Alfonso Pierantonio
Salvador Trujillo (Eds.)

LNCS 10890

Modelling Foundations and Applications

14th European Conference, ECMFA 2018
Held as Part of STAF 2018
Toulouse, France, June 26–28, 2018, Proceedings

Springer

Lecture Notes in Computer Science 10890

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

Editorial Board

David Hutchison
Lancaster University, Lancaster, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Zurich, Switzerland
John C. Mitchell
Stanford University, Stanford, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
C. Pandu Rangan
Indian Institute of Technology Madras, Chennai, India
Bernhard Steffen
TU Dortmund University, Dortmund, Germany
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbrücken, Germany

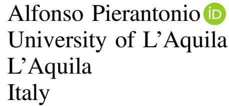
More information about this series at http://www.springer.com/series/7408

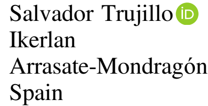
Alfonso Pierantonio · Salvador Trujillo (Eds.)

Modelling Foundations and Applications

14th European Conference, ECMFA 2018 Held as Part of STAF 2018 Toulouse, France, June 26–28, 2018 Proceedings

Springer

Editors

Alfonso Pierantonio
University of L'Aquila
L'Aquila
Italy


Salvador Trujillo
Ikerlan
Arrasate-Mondragón
Spain

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-92996-5 ISBN 978-3-319-92997-2 (eBook)
https://doi.org/10.1007/978-3-319-92997-2

Library of Congress Control Number: 2018944413

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer International Publishing AG, part of Springer Nature 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

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

Foreword

Software Technologies: Applications and Foundations (STAF) is a federation of leading conferences on software technologies. It provides a loose umbrella organization with a Steering Committee that ensures continuity. The STAF federated event takes place annually. The participating conferences and workshops may vary from year to year, but they all focus on foundational and practical advances in software technology. The conferences address all aspects of software technology, from object-oriented design, testing, mathematical approaches to modeling and verification, transformation, model-driven engineering, aspect-oriented techniques, and tools. STAF was created in 2013 as a follow-up to the TOOLS conference series that played a key role in the deployment of object-oriented technologies. TOOLS was created in 1988 by Jean Bézivin and Bertrand Meyer and STAF 2018 can be considered its 30th birthday. STAF 2018 took place in Toulouse, France, during June 25–29, 2018, and hosted: five conferences, ECMFA 2018, ICGT 2018, ICMT 2018, SEFM 2018, TAP 2018, and the Transformation Tool Contest TTC 2018; eight workshops and associated events. STAF 2018 featured seven internationally renowned keynote speakers, welcomed participants from all around the world and had the pleasure to host a talk by the founders of the TOOLS conference, Jean Bézivin and Bertrand Meyer. The STAF 2018 Organizing Committee would like to thank (a) all participants for submitting to and attending the event, (b) the Program Committees and Steering Committees of all the individual conferences and satellite events for their hard work, (c) the keynote speakers for their thoughtful, insightful, and inspiring talks, and (d) the Ecole Nationale Supérieure d'Electrotechnique, Electronique, Hydraulique et Télécommunications (ENSEEIHT), the Institut National Polytechnique de Toulouse (Toulouse INP), the Institut de Recherche en Informatique de Toulouse (IRIT), the region Occitanie, and all sponsors for their support. A special thanks goes to all the members of the Software and System Reliability department of the IRIT laboratory and the members of the INP-Act SAIC, coping with all the foreseen and unforeseen work to prepare a memorable event.

June 2018

Marc Pantel
Jean-Michel Bruel

Preface

The 14th European Conference on Modelling Foundations and Applications (ECMFA 2018) was organized by the ENSEEIHT – Ecole Nationale Supérieure d'Ingénieurs en Electrotechnique, Electronique, Informatique, Hydraulique et Télécommunications — and held in Toulouse during June 26–28, 2018, as part of the Software Technologies: Applications and Foundations (STAF) federation of conferences.

ECMFA is the key European conference aiming at advancing the state of knowledge and fostering the industrial application of model-based engineering (MBE) and related methods. MBE is an approach to software engineering that sets a primary focus on leveraging high-level and suitable abstractions (models) to enable computer-based automation and advanced analyses. Its focus is on engaging the key figures of research and industry in a dialog that will result in stronger and more effective practical application of MBE, hence producing more reliable software based on state-of-the-art research results.

In this edition, the Program Committee received 45 submissions. Each submission was reviewed by at least three Program Committee members. The committee decided to accept 18 papers, 12 papers for the Foundations Track and 6 papers for the Applications Track. Papers on a wide range of MBE aspects were accepted, including topics such as (bidirectional and unidirectional) model transformations, model management, re-engineering, modelling environments, verification and validation, and domain-specific modelling with respect to business processes, automotive software, and safety-critical software.

We thank Perdita Stevens and Mélanie Bats for their inspiring keynote talks at ECMFA 2018. Furthermore, we are grateful to all the Program Committee members for providing their expertise while reviewing the submitted papers. Their helpful and constructive feedback to all authors is most appreciated. We thank the ECMFA Steering Committee in the person of Richard Paige for his priceless advice and the organization chairs, Marc Pantel and Jean-Michel Bruel, for their prompt and continuous support. Last but not least, we are grateful to all authors who submitted papers to ECMFA 2018.

June 2018

Alfonso Pierantonio
Salvador Trujillo

Organization

Program Committee

Name	Affiliation
Shaukat Ali	Simula Research Laboratory, Norway
Anthony Anjorin	Paderborn University, Germany
Alessandra Bagnato	Softeam, France
Marco Brambilla	Politecnico di Milano, Italy
Jean-Michel Bruel	IRIT, France
Jordi Cabot	ICREA - UOC (Internet interdisciplinary institute), Spain
Carlos Cetina	Universidad San Jorge, Spain
Marsha Chechik	University of Toronto, Canada
Antonio Cicchetti	Mälardalen University, Sweden
Federico Ciccozzi	Mälardalen University, Sweden
Benoit Combermale	IRIT, University of Toulouse, France
Zinovy Diskin	McMaster University/University of Waterloo, Canada
Mahmoud El Hamlaoui	ENSIAS, Rabat IT Center, UM5R, Morocco
Romina Eramo	University of L'Aquila, Italy
Roberto E. Lopez Herrejon	Ecole de Technologie Superieure, Canada
Michalis Famelis	University of Montreal, Canada
Jacob Geisel	OBEO, France
Sebastien Gerard	CEA LIST, France
Martin Gogolla	University of Bremen, Germany
Jeff Gray	University of Alabama, USA
Joel Greenyer	Leibniz Universität Hannover, Germany
Esther Guerra	Universidad Autónoma de Madrid, Spain
Oystein Haugen	Østfold University College, Norway
Regina Hebig	Chalmers — Gothenburg University, Sweden
Reiko Heckel	University of Leicester, UK
Ludovico Iovino	Gran Sasso Science Institute, Italy
Ekkart Kindler	Technical University of Denmark, Sweden
Dimitris Kolovos	University of York, UK
Thomas Kuehne	Victoria University of Wellington, New Zealand
Timothy Lethbridge	University of Ottawa, Canada
Ralf Lämmel	Universität Koblenz-Landau, Germany
Richard Paige	University of York, UK
Alfonso Pierantonio	University of L'Aquila, Italy
Daniel Ratiu	Siemens Corporate Technology, Germany
Bernhard Rumpe	RWTH Aachen University, Germany
Houari Sahraoui	University of Montreal, Canada
Eugene Syriani	University of Montreal, Canada

Ramin Tavakoli Kolagari	Nuremberg Institute of Technology, Germany
Salvador Trujillo	Ikerlan Research Centre, Spain
Massimo Tisi	Inria, France
Juha-Pekka Tolvanen	MetaCase, Finland
Antonio Vallecillo	Universidad de Málaga, Spain
Mark Van Den Brand	Eindhoven University of Technology, The Netherlands
Hans Vangheluwe	University of Antwerp, Belgium and McGill University, Belgium and Canada
Daniel Varro	Budapest University of Technology and Economics, Hungary
Manuel Wimmer	Vienna University of Technology, Austria
Tao Yue	Simula Research Laboratory and University of Oslo, Norway
Steffen Zschaler	King's College London, UK

Additional Reviewers

Babur, Önder
Barmpis, Konstantinos
Batot, Edouard
Bousse, Erwan
Butting, Arvid
Cetina, Carlos
Doan, Khanh-Hoang
Geisel, Jacob
Gómez, Abel
Haubrich, Olga
Heinz, Marcel
Hilken, Frank
Hoyos Rodriguez, Horacio
Härtel, Johannes
Ircio, Josu
König, Harald
Leduc, Manuel
Leroy, Dorian
Lopez-Herrejon, Roberto-Erick
Ma, Tao
Markthaler, Matthias
Martínez, Salvador
Netz, Lukas
Peralta, Goiuri
Pradhan, Dipesh
Safdar, Safdar Aqeel
Safilian, Aliakbar
Schmalzing, David
Semeráth, Oszkár
Shumeiko, Igor
Zhang, Huihui

Abstracts of Invited Keynotes

Is Bidirectionality Important?

