula φ , possibly using extended expressions $v^{(i)}$, is *valid for $\mathcal{P}$* , if this formula is true for every computation of $\mathcal{P}$, that is, for all computations starting at an arbitrary initial state.

Example 1. Consider the loop $\mathcal{P}$ whose body consists of a single assignment $c := c + 2$, where c is a scalar variable. Then the formula $(\forall i)(i \geq 0 \implies c^{(i)} = c^{(0)} + 2 \cdot i)$ is valid for $\mathcal{P}$.

Note that $v^{(0)}$ is the value of v in the initial state. We will use expressions $v^{(i)}$ only when we reason about programs or assert their properties. We *will not use these expressions in programs*.

Relativised expressions $i :: e$ **and formulas** $i :: F$. Given an expression e or a formula F , we would like to “relativise” it to an iteration i . The relativised expression and formula will be denoted by $i :: e$ and $i :: F$, respectively. These expressions are only defined when e and F are non-extended expressions, that is, expressions containing no occurrences of subexpressions of the form $v^{(j)}$ for some v and j .

Definition 1. For every expression e , formula F having no occurrences of extended expressions $v^{(j)}$ for any v and j , and every integer expression i , let us define an expression $i :: e$ and a formula $i :: F$ by induction as follows. In the definition below e with indices stands for expressions and F with indices stands for formulas.

1. If v is a loop (scalar or array) variable, then $i :: v \stackrel{\text{def}}{=} v^{(i)}$.
2. $i :: (e_1[e_2]) \stackrel{\text{def}}{=} (i :: e_1)[i :: e_2]$.
3. If e is a constant or a variable (but not an array or a scalar variable) then $i :: e \stackrel{\text{def}}{=} e$.
4. If f is an interpreted function, then $i :: (f(e_1, \dots, e_n)) \stackrel{\text{def}}{=} f(i :: e_1, \dots, i :: e_n)$.
5. If P is a predicate symbol, then $i :: (P(e_1, \dots, e_n)) \stackrel{\text{def}}{=} P(i :: e_1, \dots, i :: e_n)$.
6. $i :: (F_1 \wedge \dots \wedge F_n) \stackrel{\text{def}}{=} i :: F_1 \wedge \dots \wedge i :: F_n$ and similar for other connectives instead of $\wedge$.
7. Let y be a variable not occurring in i . Then $i :: ((\forall y)F) \stackrel{\text{def}}{=} (\forall y)(i :: F)$ and similar for $\exists$ instead of $\forall$.
8. $i :: ((\forall i)F) \stackrel{\text{def}}{=} (\forall i)(F)$ and similar for $\exists$ instead of $\forall$.

For example, if F is the formula $(\forall j)(a = 0 \implies A[b] = c + j)$, where A is an array variable and a, b, c are scalar variables, then $i :: F$ is the formula $(\forall j)(a^{(i)} = 0 \implies A^{(i)}[b^{(i)}] = c^{(i)} + j)$.

Loop body and guarded assignments. For simplicity of presentation we assume that the loop body of $\mathcal{P}$ is represented by an equivalent collection of *guarded assignments*. Let us now define guarded assignments and their semantics. We call a *guarded assignment* an expression

$$G \rightarrow \alpha_1; \dots; \alpha_m, \quad (1)$$

where each of the α_j ’s is an assignment either of the form $v := e$ or of the form $A[e_1] := e_2$, and G is a formula, called the *guard* of this guarded assignment. We assume that each guarded assignment of the form (1) satisfies the following conditions.

1. The left-hand sides of all assignments are syntactically different;
2. If some of the assignments α_j has a form $A[e_1] := e_2$, and some α_k for $k \neq j$ has the form $A[e_3] := e_4$, then in every state satisfying G the expressions e_1 and e_3 have different values.

Furthermore, for every collection of guarded assignments whose guards are $G_1, \dots, G_p$ we assume that

1. for all $j, k \in \{1, \dots, p\}$, if $j \neq k$ then the formula $G_j \wedge G_k$ is unsatisfiable (that is, the guards are mutually exclusive);

2. the formula $G_1 \vee \dots \vee G_p$ is true in all states (that is, for every state at least one of the guards is true in this state).

Let us now define the semantics of collections of guarded assignments satisfying these properties and also briefly discuss how any program can be translated into an equivalent collection of guarded assignments.