Perdita Stevens

School of Informatics, University of Edinburgh, Edinburgh, UK
perdita.stevens@ed.ac.uk

Abstract. Bidirectional transformations maintain consistency between information sources, such as different models of the same software system. In certain settings this is undeniably convenient – but is it important? I will argue that developing our ability to engineer dependable bidirectional transformations is likely to be crucial to our ability to meet the demand for software in coming decades. I will discuss some of the work that has been done so far, including some I've had a hand in, and what challenges remain.

The Future of Modeling Tools

Mélanie Bats

Obeo, Colomiers, France
melanie.bats@obeo.fr

Until now, modeling tools have relied mostly on native technologies, and consequently the graphical modelers based on it are desktop applications. Today there are different initiatives to bring graphical modeling tools up to the cloud. The journey of building modeling tools has never been as exciting as it is right now. What would be the advantages of a cloud based modeling tool? What changes this requires in the architecture of such tools? At Obeo, we have been working on modeling tools in general, and on Eclipse Sirius in particular, for a long time now. During this session, we will discuss the future of development tooling, we will briefly review the progress made over the last years and where the open source community is moving towards. We will demonstrate the different levels of integration we currently have, in particular how we leverage projects like Sprotty, ELK, Theia and Che to move diagrams into the browser. We will discuss how "Server Protocols" allow to bring our tools on different platforms and environments and to run them on the cloud or locally. We will also present the Graphical Server Protocol initiative, which will define a platform-agnostic protocol between a diagram editor in the browser and a graphical server that manages the corresponding models in the cloud. Through this talk you will discover what could be the future of Eclipse Modeling on the web, discover how you can bring your own tools to the cloud thanks to Sirius, and participate in this exciting endeavour!

Contents

Is Bidirectionality Important? 1
Perdita Stevens

Towards Automatic Generation of UML Profile Graphical Editors for Papyrus 12
Athanasios Zolotas, Ran Wei, Simos Gerasimou, Horacio Hoyos Rodriguez, Dimitrios S. Kolovos, and Richard F. Paige

Optimising OCL Synthesized Code 28
Jesús Sánchez Cuadrado

Expressing Measurement Uncertainty in OCL/UML Datatypes 46
Manuel F. Bertoa, Nathalie Moreno, Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo

On the Influence of Metamodel Design to Analyses and Transformations. . . . 63
Georg Hinkel and Erik Burger

Automatic Transformation Co-evolution Using Traceability Models and Graph Transformation 80
Adrian Rutle, Ludovico Iovino, Harald König, and Zinovy Diskin

Bidirectional Method Patterns for Language Editor Migration 97
Enes Yigitbas, Anthony Anjorin, Erhan Leblebici, and Marvin Grieger

Parallel Model Validation with Epsilon 115
Sina Madani, Dimitrios S. Kolovos, and Richard F. Paige

SysML Models Verification and Validation in an Industrial Context: Challenges and Experimentation 132
Ronan Baduel, Mohammad Chami, Jean-Michel Bruel, and Iulian Ober

Property-Aware Unit Testing of UML-RT Models in the Context of MDE. . . 147
Reza Ahmadi, Nicolas Hili, and Juergen Dingel

MAPLE: An Integrated Environment for Process Modelling and Enactment for NFV Systems 164
Sadaf Mustafiz, Guillaume Dupont, Ferhat Khendek, and Maria Toeroe

Detecting Conflicts Between Data-Minimization and Security Requirements in Business Process Models 179
Qusai Ramadan, Daniel Strüber, Mattia Salnitri, Volker Riediger, and Jan Jürjens

Life Sciences-Inspired Test Case Similarity Measures for Search-Based, FSM-Based Software Testing 199
Nesa Asoudeh and Yvan Labiche

EMF Patterns of Usage on GitHub 216
Johannes Härtel, Marcel Heinz, and Ralf Lämmel

Towards Efficient Loading of Change-Based Models 235
Alfa Yohannis, Horacio Hoyos Rodriguez, Fiona Polack, and Dimitris Kolovos

Towards a Framework for Writing Executable Natural Language Rules 251
Konstantinos Barmpis, Dimitrios Kolovos, and Justin Hingorani

Model-Driven Re-engineering of a Pressure Sensing System: An Experience Report 264
Atif Mashkoor, Felix Kossak, Miklós Biró, and Alexander Egyed

Modeling AUTOSAR Implementations in Simulink 279
Jian Chen, Manar H. Alalfi, Thomas R. Dean, and S. Ramesh

Trace Comprehension Operators for Executable DSLs 293
Dorian Leroy, Erwan Bousse, Anaël Megna, Benoit Combemale, and Manuel Wimmer

Author Index 311

Is Bidirectionality Important?

Perdita Stevens[(✉)]

School of Informatics, University of Edinburgh, Edinburgh, UK
perdita.stevens@ed.ac.uk

Abstract. Bidirectional transformations maintain consistency between information sources, such as different models of the same software system. In certain settings this is undeniably convenient – but is it important? I will argue that developing our ability to engineer dependable bidirectional transformations is likely to be crucial to our ability to meet the demand for software in coming decades. I will discuss some of the work that has been done so far, including some I've had a hand in, and what challenges remain.

1 Introduction

It is usually held [15] that whenever the title of an article is given the form of a boolean question, the answer to the question should be **No**. For, if I believe the answer to be **Yes**, why have I not titled this paper, and the talk it accompanies, "Bidirectionality is Important"?

Any reader who started at the abstract, or indeed, who knows me, will guess that – pace Betteridge [4] and Hinchcliffe [12] – this maxim does not apply here. In this case, the question is a marker for some complexity that I wish to discuss. What do we, or should we, mean by "important"? What is "bidirectionality"? Where are we going? What is it all for?

In order to tackle these ridiculously large questions, let me start with some philosophical background.

2 What Does "Important" Mean to Humans?

To say, to an audience of researchers in software engineering, that a topic is "important", is to say that it deserves research attention, because paying it this attention may eventually have a positive impact on the practice of software engineering.

This involves not only a prediction ("may eventually") but also a value judgement ("positive"). Indeed, whenever we decide to classify an issue as "important", we are making a value judgement. Such a judgement may involve an appeal to an individual's philosophy or religion; but some things are certain.

1. We make the judgement using our brain, which has evolved, over many millions of years of natural selection, to enhance our individual survival by prioritising what to pay attention to. What an individual, of any species, needs to pay attention to depends on the nature of the species. As humans:

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 1–11, 2018.
https://doi.org/10.1007/978-3-319-92997-2_1

2. We are members of a social species. That is – like a variety of animals from lions to mole-rats to bees – we tend to form cooperative groups which we will call societies. Sometimes it is important to be aware that this fact underlies the things we think of as worst, as well as those we think of as best, about humanity. *A* society has a variety of relations between its individuals, and there are most likely also individuals who are considered to be entirely outside it. We cooperate, but not always, and not perfectly.
3. We are members of a species that uses tools. Indeed, like primates that strip leaves off twigs to fish for insects, we *make* tools, and like beavers, we engineer our own ecosystem. We take deliberate action to change our environment from how it is, to how (we think) we would prefer it to be.

Were any of these things not the case, this conference could not exist. What, though, are the implications for our subject matter?