Consider a guarded assignment $G \rightarrow e_1 := e'_1; \dots; e_m := e'_m$. The sequence of assignments in a guarded assignment has the semantics of a *simultaneous assignment*

$$(e_1, \dots, e_m) := (e'_1, \dots, e'_m).$$

For example, the guarded assignment $\text{true} \rightarrow x := 0; y := x$ changes any state in which $x = 1$ to a state in which $y = 1$ but not $y = 0$.

One can automatically transform any loop body into an equivalent finite set of guarded assignments [7,21]. In general, such a transformation may result in a set of guarded assignments of size exponential in the size of the loop body, but one can also avoid exponential size by using a slightly different notion of guarded assignment. To satisfy the condition on the left-hand side of guarded assignments one can add extra equalities and inequalities in the guards. For example, the loop body consisting of the sequence of assignments $A[a] := 0; A[b] := 1$ can be transformed into the system consisting of two guarded assignments:

$$\begin{aligned} a \neq b &\rightarrow A[a] := 0; A[b] := 1 \\ a = b &\rightarrow A[b] := 1. \end{aligned}$$

Let us consider an example.

Example 2 (Partition). Consider the partition program of Figure 1. Then the loop body of this program has the following representation in the guarded assignment form:

$$A[a] \geq 0 \rightarrow B[b] := A[a]; b := b + 1; a := a + 1 \quad (2)$$

$$\neg A[a] \geq 0 \rightarrow C[c] := A[a]; c := c + 1; a := a + 1. \quad (3)$$

General setting. Given a loop $\mathcal{P}$ we would like to generate invariants of this loop, that is, find formulas that are true after n iterations of the loop, where n is an arbitrary non-negative integer. These formulas will express the values of loop variables after n iterations in terms of their initial values, i.e. values of loop variables after 0 iterations. To find these formulas, we will write some general properties of loop variables at an arbitrary iteration between 0 and n , using formulas with extended expressions $v^{(i)}$. In the sequel we assume that n is an arbitrary but fixed non-negative integer. We will also use a constant with the same name n in formulas to denote the number n . When we discuss iteration steps, we are only interested in iterations between 0 and $n - 1$. To this end, we introduce a *predicate iter* denoting such iterations. To improve readability, we will normally write $e \in \text{iter}$ instead of $\text{iter}(e)$, where e is an expression. The predicate *iter* has the following definition:

$$(\forall i)(i \in \text{iter} \iff 0 \leq i \wedge i < n). \quad (4)$$

Saturation theorem proving. In our approach to invariant generation, we rely on a saturation prover to infer automatically first-order formulas with equality as quantified invariants from a set of first-order loop properties extracted from loops. We shortly describe the basic ideas of saturation theorem proving, and refer to [24] for more details.

First-order theorem provers using saturation algorithms employ a *superposition calculus*, see e.g. [24]. This calculus works with *clauses* (disjunctions of atomic formulas and their negations) and consists of inference rules that allow one to derive new clauses from existing clauses. To prove a formula F , saturation-based provers convert $\neg F$ to a set of clauses and try to derive the empty clause from this set. If the empty clause is derived, then $\neg F$ is unsatisfiable and so F is a theorem. In saturation-based provers the newly derived clauses are normally consequences of the initial clauses. We use this property to derive invariants instead of establishing unsatisfiability: starting with the set of initial clauses, we derive new clauses from it using a superposition calculus and special kinds of reduction orderings and check if some of the newly derived clauses can be used as invariants.

4 Update Predicates

To make a saturation-based theorem prover find loop invariants we have to extract some properties of the loop and give them to the prover as initial formulas. Our technique for doing this is based on the analysis of updates to arrays. To analyse updates we introduce so-called *update predicates* and some axioms about these predicates. There are also other formulas we extract automatically from the loop description, they are described in the next section.

For each array variable V that is updated in the program we introduce two predicates:

1. $upd_V(i, p)$: at the loop iteration i the array V is updated at the position p ;
2. $upd_V(i, p, v)$: at the loop iteration i the array V is updated at the position p by the value v .