Software systems, just like the insect-fishing twigs, are tools. The purpose of developing a software system is to modify something about the environment, broadly conceived (e.g. in that people are part of one another's environments). Because we are a social species, this may not always be immediately apparent to someone who is working on the software. Somebody, somewhere, considers the modification to the environment, that developing the software system effects, to be beneficial; but that someone may be socially a long way removed from those who develop it.

For example, I am writing this paper using a text editor named Emacs, and I consider the environment in which I can do this much better than one in which I would have to write it using a typewriter. The earliest editors were developed by people who wanted to use them themselves, as well as to make them available to others. When I was developing software to help my employer chase up non-paying customers, though, the benefit to me was only indirect (I got paid), and the people most affected by the existence of that software certainly did not consider it beneficial.

Thus, it may not be evident to one person that another person considers a piece of software to be having a beneficial effect. It is worse than that: neither within, nor between, societies are our interests perfectly aligned. Therefore, wherever more than one person is affected by some software, there is the potential for conflict, which must somehow be resolved. Indeed, all human conflict is about reconciling competing interests. On a small scale – between people who are socially close – we do this informally, using our faculties of empathy and perspective taking. On a larger scale, where these faculties prove insufficient, we resort to making agreements, often explicit treaties, or *contracts*, expressing desired relationships between things different people care about.

In the technical context, we shall return to this in Sect. 6. But first, let us look more closely at computer systems.

3 What Do Computer Systems Do?

Evolutionarily speaking, the important thing that computer systems do is to affect the environment of one or more human beings. They may do this rather directly, like computer games, providing stimuli that affect the brain's sensations of pleasure. They may give someone more, or less, money. They may enable communication between several humans. They may be instrumental in the growing of food, or in the transportation of a human from Edinburgh to Toulouse.

It is not important that computer systems implement computable functions from some inputs to some output. That is merely part of how they do the important things. As soon as we move from mathematics to software engineering, we have to move up and out.

The attraction of abstracting what our tools do as mathematical functions is that these are easy to think about. However, *effects* such as those discussed are, compared to pure functions, difficult to reason about, and because we do not (so far) have a relationship of empathic trust with our computer systems, we need to be able to reason about what they do. The paradigm of software structured as objects, which have (encapsulated) state, behaviour and identity [5], and communicate by a predefined collection of messages that have limited capacity to be parameterised, fits easily into a human mind. It is an admittedly limited approach to managing just the effect of statefulness, but it is hard to argue with the overwhelming success of object orientation. (Managing state and other effects using monads is more powerful, but jokes about the plethora of object tutorials would fall flat [11].[1])

If reasoning about effects is hard, but software must have effects, what gives? To date, it has been relatively straightforward to step over the gap between inside and outside a computer system without noticing it. We abstract the gap in terms of sensors and actuators, or in terms of a user interface in which all the important interfacing is done by the user. The interaction between a computer system and a human is typically of the same order of complexity as that between a human and a company they do business with: it can be governed by a relatively impoverished contract.

The time in which that gap has been easy to ignore, though, may now be coming to an end. Artificial intelligence is talked of by the general public again, and concern is rising about the way in which computer agents can be disguised as human ones – even though, so far, the disguises (of nefarious Twitter bots for example) are rather crude. The common feature is that the effects these computer systems may have on our environment – social, political or physical – are not predictable; given that we also do not have reason to trust them, the result is fear. As the interface between computer systems and human individuals and societies becomes more fractal, the interaction becomes more like that between humans, in how we must reason about it, than we are used to; this happens

[1] See also https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/,https://wiki.haskell.org/Monad_tutorials_timeline.

even if the computer side does not have properties we are ready to label as intelligence.

In order to get to the implications of this for software engineering, let us think about how we handle predicting or trusting one another. When we talk about our own behaviour, we talk about having good *habits*; interpersonally, the specification each person thinks they should obey is called their *principles*; our attempts to ensure that other people behave in a principled way towards is us called having healthy *boundaries*.

What does it mean for a software system to have good habits, principles, boundaries? How can we possibly manage such concepts? Before we return to these matters let us consider, more concretely, the problems of today's software engineering and how models, and bidirectionality, fit in.

4 What Are the Problems of Software Engineering?

Does software engineering have problems? The term "software crisis" seems to have been coined at the NATO Conference on Software Engineering in 1968. From then on, peaking about 1990, it was a commonplace that we had a software crisis: that is, an inability to develop enough software, with high enough dependability, to meet demand.

Then, use of this phrase began to fall away, and we came to take for granted that software systems, even large ones, even safety critical ones, can indeed be developed predictably enough – at least without hugely more difficulty than, say, tram systems [6]. Concern turned to the supply of talented developers needed to meet the ever-increasing demand for software; to attempt to tackle this, we see educational initiatives aimed at persuading young children to learn to code and consider software engineering as a career.

Throughout, the "pain" experienced in software engineering has been in two main areas: requirements, and maintenance. Curiously, I suggest, the innovations that have led to the demise of the "software crisis" have not focused on either of these areas. Books could be written on what those innovations are: my point here is that what we have *not* seen is a revolution in how requirements are gathered and managed, nor radically new ways to handle maintenance of software systems. Rather, in adopting *object orientation* we have learned to structure our system in terms of relatively stable units, viz. classes, rather than functions, setting boundaries around state. We have streamlined the process of developing dependable software. For example, we have developed ways to catch errors early, in the form of *automated testing*. We have derived some benefit from verification, especially lightweight fully automated verification such as powerful *type systems*, which render certain classes of error impossible even in principle. In parallel, we have increased the *agility* of the software development process, in which developers with highly disciplined habits are able to "embrace change" [3]. Some of these advances work well together; others, as yet, do not.

It is, I think, no accident that these key elements connect so clearly with the habits, principles and boundaries we identified in the previous section. In the end,

all of the problems of software engineering come down to one thing: a human being can only hold so much in their head. We all hate the feeling of being overwhelmed, of knowing that there are vital facts that we have temporarily forgotten. In order to make progress, we invent means of managing complexity, and putting in front of ourselves all, and only, the information that we need at a given time for a given purpose. (Indeed, arguably this is also our motivation for developing habits, principles, and boundaries: all of these limit the range of possible behaviours that we need to consider.) The key challenge is typically to identify what to leave out, whether that is possible future requirements (YAGNI, "you ain't gonna need it") or detail about what an existing piece of software does (a more detailed specification, even if it is correct, is not better, if it includes information that is of no benefit to its user). That brings us, naturally, to models.

5 What Are Models, and What Are They for?

My favourite one-sentence definition of a model is this:

> A model is an abstract, usually graphical, representation of some aspect of a system.

As is often observed, this, like other definitions of models, does not technically exclude much – "everything's a model". What it does do is to emphasise what is important about models: they represent some things, *but not other things*, that are true about systems. A model has, conceptually, a boundary: a piece of information may be inside the model, or not. That is, they allow us to separate concerns [8]. A model may be designed for a particular purpose – which may be prescriptive or descriptive – to include all and only that information that is necessary for the purpose.

Often the purpose is supporting the work of a particular group of stakeholders in the system: that is, different people may use different models. Models still provide benefit, though, and people still spontaneously develop them, any time there are discernibly different concerns that it is helpful to separate. The purpose of a model is to focus attention on what is important to some person at some time.

Different concerns require access to different information. Life is easiest when they require access to *completely* different information: the models are orthogonal, in that a change in one can be made without any implications for another. Recalling that everything is a model, we may for example think about two classes in different subsystems, that do not interact, as possible models. The developer of each one may be able to work quite happily on their own class, unaware of what the developer of the other is doing. However, this happy state of affairs is rare.

6 What Is Bidirectionality?

Finally, it is time to mention bidirectionality, and bidirectional transformations. There is a wide field of research on this topic which we will not survey: one place

to turn for further reading is a recent set of tutorial lectures [9], especially its introductory chapter [2]. The Bx Wiki[2] gives further pointers.

The essence of bidirectionality in a situation is:

- There is separation of concerns into explicit parts such that
- more than one part is "live", that is, liable to have decisions deliberately encoded in it in the future; and
- the parts are not orthogonal. That is, a change in either part may necessitate a change in the other.

Bidirectionality may be present, and it may be helpful to think in these terms, even without there being any relevant automation. The management of a bidirectional situation may be automated to a greater or lesser degree, and this is the job of a bidirectional transformation.

A bidirectional transformation is a means of maintaining consistency between some data sources, such as models of a software system. It is often convenient to separate this job conceptually into two tasks:

1. check whether the sources are consistent;
2. if not, change at least one of them (we may or may not wish to specify which), such that they become consistent.

Each of these tasks could, of course, be carried out by a conventional (unidirectional) program. The key observation justifying the study of bidirectional transformations as a distinct idea is that the tasks are so tightly coupled that, to ensure that they have behaviour that is jointly sensible, they should be engineered together. For example, we normally want a guarantee that the process of consistency restoration should, indeed, result in consistent sources (it should be "correct"); and this should remain true, even if the notion of consistency changes during the course of development, so that the bidirectional transformation must itself be updated. Writing separate programs to carry out the tasks involves duplication of effort and requires a separate check of whatever coherence properties between the tasks are required. Therefore it is extremely helpful for the bidirectional transformation to be written as a single program in a bidirectional language, one artefact incorporating both the definition of consistency and the instructions about how to restore it properly.

Thus, a bidirectional transformation must include a definition of what it is to be "consistent". The term sometimes gives difficulty to people who are used to using it in a logical sense. For our purposes here, consistency is nothing but a mathematical relation on the sets of models we consider. Given a tuple of models, it is possible (in principle) to say whether they are, or are not, consistent. If they are, and if their own groups of stakeholders are each happy with their own model, we consider that they are in a (relatively) good state from which to continue development; if not, something needs to be fixed – perhaps not immediately, but eventually. This is a very flexible notion. Given two sets of models, there

[2] http://bx-community.wikidot.com/.

is a wide choice of possible consistency relations, depending on what kind of consistency we are interested in for the particular development scenario, and how much automation we choose. Crucially, consistency need not be a bijection, but this does *not* imply consistency restoration will be non-deterministic (the process may look at both models, or even more information than that).

As a concrete example, suppose that one model is the Java source file for a class, and the other is that of a JUnit test class. (Recall that while models are usually graphical, everything's a model – this certainly includes everything we call code, which sometimes yields the most familiar examples.) Our bidirectional transformation might incorporate any of the following notions of consistency (in which of course we elide some details), or many others:

1. The files compile together without error.
2. 1. holds, and the JUnit file includes a test for every public method.
3. 2. holds, and all the tests pass.
4. 3. holds, and a certain coverage criterion is met.

The more stringent the notion of consistency we use, the more difficult may be the task of restoring consistency when one of the models is changed; on the other hand, the more work may be saved for the users of the models. There is a trade-off between work invested in automating the bidirectional transformation, and work invested in manually updating the models.

(The connection to the logical – strictly, model-theoretic, for a different sort of "model"! – sense is that *if* we identify a model with a set of statements about a system, then the relation we are interested in *may* be that the union of the statements given by all the models is logically consistent, so that there is a system about which all the statements are true. But for reasons we will not go into further here, this is not normally a helpful perspective e.g. because models and their relationships do more than make statements about a hypothetical system: they also facilitate development.)

To use bidirectional transformations in practice, we need both theoretical underpinnings and support from languages and tools. At present, the design space of ways to represent bidirectional transformations is wide open. Many languages have been proposed, a few of which have gone beyond being academic prototypes; despite early successes, none has yet achieved more than a tiny degree of real-world penetration. We should not be despondent about this: the problem is hard. While, collectively, we have decades of experience designing hundreds of unidirectional languages in several paradigms, for bidirectional languages that experience does not yet exist. Even basic questions remain unanswered. In some cases, we may eventually reach consensus; in others, it will likely turn out that the right answer depends on the circumstances. Here are a few examples, each of which has both theoretical and engineering aspects.

- What properties should a language enforce on every bidirectional transformation, and what will we need other mechanisms to check? For example, should the language enforce any formal least-change property [7], to capture the idea that consistency restoration should not change a model *gratuitously*?

- To what extent, if any, should a bidirectional transformation maintain and use information beyond the models themselves, e.g. a record of the relationship between parts of the models it relates (trace links)?
- Should we directly program transformations which are symmetric (that is, between models *each* of which contains information not present in the other? Or should we program asymmetric transformations (between a source and a view which is an abstraction of the source), relying on span or cospan constructions [10] to give the symmetric behaviour?
- Should our languages directly support maintaining consistency between more than two models? Or should we program binary bidirectional transformations, and rely on separate mechanisms to maintain consistency in networks of models related by these [13,14]?
- How should our language handle effects, such as state, non-determinism, exceptions, and user interaction, while still maintaining guarantees of good behaviour [1]? (Getting this right is the key to managing the other perennial trade-off of automation, between increased reliability of the automated process and decreased flexibility, compared to the manual process.)

If we can develop good-enough bidirectional transformation languages, and other supporting tools, the rewards should be great. To date, the adoption of model driven development has been limited by the difficulty of incorporating it into agile development. As long as models have to be maintained manually, separately from code and from one another, this difficulty remains. Bidirectional transformations have potential to allow the automation of this process. Imagine if developers could work with whatever model was most appropriate to the change they were making, with all others, including the final system, automatically updated to match. This would be an advance worth working for, even just considering the needs of today's software engineering. It could increase the speed of producing software, by enabling developers to work with the most cognitively efficient representations of what they need to have in mind. It could enable the development of software that is both agile and continuously verified and documented by means of tools working on appropriate, automatically updated, models.

7 What Is the Future of Software Engineering?

So far, we have been thinking inside the confines of software engineering as it is now. It is inconceivable, though, that the development of software fifty or a hundred years hence will be done in the same way as today. We have already alluded to one of the forces for change: the inexorably increasing demand for software which is updated ever more frequently. Pushing to solve this just by recruiting more software developers is absurd. We will have more important things for those five-year-olds to do when they grow up. Simultaneously, we are fractalising the boundary between software and the rest of our environment and ourselves, even as we demand greater dependability – trustworthiness – from

the software. The more our software becomes intelligent, the harder it will be to blame its developers for every wrong decision of the software – but we will demand someone be blamed.

Software will be so different, and will be developed so differently, that we may even have some difficulty in even recognising the software and its development process. I would not be surprised if today's programming languages look as puzzling to my grandchildren (if any) as the punched cards my father used to work with look to me. Doubtless, I am completely failing to foresee some important changes. Fundamentally, though, it is economically essential that more of the decisions about what our software should do will be moved away from software specialists, and towards people whose jobs only touch on software development. They may include people who use the software in different roles, but also, specialists in, say, safety, protection of personal data, or support for users with special needs. Rather than having to commission changes in the software from software specialists, or put up with a poor fit with their needs, they will be able to make changes directly. In order to make their decisions, these people will have to be provided with information, some of which they will change. The information given to a particular person is what we term a model. Such models blur the distinction between developer and user. (Lest this sound too fanciful, bear in mind that we already live in this world to some extent: what proportion of the software systems you work with on a daily basis have a settings screen, or similar? This is nothing other than a model, albeit one that may often seem too simple for your needs.) In any setting where more than one person has a model, there will have to be a task of reconciling decisions that the people make and record in their models; that is what we have termed bidirectionality.

We cannot possibly expect to specify in detail the behaviour of this exploding mass of ever-changing software. Rather, we will have to develop analogues of the habits, principles and boundaries that we expect human participants in our world to have. These analogues will abstractly represent certain aspects of the system's behaviour: that is, they are models. Just as humans adjust their behaviour to meet one another's (most vital) expectations, even when they encounter unexpected situations, so will our software have to do in future. This will require the models to adjust, automatically, to meet non-orthogonal expectations – bidirectionality again.

We already touched, in the more mundane setting of relating a class with its tests, on the fact that different degrees of automation, offering different guarantees, are possible. The same will apply in the more challenging future we envisaged. Indeed we do not expect to be able to rely on our fellow humans to behave perfectly first time in every interaction; it would be unreasonable to expect that of computers, once the interface becomes comparable in complexity.