The definition of these update predicates can be extracted automatically from the collection of guarded assignments associated with the loop. For example, guarded assignments (2) and (3) result in the following update predicates for B :

$$upd_B(i, p) \iff i \in iter \wedge p = b^{(i)} \wedge A^{(i)}[a^{(i)}] \geq 0; \quad (5)$$

$$upd_B(i, p, v) \iff i \in iter \wedge p = b^{(i)} \wedge A^{(i)}[a^{(i)}] \geq 0 \wedge v = A[a^{(i)}]. \quad (6)$$

We introduce these update predicates to express the following key properties of array updates:

1. if an array V is never updated at an index p then the final value of $V[p]$ is constant;
2. if an array V is updated at an index p at an iteration i and not updated at any further iteration, then $V[p]$ receives its final value at the iteration i .

These two properties do not depend on the loop. For the array B they are formally expressed as follows.

$$(\forall i) \neg upd_B(i, p) \implies B^{(n)}[p] = B^{(0)}[p]; \quad (7)$$

$$upd_B(i, p, v) \wedge (\forall j > i) \neg upd_B(j, p) \implies B^{(n)}[p] = v. \quad (8)$$

We will refer to these two properties as the *stability property* and the *last update property* for B , respectively.

5 Extracting Loop Properties

In this section we will describe some properties that can be automatically extracted from the loop. Given the loop body, we add all these properties as additional axioms to the theorem prover *Vampire* to help it generate loop invariants.

Constant array. If we have an array A that is never updated in the loop, we can add an axiom $(\forall i)(A^{(i)} = A^{(0)})$. A simpler approach (and the one we adopt here) is to treat such an array A as a constant and simply use A instead of $A^{(i)}$. In our example, A is such an array so we will simply write $A[p]$ instead of $A^{(i)}[p]$.

Monotonicity properties. Let us call a scalar variable v *increasing* if it has the property $(\forall i \in \text{iter})(v^{(i+1)} \geq v^{(i)})$ for all possible computations of the loop. Likewise, a variable is called *decreasing* if it has the property $(\forall i \in \text{iter})(v^{(i+1)} \leq v^{(i)})$ for all possible computations of the loop. A *monotonic variable* is a variable that is either increasing or decreasing.

The monotonicity properties can be discovered either by program analysis tools or by some light-weight analysis. For example, if all assignments to a variable v in the loop have the form $v = v + c$ where c is a non-negative integer constant, then v is obviously increasing. In our example, the variables a , b and c can be identified as increasing using such light-weight analysis.

We can introduce a more fine-grained classification of monotonic variables. A variable v is called *strictly increasing* if it has the property $(\forall i \in \text{iter})(v^{(i+1)} > v^{(i)})$. *Strictly decreasing variables* are defined similarly. In our example the variable a is strictly increasing.

Let us call an increasing integer variable v *dense* if it has the property

$$(\forall i \in \text{iter})(v^{(i+1)} = v^{(i)} \vee v^{(i+1)} = v^{(i)} + 1)$$

for all possible computations of the loop, and similarly for decreasing variables. In our example, the variables a , b , c are all dense.

Let us now formulate properties that we extract from loops automatically for various kinds of monotonic variable. We will only formulate them for increasing variables, leaving the case of decreasing variables to the reader.

1. If a variable v is strictly increasing and dense, then we add the following property:

$$(\forall i)(v^{(i)} = v^{(0)} + i).$$

Note that we do not restrict i in this formula to range over iterations only, as well as we did so in formulas (7) and (8): one can prove that our approach is still sound if we use these more general formulas.

2. If a variable v is strictly increasing but not dense, then we add the following property:

$$(\forall j)(\forall k)(k > j \implies v^{(k)} > v^{(j)}).$$

3. If a variable v is increasing but not strictly increasing, then we add the following property:

$$(\forall j)(\forall k)(k \geq j \implies v^{(k)} \geq v^{(j)}).$$

4. If a variable v is increasing and dense but not strictly increasing, then we add the following property:

$$(\forall j)(\forall k)(k \geq j \implies v^{(j)} + k \geq v^{(k)} + j).$$

Note that, under the monotonicity and density assumptions stated above, the above formula follows from the property $(\forall j)(\forall k)(v^{(k)} \leq v^{(j)} + k - j)$.