Therefore, to my mind, the most fundamental change we must, as researchers and engineers, facilitate, is that software must in future be able to *explain* its past behaviour and *negotiate* its future behaviour. Our non-software-specialists manipulating their models will sometimes see behaviour they do not expect. They need to be able to ask "why did that happen?" and get an answer they

can understand, in order to know whether to continue to trust the software or not. “Computer says No” will not do in future: it must say why not. It may also take on board a human’s explanation of why it should have said Yes, adjusting a model accordingly. (We have already explored simple cases where a bidirectional transformation may be refined by interaction with a human user [1]. Human-directed explanation will be much harder.) Explanations will have to be made in terms that both computers and humans can understand; non-software-specialists are not going to learn to do debugging the way software engineers do. Explanations must be trustworthy; they must stand up in a court of law; they must be framed like other such explanations, in terms of how the software interpreted its knowledge and goals in the context of its principles.

8 Conclusions

In this essay, I have given some background to the current interest in bidirectional transformations, and have tried to sketch how I think software engineering will change in future and the role of bidirectionality in that future.

A situation involves bidirectionality in an essential way once: there are separated concerns; more than one concern is live; and the concerns are not orthogonal. This already applies to most software development, so that better understanding and automation of bidirectionality would benefit us now. In the future, I think that the increasing complexity of interactions between humans and computers will render advances in the management of bidirectionality even more beneficial.

In conclusion: yes, bidirectionality is important.

References

1. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Notions of bidirectional computation and entangled state monads. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 187–214. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19797-5_9
2. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Introduction to bidirectional transformations. In: Gibbons and Stevens [9], pp. 1–28 (2018)
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
4. Betteridge, I.: Techcrunch: irresponsible journalism. Technovia.co.uk, February 2009. Accessed via [15] 18 Apr 2018
5. Booch, G.: Object Oriented Analysis and Design with Applications. Benjamin/Cummings, San Francisco (1991)
6. Brocklehurst, S.: Going off the rails: the Edinburgh trams saga. http://www.bbc.com/news/uk-scotland-edinburgh-east-fife-27159614
7. Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: On principles of least change and least surprise for bidirectional transformations. J. Object Technol. **16**(1), Article no. 3, 1–31 (2017)

8. Dijkstra, E.W.: Selected writings on Computing: A Personal Perspective. Chapter On the Role of Scientific Thought, pp. 60–66. Springer (1982). https://doi.org/10.1007/978-1-4612-5695-3
9. Gibbons, J., Stevens, P. (eds.): Bidirectional Transformations. LNCS, vol. 9715. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-79108-1
10. Johnson, M., Rosebrugh, R.: Cospans and symmetric lenses. In: Proceedings of the 7th International Workshop on Bidirectional Transformations. ACM (2018)
11. Petricek, T.: What we talk about when we talk about monads. Art Sci. Eng. Program. **2**(3), Article no. 12 (2018)
12. Shieber, S.M.: Is this article consistent with Hinchliffe's rule? Ann. Improbable Res. **21**(3), 18–19 (2015)
13. Stevens, P.: Towards sound, optimal, and flexible building from megamodels. Talk at Bx 2018 (paper in preparation)
14. Stevens, P.: Bidirectional transformations in the large. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 1–11. IEEE (2017)
15. Wikipedia contributors. Betteridge's law of headlines—Wikipedia, the free encyclopedia (2018). Accessed 18 April 2018

Towards Automatic Generation of UML Profile Graphical Editors for Papyrus

Athanasios Zolotas(✉), Ran Wei, Simos Gerasimou, Horacio Hoyos Rodriguez, Dimitrios S. Kolovos, and Richard F. Paige

Department of Computer Science, University of York, York, UK
{thanos.zolotas,ran.wei,simos.gerasimou,dimitris.kolovos, richard.paige}@york.ac.uk, horacio_hoyos_rodriguez@ieee.org

Abstract. We present an approach for defining the abstract and concrete syntax of UML profiles and their equivalent Papyrus graphical editors using annotated Ecore metamodels, driven by automated model-to-model and model-to-text transformations. We compare our approach against manual UML profile specification and implementation using Archimate, a non-trivial enterprise modelling language, and we demonstrate the substantial productivity and maintainability benefits it delivers.

1 Introduction

The Unified Modeling Language (UML) [10] is the *de facto* standard for software and systems modelling. Since version 2.0, UML has offered a domain-specific extensibility mechanism, *Profiles* [5], which allows users to add new concepts to the modelling language in the form of *Stereotypes*. Each stereotype extends a core UML concept and includes extra information that is missing from it. With profiles, UML offers a way for users to build domain-specific modelling languages (DSML) by extending UML concepts, thus lowering the entry barrier to DSML engineering by building on engineer familiarity with UML and UML tools (a detailed comparison of using UML profiles versus domain-specific modelling technology such as [2,17] is beyond the scope of this paper).

Papyrus [16] is a leading open-source UML modelling tool and after a decade in development, it is developing a critical mass for wider adoption in industry as means of (1) escaping proprietary UML tooling lock-in, (2) leveraging the MBSE-related developments in the Eclipse modelling ecosystem enabling automated management of UML models, and (3) enabling multi-paradigm modelling using a combination of UML and EMF-based DSLs. Papyrus offers support for the development of UML profiles; however, this is a manual, tedious and error-prone process [23], and as such it makes the development of graphical editors that are based on such profiles difficult and expensive.

In this paper, we automate the process of developing UML profiles and graphical editors for Papyrus. We propose an approach, called AMIGO, supported by

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 12–27, 2018.
https://doi.org/10.1007/978-3-319-92997-2_2

a prototype Eclipse plugin, where annotated Ecore metamodels are used to generate fully-fledged UML profiles and distributable Papyrus graphical editors. We evaluate the effectiveness of our approach for the automatic generation of a non-trivial enterprise modelling language (Archimate). Furthermore, we apply our approach on several other DSMLs of varying size and complexity [22], demonstrating its generality and applicability.

2 Background and Motivation

In this section we outline the process for defining a UML profile and supporting model editing facilities in Papyrus. We highlight labour-intensive and error prone activities that motivate the need of automatic generation of those artefacts.

2.1 UML Profile

In order to create a new UML Profile, developers need to create a new UML model and add new elements of type *Stereotype, Property, Association, etc.* to create the desired stereotypes, their properties and their relationships. Papyrus offers, among other choices, that of creating the UML profile via a *Profile Diagram*. Users can drag-and-drop elements from the palette to construct the profile. The properties of each element (e.g., multiplicity, navigability, etc.) can be then set using the properties view. In a profile, each stereotype needs to extend a UML concept (hereby referred to as *base element* or *meta-element*). Thus, users must import the meta-elements and add the appropriate extension links. This process can be repetitive and labour-intensive, depending on the size of the profile.

One of the limitations of UML profiles in Papyrus is that links between stereotypes can be displayed as *edges* only if they extend a *Connector* meta-element. These connectors do not hold any information about the stereotypes that they can connect. Users need to define OCL constraints to validate if source and target nodes are of the desired type and if the navigation of the edges is in the correct direction. These constraints can be rather long and need to be manually written and customised for *each* connector. This can also be a labour-intensive and error-prone process.

2.2 Distributable Custom Graphical Editor

At this point, the created profile can be applied on UML diagrams. Users select a UML element (e.g., Class) and manually apply the stereotype. A stereotype can only be applied to the UML element that was defined as its base element. This task might be problematic as users need to remember the base elements for each stereotype. To address this recurring concern, Papyrus offers at least three possible options which allow users to apply selected stereotypes on UML elements in a single step though palettes. The first involves customisation through a user interface which has to be done manually everytime a new diagram is created and it is not distributable. The other two options require the manual definition

of palette configuration files that are loaded every time the profile is applied on a diagram. Although the first is simpler and requires the definition of a single XML file, it is not encouraged as it is based on a deprecated framework.

The definition of custom shapes for the instantiated stereotypes is another common requirement. SVG shapes can be bound to stereotypes at the profile creation process. However, to make these shapes visible, users must set the visibility of the shape of *each* element to true. Another drawback is that by default the shape overlaps with the default shape of the base meta-element. Users can hide the default shapes by writing CSS rules. The rules can be written once but need to be loaded each time *manually* on every diagram.