In our example the following properties of the monotonic variables a, b, c will be added:

$$\begin{aligned} (\forall i)(a^{(i)} &= a^{(0)} + i). \\ (\forall j)(\forall k)(k \geq j &\implies b^{(k)} \geq b^{(j)}). \\ (\forall j)(\forall k)(k \geq j &\implies c^{(k)} \geq c^{(j)}). \\ (\forall j)(\forall k)(k \geq j &\implies b^{(j)} + k \geq b^{(k)} + j). \\ (\forall j)(\forall k)(k \geq j &\implies c^{(j)} + k \geq c^{(k)} + j). \end{aligned} \tag{9}$$

To describe the other properties extracted from loops we will assume that the loop has the following presentation by guarded assignments:

$$\begin{aligned} G_1 &\rightarrow \alpha_1, \\ &\dots \\ G_m &\rightarrow \alpha_m. \end{aligned} \tag{10}$$

Update properties of monotonic variables. Suppose that x is a monotonic variable. Intuitively, an update property for this variable expresses that, if the variable changes its value, then there exists a program point at which conditions for this change have been enabled. As before, we will only formulate these properties for increasing variables.

Suppose that x is increasing. Further, assume that $U \subseteq \{1, \dots, m\}$ is the set of guarded assignments that may update the value of x , that is, $u \in U$ if and only if α_u contains an assignment to x . Then, if x is dense, we add the following property:

$$(\forall v)(v \geq x^{(0)} \wedge x^{(n)} > v \implies (\exists i \in \text{iter})(\bigvee_{u \in U} (i :: G_u) \wedge x^{(i)} = v)).$$

If x is not dense, then the property is slightly more complex:

$$(\forall v)(v \geq x^{(0)} \wedge x^{(n)} > v \implies (\exists i \in \text{iter})(\bigvee_{u \in U} (i :: G_u) \wedge v \geq x^{(i)} \wedge x^{(i+1)} > v)).$$

For our example the following two axioms will be added:

$$\begin{aligned} (\forall v)(v \geq b^{(0)} \wedge b^{(n)} > v &\implies (\exists i \in \text{iter})(b^{(i)} = v \wedge A[a^{(i)}] \geq 0)); \\ (\forall v)(v \geq c^{(0)} \wedge c^{(n)} > v &\implies (\exists i \in \text{iter})(c^{(i)} = v \wedge \neg A[a^{(i)}] \geq 0)). \end{aligned} \tag{11}$$

Translation of guarded assignments. Suppose that $G \rightarrow e_1 := e'_1; \dots; e_k := e'_k$ is a guarded assignment in the loop representation and $v_1, \dots, v_l$ are all scalar variables of the loop not belonging to $\{e_1, \dots, e_k\}$. Define the *translation* $t(e_j)$ at iteration i of a left-hand side of an assignment as follows: for a scalar variable x , we have $t(x) \stackrel{\text{def}}{=} x^{(i+1)}$, and for an array variable X and expression e we have $t(X[e]) \stackrel{\text{def}}{=} X^{(i+1)}[e^{(i)}]$. Then we add the following axiom:

$$(\forall i \in \text{iter})(i :: G \implies \bigwedge_{j=1, \dots, k} t(e_j) = (i :: e'_j) \wedge \bigwedge_{j=1, \dots, l} v_j^{(i+1)} = v_j^{(i)}).$$

For our running example, we add the following two formulas:

$$\begin{aligned} (\forall i \in \text{iter})(A[a^{(i)}] \geq 0 \implies & B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \wedge \\ & b^{(i+1)} = b^{(i)} + 1 \wedge \\ & c^{(i+1)} = c^{(i)}); \\ (\forall i \in \text{iter})(\neg A[a^{(i)}] \geq 0 \implies & C^{(i+1)}[c^{(i)}] = A[a^{(i)}] \wedge \\ & c^{(i+1)} = c^{(i)} + 1 \wedge \\ & b^{(i+1)} = b^{(i)}). \end{aligned} \tag{12}$$

6 Invariant Generation

Our method of invariant generation works as follows.

1. Given a loop, create its representation by a collection of guarded assignments.
2. Generate loop invariants over scalars using *Aligator*. Note, that any other static analysis tool, e.g. [6,23], can be also used.
3. Extract, using the techniques of Sections 4 and 5, first-order properties of the loop in the logic using expressions $v^{(i)}$. Note that these first-order properties use auxiliary function and predicate symbols that cannot occur in the invariants.
4. Eliminate auxiliary function and predicate symbols by running the saturation theorem prover *Vampire* on the collection of first-order properties of the loop obtained in steps 2 and 3, and finding consequences not using these symbols.

The rest of this section discusses how one can eliminate auxiliary symbols and generate invariants using *Vampire*.

Modern resolution theorem provers [25,26,27] lack several features essential for implementing our procedure for invariant generation. These are

1. reasoning with linear integer arithmetic;
2. procedures for eliminating symbols.

The first problem is very hard (see, e.g. [17] for some results on combining first-order superposition provers and arithmetic). However, one can provide a sound but incomplete axiomatisation of linear integer arithmetic that is sufficient for proving many essential properties of integers. In our experiments we used the following very simple

axiomatisation of the arithmetical relations $>$ and $\geq$, and the successor function s (in our examples we substituted $s(e)$ instead of expressions $e + 1$):

$$\begin{aligned} x \geq y &\iff x > y \vee x = y; \\ x > y &\implies x \neq y; \\ x \geq y \wedge y \geq z &\implies x \geq z; \\ s(x) &> x; \\ x \geq s(y) &\iff x > y. \end{aligned}$$

To solve the second problem (changing a theorem prover to handle symbol elimination) we used the following idea. For every array and scalar variable v that occurs on the left-hand side of an assignment we introduce two new symbols v_0 and v' together with the following axioms: $v^{(0)} = v_0$ and $v^{(n)} = v'$. We call these new symbols *target symbols*. Let us call a clause *useful* if it satisfies the following conditions (by a symbol below we mean a signature symbol, that is, a non-variable).

1. Every symbol in this clause is either a target symbol, or an interpreted symbol or a skolem function introduced by Vampire. We call such symbols *usable* and all other symbols *useless*.
2. The clause contains at least one target symbol or a skolem function.

We are interested in deriving only useful clauses. Indeed, all other clauses either contain symbols, such as update predicates, that cannot occur in invariants and so should be eliminated, or represent valid arithmetical properties and so are irrelevant to the loop.

To this end, we make Vampire use a reduction ordering that makes all useless symbols large in precedence and having a large weight in the Knuth-Bendix ordering used by Vampire¹. We also make Vampire output all generated useful clauses.

If we derive a useful clause containing no skolem functions, then this clause denotes an invariant of the loop for all initial states satisfying the condition $v = v_0$ for all loop variables, after replacing all variables v' by v . For example, from the properties presented in Sections 4 and 5, Vampire derived the following useful clause:

$$\neg x \geq b' \vee B'[x] = B_0[x],$$

which denotes the invariant $\neg x \geq b \vee B[x] = B_0[x]$ and can also be written as

$$(\forall x)(x \geq b \implies B[x] = B_0[x]).$$

If a clause with skolem functions is derived, we can de-skolemise this clause by introducing existential quantifiers. For example, Vampire derived the clause

$$\neg b' > x \vee \neg x \geq 0 \vee A[\$i(x)] = B'[x], \quad (13)$$

where $\$i$ is a skolem function. This clause can be de-skolemised into a *quantified invariant*

$$(\forall x)(b > x \wedge x \geq 0 \implies (\exists y)A[y] = B[x]). \quad (14)$$

¹ Essentially, resolution theorem provers prefer to apply inferences with atoms containing large and heavy symbols and thus eventually remove these atoms from clauses.

However, there are reasons to use clauses with skolem functions directly rather than de-skolemise them. Consider, for example, the following formula derived by *Vampire* for our running example.

$$\neg b' > x \vee \neg x \geq 0 \vee A[\$i(x)] \geq 0. \quad (15)$$

It can be de-skolemised into the invariant

$$(\forall x)(b > x \wedge x \geq 0 \implies (\exists y)A[y] \geq 0). \quad (16)$$

The problem is that (13) and (15) imply the following invariant:

$$(\forall x)(b > x \wedge x \geq 0 \implies (\exists y)(A[y] = B[x] \wedge A[y] \geq 0)),$$

which is not implied by their de-skolemised forms (14) and (16).

7 Experiments with *Vampire*

We made experiments with invariant generation for our running example and also for an example of [22] where an array is filled with 0's at positions from 0 to $n - 1$. In [22] it took 0.01 seconds to generate an invariant for proving the assertion that all elements of the array are zeros. *Vampire* derived this property as a loop invariant in less than 0.01 seconds: more precisely, it derived that all array elements up to the loop counter n are zeros.

For our running example, among all generated invariants we were interested in finding out how fast *Vampire* can derive the following two properties:

1. Array B does not change at positions greater than or equal to the final value of b , that is

$$\forall p(p \geq b' \implies B'[p] = B_0[p]).$$

The corresponding clause was generated in 0.73 seconds.