To create a distributable graphical editor that has diagrams tailored for the profile and to avoid all the aforementioned drawbacks, users need to *manually* create several models and files shown in Fig. 1. We propose an approach that uses a single-source input to automate this *labour-intensive*, *repetitive* and *error-prone* process. This work is motivated by the increasing interest among our industrial partners on exploring the viability of Papyrus as a long-term open-source replacement for proprietary UML tools. While Papyrus provides comprehensive profile support, the technical complexity is high due to the multitude of interconnected artefacts required (see Fig. 1), which can be a significant hurdle for newcomers. We aim to lower the entry barrier for new adopters and help them achieve a working (but somewhat restricted) solution with minimal effort.

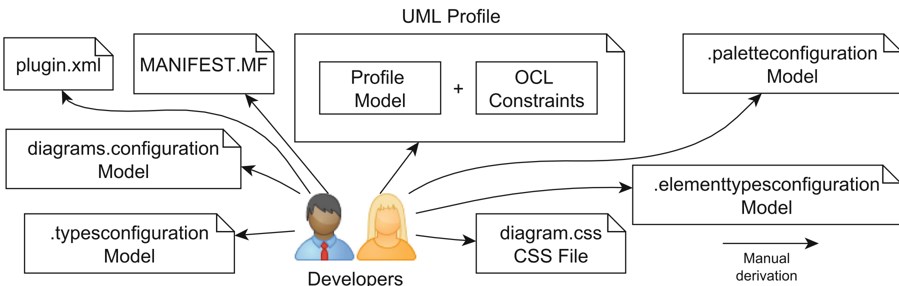


Fig. 1. All the artefacts users need to write to develop a distributable editor.

3 Proposed Approach

We propose AMIGO[1], an automatic approach in which information like stereotypes that should be instantiated in the profile, structural (e.g., references) and graphical information (e.g., shapes) are captured as high-level annotations in an Ecore metamodel, and is transformed into Papyrus-specific artefacts using automated M2M and M2T transformations. Figure 2 shows an overview of AMIGO.

[1] The code and instructions are available at http://www.zolotas.net/AMIGO.

All EClasses in the Ecore metamodel are automatically transformed into stereotypes. Annotated EReferences can also create stereotypes. Developers use the annotations listed below to specify the required graphical syntax of the stereotype (i.e., if it should be represented as a *node* or as an *edge* on the diagram). A detailed list of all valid annotation properties is given in the Appendix.

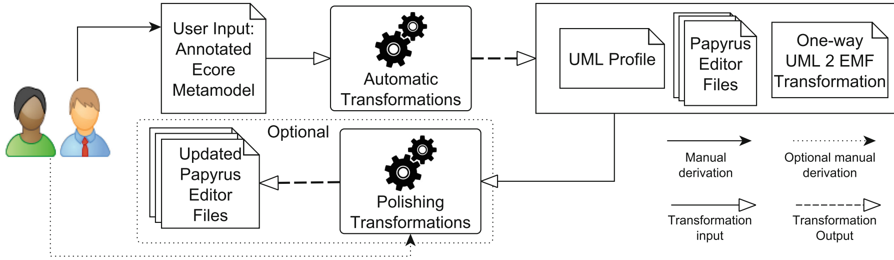


Fig. 2. An overview of the proposed approach

(1) **@Diagram** annotations define diagram-specific information like the name and the icon of the diagram type. This annotation is always placed at the top package of the Ecore metamodel.
(2) **@Node** annotations are used for stereotypes that should be instantiated as nodes in the diagrams. The UML meta-element that this stereotype extends is provided through the *base* property, while the SVG shape and the icon in the palette are specified through the *shape* and *icon* properties, respectively.
(3) **@Edge** annotations are used for stereotypes that should be instantiated as edges and it can be applied to both EClasses and EReferences. The base UML element is provided through the *base* property. The icon in the palette is also passed as property along with the desired style of the line.

The annotation of the Ecore metamodel is the only manual process required in our approach. This Ecore metamodel is then consumed by M2M and M2T transformations shown in Fig. 4 and described in Sect. 4. The transformations are written in the Epsilon Transformation Language (ETL) [13] and the Epsilon Generation Language (EGL) [18] but in principle, any other M2M and M2T language could be used. The automated workflow produces the UML profile with the OCL constraints and all the configuration models/files needed by Papyrus. In addition, a *one-way* M2M transformation, that can be used to transform the UML models back to EMF models that conform to the original Ecore metamodel, is also generated. Thus, model management programs already developed to run against models conforming to the EMF metamodel can be re-used.

AMIGO provides the option to execute *polishing transformations* that allow fine-tuning of the generated artefacts. In the following section, the proposed approach is explained via a running example.

3.1 Running Example

We use AMIGO to define and generate the UML profile and the Papyrus graphical editor for a Simple Development Processes Language (SDPL). We start by defining Ecore metamodel using Emfatic (see Listing 1.1). A process defined in SDPL consists of *Steps*, *Tools* and *Persons*. Each person is familiar with certain tools and has different *Role*s, while each step refers to the next step using the *next* reference. To generate the UML profile and the Papyrus graphical editor, we add the following concrete syntax-related annotations shown in Listing 1.1. AMIGO produces the SDPL Papyrus editor presented in Fig. 3.

- **Line 2:** The name and the icon that should be used in Papyrus menus are defined using the *name* and *icon* properties of the *@Diagram* annotation.
- **Lines 5, 10 & 14:** The *@Node* annotation is used to define that these types should be stereotypes that will be represented as nodes on the diagram. The *base* parameter defines the UML meta-element the stereotype should extend. The shape and the palette icon are given using the *shape* and *icon* details.
- **Lines 17 & 20:** The *familiarWith* EReference and the *Role* EClass are extending the meta-element *Association* of UML. These stereotypes should be shown as links in the diagrams. In contrast with the *familiarWith EReference*, the types the *Roles* edge should be able to connect are not known and need to be specified as properties of the annotation (i.e., *source* = *"src"* and *target* = *"tar"*). This denotes that the source/target nodes of this connector are mapped to the values of the *src*/*tar* EReferences, respectively.
- **NB Line 8:** The *next* EReference is not required to be displayed as an edge on the diagram thus it is not annotated with *@Edge*.

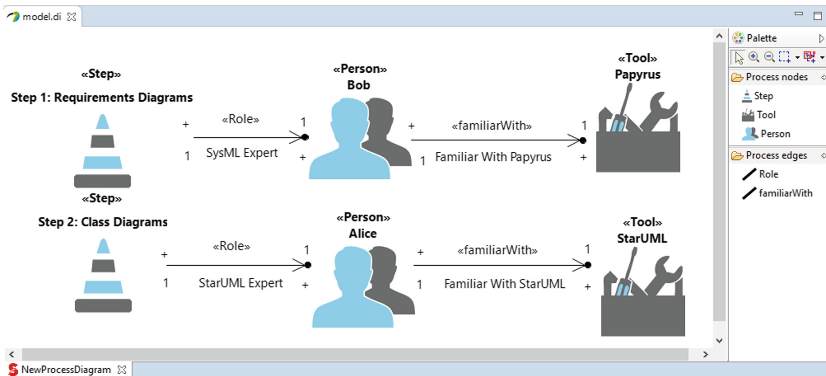


Fig. 3. The SDPL editor for Papyrus generated using AMIGO.

```
1  @namespace(uri="sdpl",prefix="sdpl")
2  @Diagram(name="SDPL", icon="sdpl.png")
3  package Process;
4
5  @Node(base="Class", shape="step.svg",
       icon="step.png")
6  class Step {
7    attr String stepId;
8    ref Step[1] next;
9  }
10 @Node(base="Class", shape="tool.svg",
       icon="tool.png")
11 class Tool {
12   attr String name;
13 }
14 @Node(base="Class", shape="per.svg", icon
       ="per.png")
15 class Person {
16   attr String name;
17   @Edge(base="Association", icon="line.
         png")
18   ref Tool[*] familiarWith;
19 }
20 @Edge(base="Association", icon="line.png"
       , source="src", target="tar")
21 class Role {
22   attr String name;
23   ref Step[1] src;
24   ref Person[1] tar;
25 }
```

Listing 1.1. The annotated ECore metamodel of SDPL.

3.2 Polishing Transformations

The generated editor is fully functional but it can be further customised to fit custom user needs. In this example the labels should be in red font. This can be achieved by manually amending the generated CSS file. However, the CSS file will be automatically overridden if the user regenerates the editor. To avoid this, the user can use the CSS polishing transformation (#6b in Fig. 4) shown in Listing 1.2. Every time the profile and editor generation is executed, the polishing transformation will amend the original CSS file with the information shown in Listing 1.3.