2. Every value in $\{B[0], \dots, B[b-1]\}$ is a non-negative value in $\{A[0], \dots, A[a-1]\}$:

$$\forall p(b' > p \wedge p \geq 0 \implies B'[p] \geq 0 \wedge \exists k(a' > k \wedge k \geq 0 \wedge A[k] = B'[p])).$$

There are four clauses the conjunction of which imply this formula and which were derived by *Vampire*, one of them is (13). The derivation was found in about 53 seconds.

8 Related Work

Recently, the problem of automatically generating quantified invariant properties for loops with arrays received a considerable attention [9,4,20,16,2,12,11]. Based on the abstract interpretation framework [5], the approaches described in [4,9,11,12,2] use a

set of a priori defined atomic predicates over program variables, from which universally quantified array properties are then inferred. Paper [9] iteratively approximates the strongest boolean combination of a given set of suitable predicates for the loop, until a *fixpoint*, i.e. an invariant, is reached. The approach is based on predicate abstraction with skolem constants for the quantified variables, and implements heuristics for guessing some of the appropriate predicates used further for invariant generation. Iterative computation of invariant predicates is also used in [20]. In [11] a priori fixed templates describing candidate invariant properties are used to generate quantified invariants by *under-approximation* algorithms of logical boolean operators for building abstract interpreters over quantified abstract domains. However, these approaches require a given set of predicates from which invariants can be built; some of them also require user guidance.

Using the combined theory of linear arithmetic and uninterpreted function, [2] presents a *constraint-based* invariant synthesis. The method relies on user-given invariant templates over program variables. Constraints on the unknown parameters of the template invariants are generated based on the inductiveness property of an invariant assertion. Solutions to these constraints are substituted for parameters in the template to derive (universally quantified) invariants. Using counterexample guided abstraction, the method is further extended in [3] to the generation of *path invariants*. A counterexample guided abstraction refinement method is presented also in [16], where *range predicates* are used to characterize properties of array segments between specified bounds. Array invariants are then inferred from the predefined range predicates by interpolation-based techniques. The appropriate range predicates are however supplied manually.

A fundamental difference of our approach compared to these works is that we do not require user-defined templates or a fixed collection of predicates. Our invariants can be arbitrary assertions inferred by a theorem prover from assertions over variables obtained by recurrence solving and quantifier elimination methods and by a light-weight analysis of monotonic variables of loops. The advantage of using general recurrence solving methods together with quantifier elimination is also confirmed by comparing our framework to [19] where loop invariants are inferred by providing predefined solutions for a special subclass of recurrences over scalar variables. Moreover, unlike our approach, the above mentioned methods do not infer automatically polynomial/linear relations among scalar variables as invariants.

Based on the abstract interpretation framework, [10,13] infer universally quantified array invariants. Their approach requires *no user guidance*. The key idea is to partition values used as array indexes into symbolic intervals and use abstract interpretation. Paper [10] infer invariants of a special form essentially involving a single array index, such as $(\forall i \leq n)(A[i] > 0)$. Paper [13] goes further and, using more sophisticated analysis, derives invariants that may involve several arrays in which indexes are obtained from each other by using a “shift” by an expression. An example of such an invariant is $(\forall i \leq n)(A[i] = B[i + e])$, where e is an expression in which i does not occur. These papers do not derive properties with quantifier alternations but [13] treats nested loops. It seems that we can benefit from integrating the approach of [10,13] into ours, both by deriving properties of a single loop iteration and by using their invariants as additional formulas in a theorem prover.

In [22], based on an earlier work [15], a saturation theorem prover together with elimination of symbols is used for generating interpolants and proving loop properties over arrays. Although our approach has much in common with that of [22], there are essential differences. First, we do not require to have a loop property for generating invariants so our approach can be useful for generating properties of loops embedded into large programs. Second, we support richer arithmetic reasoning by using symbolic computation methods. Third, we are able to generate invariants also containing quantifier alternations. Finally, [15,22] require some form of guidance by providing a growing sequence of sets of atoms from which invariants can be built and the efficiency of their analysis may crucially depend on the choice of such a sequence.

9 Conclusion

We showed how quantified loop invariants of programs over arrays can be automatically inferred using a first order theorem prover, reducing the burden of annotating loops with complete invariants. For doing so, we deploy symbolic computation methods to generate numeric invariants of the scalar loop variables and then use update predicates of the loop. Using this information quantified array invariants, including those with alternating quantifiers, are derived with the help of a saturation prover. In particular, our method does not require the user to give a post-condition, a predefined collection of predicates or any other form of human guidance and avoids inductive reasoning. Our initial experimental results on some benchmark examples demonstrate the potential of our method. Modifications of theorem provers are required to carry out large-scale experiments with our method.

Our work was partially inspired by an analysis of loops with arrays occurring in very large programs performed by Thibaud Hottelier, Andrey Rybalchenko and the first author (personal communication): it turned out that many uses of arrays involve either array initialisation, or array copying, either to another array or to itself, or simple iterations over array elements. In other words, typical loops for programs with arrays are not much more complex than the loop of our running example. This made us believe that analysis of counters and other monotonic variables in such loops may provide enough information to generate complex invariants.

Future work. To make our technique widely applicable one needs to extend first-order theorem provers by symbol elimination and generating various classes of clause sets with eliminated symbols: for example, minimal sets so that clauses in this set do not imply each other. Minimality is, obviously, undecidable, so we can instead use some light-weight removal of clauses implied by other clauses.

[22] formulates some results related to symbol elimination in resolution theorem proving. In general, it is interesting to develop a theory for symbol elimination and consequence finding, which is not well-understood in presence of equality.

It is possible that similar techniques can be successfully applied to programs with pointers. To this end one should find out which properties of loops should be extracted automatically to derive interesting invariants for such programs. Another interesting extension would be programs with nested loops: we believe many of them can be handled using the same techniques.

It is also interesting to see how our method can be used for proving loop properties rather than generating them. To this end one can try to embed it into systems for proving program properties, such as [8,1,14].

We also believe that more complex kinds of loop analysis followed by theorem proving would be able to discover non-trivial invariants of logically much more complex loops, such as implementations of quick-sort and union-find algorithms.

We did not treat nested loops or multi-dimensional arrays due to a lack of space, though they can be treated in a similar way, by using two loop counters and presenting arrays as functions of more than one argument and modifying update predicates and their automatically generated properties. One needs extensive experiments to understand the efficiency of the method for these extensions too. We are going to make such experiments after modifying Vampire.

Acknowledgments. We thank Andrey Rybalchenko for motivating discussions, Thibaud Hottelier whose work provided parts of the infrastructure used by us, Rustan Leino whose numerous remarks and careful proofreading helped us to improve the paper considerably, and Sumit Gulwani and Shaz Qadeer for discussions on our method and related work.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Beyer, D., Henzinger, T., Majumdar, R., Rybalchenko, A.: Invariant Synthesis for Combined Theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
3. Beyer, D., Henzinger, T., Majumdar, R., Rybalchenko, A.: Path Invariants. In: Proc. of PLDI (2007)
4. Cousot, P.: Verification by Abstract Interpretation. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 243–268. Springer, Heidelberg (2004)
5. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL, pp. 238–252 (1977)
6. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: Proc. of POPL, pp. 84–96 (1978)
7. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM 18(8), 453–457 (1975)
8. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proc. of PLDI (2002)
9. Flanagan, C., Qadeer, S.: Predicate Abstraction for Software Verification. In: Proc. of POPL, pp. 191–202 (2002)
10. Gopan, D., Reps, T.W., Sagiv, M.: A Framework for Numeric Analysis of Array Operations. In: POPL, pp. 338–350 (2005)
11. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting Abstract Interpreters to Quantified Logical Domains. In: Proc. of POPL, pp. 235–246 (2008)
12. Gulwani, S., Tiwari, A.: An Abstract Domain for Analyzing Heap-Manipulating Low-Level Software. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 379–392. Springer, Heidelberg (2007)