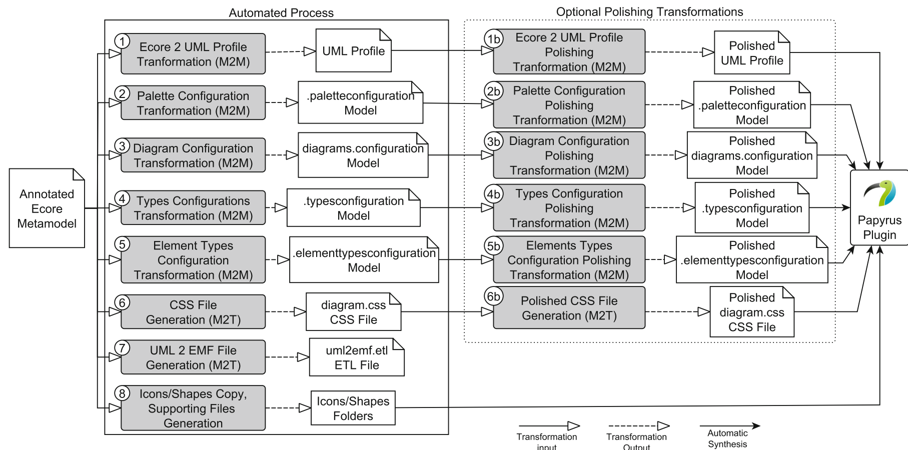


Fig. 4. An overview of the transformation workflow.

```
1 var allNodeStereotypes = Source!EClass.all().select(c|c.getEAnnotation("Node").isDefined());
2 for (stereo in allNodeStereotypes) {%]
3 [appliedStereotypes~=[%=stereo.name%]][% if (hasMore){%] , [%}}
4 %] {
5   fontColor:red;
6 }
```

Listing 1.2. A CSS polishing transformation written in EGL.

```
1 [appliedStereotypes˜=Step],[appliedStereotypes˜=Tool],[appliedStereotypes˜=Person]{
2   fontColor:red;
3 }
```

Listing 1.3. The output that is amended in the original CSS file.

4 Implementation

This section discusses the implementation of the proposed approach. Figure 4 shows the transformations workflow. As the transformations consist of about 1K lines of code, we will describe them omitting low level technical details (see footnote 1). Every step in the process, except that of polishing transformations described in Sect. 4.8 is fully automated, as the only required manual step in AMIGO, is that of annotating the ECore metamodel.

4.1 EMF to UML Profile Generation (#1)

The source model of this transformation is the annotated Ecore metamodel and the target model is a UML profile model. This transformation consists of two rules: the first creates one stereotype for each EClass in the metamodel and the second creates a stereotype for EReferences annotated as @Edge.

When all stereotypes are created, a number of post-transformation operations are executed to (1) create the generalisation relationships between the stereotypes, (2) add the references/containment relationships between the stereotypes, (3) create the extension with the UML base meta-element and (4) generate and add the needed OCL constraints for each edge:

(1) For each of the superclasses of an EClass in the metamodel we create a *Generalisation* UML element. The generalisation element is added to the stereotype created for this specific EClass and refers via the *generalization* reference to the stereotype that was created for the superclass.
(2) For each reference (ref or val) in the metamodel a new *Property* UML element is created and added to the stereotype that represents the EClass. A new *Association* UML element should also be created and added to the stereotype. The name and the multiplicities are also set.
(3) By default the stereotypes extend the *Class* base element unless a different value is passed in the *base* property of the @Node/@Edge annotation. In this post-transformation operation the necessary *Import Metaclass* element and *Extension* reference are created and attached to the stereotype.
(4) The OCL constraints are created for each stereotype that will be shown as an edge on the diagram. Two *Constraint* and two *OpaqueExpression* elements are created for each edge stereotype that check the two necessary constraints.

4.2 Constraints

To illustrate the OCL constraints, we provide a partial view of the SDPL UML profile in Fig. 5[2].

[2] The attributes of the stereotypes are omitted for simplicity.

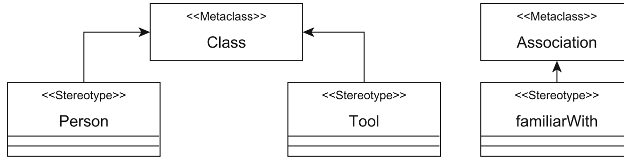


Fig. 5. Example UML profile for SDPL.

In Fig. 3, the *familiarWith* association is used to connect *Person Alice* with *Tool StarUML*. However, the *familiarWith* stereotype can be applied to any *Association*, and not strictly to *Associations* which connect *Person* and *Tool* stereotyped elements. Therefore, constraints are needed to check two aspects:

- **End Types:** the elements that a *familiarWith* association connects have *Person* and *Tool* stereotypes applied;
- **Navigability:** the *familiarWith* association *starts* from an element stereotyped as *Person* and *points* to an element stereotyped as *Tool*.

End Types. In Listing 1.4, line 1 accesses the types that *familiarWith* connects. Lines 2 and 3 check if the types that *familiarWith* connects are of type that either has stereotype *Person* or *Tool*.

```
1 let classes = self.base_Association.endType→selectByKind(UML::Class) in
2   classes→exists(c|c.extension_Person→notEmpty()) and
3     classes→exists(c|c.extension_Tool→notEmpty())
```

Listing 1.4. The End Types constraint in OCL.

```
1 let memberEnds=self.base_Association.memberEnd in
2 let toolEnd=memberEnds→select(type.oclIsKindOf(UML::Class) and type.oclAsType(UML::
        Class).extension_Tool→notEmpty()),
3 personEnd=memberEnds→select(type.oclIsKindOf(UML::Class) and type.oclAsType(UML::
        Class).extension_Person→notEmpty()) in
4 if personEnd→notEmpty() and toolEnd→notEmpty() then
5 personEnd→first().isNavigable() = false and
6 toolEnd→first().isNavigable() = true
7 else false endif
```

Listing 1.5. The Navigability constraint in OCL.

Navigability. In Listing 1.5, in lines 2 and 3, we obtain the member ends that *familiarWith* connects. If these ends are obtained successfully (line 4), we check that the *personEnd* (connecting element stereotyped as *Person*) is not navigable (line 5) and the *toolEnd* (connecting element stereotyped as *Tool*) is navigable (line 6). Therefore, we are checking that a *familiarWith* association can only go from *Person* to *Tool*.

We use the *End Types* and *Navigability* constraints as templates with dynamic sections, where the specific stereotype names are inserted dynamically.

4.3 Palette Generation (#2)

This transformation is responsible for creating a model (file .paletteconfiguration) that configures the custom palette. The model conforms to the *PaletteConfiguration* metamodel that ships with Papyrus. The transformation creates a new *PaletteConfiguration* element and adds two new *DrawerConfiguration* elements that represent the two tool compartments in our palette (i.e., nodes and edges). For each element annotated as @Node/@Edge, a new *ToolConfiguration* element is created and added to the appropriate drawer. An *IconDescriptor* element is added to the *ToolConfiguration* pointing to the path of the icon for that tool.

4.4 Diagram Configuration (#3, #4 & #5)

Firstly, in order for Papyrus to offer the choice of creating new custom diagrams for the generated profile via its menus, a *ViewpointConfiguration* needs to be created. This configuration hides the default palette and attaches the one created before. It also binds the generated CSS stylesheet file (see transformation #6) to the diagram. Transformation #3 creates a new model that conforms to the *Configuration* metamodel and stores this new *ViewpointConfiguration* element.

Transformation #4 creates the types configuration model (i.e., .typesconfiguration file) that conforms to the *ElementTypesConfiguration* metamodel provided by Papyrus. This model is responsible for binding the types of the drawn elements to stereotypes. For each stereotype a new *SpecializationTypeConfiguration* element is created and a unique *id* is created in the format "ProfileName.StereotypeName" (e.g., "SDPL.Step"). The value of the *Hint* attribute is set to the qualified name of the meta-element that this type specialises (e.g., "UML::Class"). A new *StereotypeApplicationMatcherConfiguration* element is also created that holds the qualified name of the stereotype that should be applied to the drawn element. Binding is performed by creating a new *ApplyStereotypeAdviceConfiguration* element that points to the equivalent stereotype application matcher configuration element created before. This way, when an element of a specific type is drawn the appropriate stereotype is applied automatically.