13. Halbwachs, N., Peron, M.: Discovering Properties about Arrays in Simple Programs. In: Proc. of PLDI, pp. 339–348 (2008)
14. Henzinger, T.A., Hottelier, T., Kovacs, L.: Valigator: A Verification Tool with Bound and Invariant Generation. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR. LNCS, vol. 5330, pp. 333–342. Springer, Heidelberg (2008)
15. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
16. Jhala, R., McMillan, K.L.: Array Abstractions from Proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
17. Korovin, K., Voronkov, A.: Integrating Linear Arithmetic into Superposition Calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)
18. Kovacs, L.: Reasoning Algebraically About P-Solvable Loops. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 249–264. Springer, Heidelberg (2008)
19. Kroening, D., Weissenbacher, G.: Counterexamples with Loops for Predicate Abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 152–165. Springer, Heidelberg (2006)
20. Lahiri, S.K., Bryant, R.E.: Indexed Predicate Discovery for Unbounded System Verification. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 135–147. Springer, Heidelberg (2004)
21. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
22. McMillan, K.L.: Quantified Invariant Generation Using an Interpolating Saturation Prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
23. Miné, A.: The Octagon Abstract Domain. In: Proc. of WCRE, pp. 310–319 (2001)
24. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 7, vol. 1, pp. 371–443. Elsevier, Amsterdam (2001)
25. Riazanov, A., Voronkov, A.: The Design and Implementation of Vampire. AI Communications 15(2-3), 91–110 (2002)
26. Schulz, S.: System description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 223–228. Springer, Heidelberg (2004)
27. Weidenbach, C., Schmidt, R.A., Hillenbrand, T., Rusev, R., Topic, D.: System description: SPASS version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007)

Author Index

- Acharya, Mithun 370
Aguirre, Nazareno M. 155
Al-Kofahi, Jafar M. 440
AlTurki, Musab 262
Androutsopoulos, Kelly 216
Autili, Marco 124
- Benedetto, Paolo Di 124
Boronat, Artur 18
Bottoni, Paolo 278
Brucker, Achim D. 417
- Calinescu, Radu 421
Chander, Ajay 262
Chatterjee, Shaunak 385
Chen, Qichang 425
Chockler, Hana 201
Clark, Allan 1
Clark, David 216
- Daniel, Brett 171
Dhurjati, Dinakar 262
- Ehrig, Hartmut 325
Ermel, Claudia 325
- Farchi, Eitan 201
Frias, Marcelo F. 155
Fülleborn, Alexander 294
- Giannakopoulou, Dimitra 94
Gilmore, Stephen 1
Godlin, Benny 201
Gray, Kathryn E. 186
Guerra, Esther 278
- Halbwachs, Nicolas 140
Harman, Mark 216
Haugen, Øystein 34
Heckel, Reiko 18
Heisel, Maritta 294
Hermann, Frank 325
Huisman, Marieke 340
- Inamura, Hiroshi 262
Inverardi, Paola 124
- Jagannath, Vilas 171
Jahier, Erwan 140
Julien, Christine 401
Jurack, Stefan 49
Juvekar, Sudeep 385
- Kosiuczenko, Piotr 246
Kovács, Laura 470
Kugler, Hillel 79
Kwiatkowska, Marta 421
- Lambers, Leen 49
Lamo, Yngve 64
Lara, Juan de 278
Lee, Yun Young 171
Leino, K. Rustan M. 231
Li, Zheng 216
- Maibaum, Thomas S.E. 155
Marinov, Darko 171
Meffert, Klaus 294
Mehner, Katharina 49
Meseguer, José 18
Middelkoop, Ronald 231
Møller-Pedersen, Birger 34
Moscato, Mariano M. 155
Mycroft, Alan 186
- Narasamdya, Iman 309
Nguyen, Hoan Anh 440
Nguyen, Tien N. 440
Nguyen, Tung Thanh 440
Novikov, Sergey 201
- Oldevik, Jon 34
- Păsăreanu, Corina S. 94
Payton, Jamie 401
Périn, Michaël 309
Pham, Nam H. 440
Plock, Cory 79
Pnueli, Amir 79
- Rajamani, Vasanth 401
Raymond, Pascal 140
Roman, Gruia-Catalin 401

- Rossini, Alessandro 64
Rutle, Adrian 64
Sen, Koushik 385
Šerý, Ondřej 456
Stoller, Scott D. 425
Sumner, William N. 355
Taentzer, Gabriele 49
Tamalet, Alejandro 340
Tratt, Laurence 216
Tribastone, Mirco 1
Voronkov, Andrei 470
Wachsmuth, Guido 109
Wang, Liqiang 425
Wassying, Alan 155
Wierse, Gerd 49
Wolff, Burkhard 417
Wolter, Uwe 64
Xie, Tao 370
Yang, Zijiang 425
Yu, Dachuan 262
Zhang, Xiangyu 355