The last model (i.e., the .elementtypesconfiguration file) created by transformation #5 is one that conforms to the *ElementTypesConfiguration* metamodel. This model is responsible for specializing the meta-element *shapes* to the custom ones created by the profile. For each stereotype, a new *SpecializationTypeConfiguration* element is created pointing to two elements that it specialises: the specialization type configuration created in transformation #4 and the shape of the UML meta-element that this element specialises.

4.5 Stylesheet Generation (#6)

In Papyrus, the look and feel of diagram elements can be customised using CSS. Each node on a diagram has a set of compartments where the attributes, the shape, etc. appear. Initially, we create a CSS rule to hide all their compartments

and another rule to enable the compartment that holds the shape. The latter rule also hides the default shape inherited from the meta-element the stereotype extends. Then, for each stereotype that appears as a node, a CSS rule is generated to place the SVG figure in the shape compartment. Finally, we generate the CSS rules for each edge, e.g., if a *lineStyle* parameter is set, then the *style* property for that Edge stereotype is set to the value of the lineStyle parameter (e.g., "solid").

4.6 UML to EMF Transformation Generation (#7)

This M2T transformation generates the ETL file that can transform the UML models that conform to the UML Profile, back to EMF models that conform to the source Ecore metamodel. One rule is generated for each of the stereotypes that transforms the elements having this stereotype applied to them back to the appropriate type of the Ecore metamodel. Each stereotype has the same attributes and references as the original EClass thus, this M2T script also generates the statements that populate the attributes and the references. An example of an auto-generated rule is shown in Listing 1.6. The rule transforms elements stereotyped as "Person" in the UML model to elements of type "Person" in an EMF model which conforms to the Ecore metamodel presented in Listing 1.1.

```
1 rule PersonUML2PersonEMF
2   transform s: UMLProcess!Person
3   to t: EMFProcess!Person {
4     t.name = s.name;
5     t.age = s.age;
6     t.familiarWith ::= s.familiarWith;
7   }
```

Listing 1.6. Example of an auto-generated ETL rule.

4.7 Icons, Shapes and Supporting Files (#8)

The approach creates a Papyrus plugin, thus the "MANIFEST.MF", the "plugin.xml" and the "build.properties" files are created. The first includes the required bundles while defines the necessary extensions for Papyrus to register the UML profile and create the diagrams. The third points the project to the locations of the "MANIFEST.MF" and "plugin.xml" files. Finally, two files necessary for Papyrus to construct the UML profile model, (namely "model.profile.di" and "model.profile.notation") are generated.

4.8 Polishing Transformations (#1b–#6b)

For each of the transformations #1–#6, users are able to define polishing transformations that complement those included in our implementation. After each built-in transformation is executed, the workflow looks to find a transformation with the same file name which is executed against the Ecore metamodel and updates the already created output model of the original transformation.

5 Evaluation

AMIGO is evaluated by firstly, applying it to generate a Papyrus editor for the non-trivial Archimate UML profile [11,12]. The Adocus Archimate for Papyrus[3] is an open-source tool that includes a profile for Archimate and the appropriate editors for Papyrus. We can compare the proportion of the tool that AMIGO is able to generate automatically, the number of polishing transformations that the user needs to write to complete the missing parts and finally, identify the aspects of the editor that our approach is not able to generate. As a result we can measure the *efficiency* of AMIGO in generating profiles/editors. Secondly, we assess the *completeness* of our approach by applying it on nine other metamodels collected as part of the work presented in [22] testing if our approach can successfully generate profiles and editors for a wide variety of scenarios.

5.1 Efficiency

The Archimate for Papyrus tool offers five kind of diagrams (i.e., Application, Business, Implementation and Migration, Motivation and Technology diagrams). Thus, in this scenario we need to create the 5 Ecore metamodels and annotate those EClasses/EReferences to generate the profiles and the editors. AMIGO successfully generated the Papyrus editor for Archimate, however, some special features that the Archimate for Papyrus tool offers need further work. For example, the tool offers a third drawer that is called “Common” and includes two tools (i.e., “Grouping” and “Comment”). To implement such missing features, we need to write the extra polishing transformations. For brevity, we will not go into details on the content of the polishing transformations for this specific example.

Table 1, summarises the lines of code (LOC) we had to write to generate the editors using AMIGO versus the lines of code the authors of the Archimate for Papyrus had to write. Since all the artefacts except the CSS file are models, we provide in parentheses the number of model elements users have to instantiate. For the polishing transformations we only provide the LOC metric as the models are instantiated automatically by executing the transformation scripts and not manually. Our approach requires about 91% less handwritten LOC to produce the basic diagrams and about 86% less code to produce the polished editor. In terms of model elements, we need to manually instantiate about 63% less model elements (668 vs. 1828) for the basic editor. Our approach creates the five diagram editors which offer the same functionality and features as the original Archimate for Papyrus tool five diagram editors but also atop that the ETL transformation and the OCL constraints.

[3] https://github.com/Adocus/ArchiMate-for-Papyrus.

Table 1. Lines of manually written code (and model elements in parenthesis) of each file for creating a Papyrus UML profile and editor for ArchiMate.

File	AMIGO			Archimate for Papyrus
	Handwritten	Handwritten (polishing)	Total	Total handwritten
Ecore	436 (668)	0	436 (668)	0
Profile	0	0	0	1867 (1089)
Palette configuration	0	24	24	1305 (323)
Element types configuration	0	11	11	237 (61)
Types configuration	0	10	10	788 (327)
Diagram configuration	0	0	0	58 (28)
CSS	0	195	195	537
Total	436 (668)	240	676 (668)	4792 (1828)

5.2 Completeness

In addition, we tested AMIGO with nine more Ecore metamodels from different domains. The names and their sizes (in terms of types) are given in Table 2. Next to the size, in parentheses, the number of types that should be transformed so they can be instantiated as nodes/edges is also provided. The approach produced the profiles and the editors for *all* the metamodels, demonstrating that it can be used to generate the desired artifacts for a wide spectrum of domains.

Table 2. The metamodels used to evaluate the completeness of AMIGO.

Name	#Types (#Nodes/#Edges)	Name	#Types (#Nodes/#Edges)
Professor	5 (4/5)	Ant scripts	11 (6/4)
Zoo	8 (6/4)	Cobol	13 (12/14)
Usecase	9 (4/4)	Wordpress	20 (19/18)
Conference	9 (7/6)	BibTeX	21 (16/2)
Bugzilla	9 (7/6)	Archimate	57 (44/11)

5.3 Threats to Validity

There were a few minor features of the original Archimate for Papyrus tool that our approach could not support. Most of them are related to custom menu

entries and wizards. For those to be created, developers need to extend the "plugin.xml" file. In addition, the line decoration shapes of stereotypes that extend the aggregation base element (i.e., diamond) can only be applied dynamically by running Java code that will update the property each time the stereotype is applied. Our default and polishing transformations are not able to generate those features automatically; these should be implemented manually. For that reason, we *excluded* these lines of code needed by Archimate for Papyrus to implement these features from the data provided in Table 1 to achieve a fair comparison.

6 Related Work

Over the past years, several UML profiles have been standardised by the OMG (e.g., MARTE [9], SySML [4]) and are now included in most major UML tools (e.g., Papyrus [16]). A list of recently published UML profiles is available in [17]. Irrespective of the way these UML profiles were developed, either following ad-hoc processes or based on guidelines for designing well-structured UML profiles [5,19], they required substantial designer effort. Our approach, subject to the concerns raised in Sect. 5, automates the process of generating such profiles and reduces significantly the designer-driven effort for specifying, designing and validating UML Papyrus profiles and editors.

Relevant to our work is research introducing methodologies for the automatic generation of UML profiles from an Ecore-based metamodel. The work in [15] proposes a partially automated approach for generating UML profiles using a set of specific design patterns. However, this approach requires the manual definition of an initial UML profile skeleton, which is typically a tedious and error-prone task [23]. The methodology introduced in [7,8] facilitates the derivation of a UML profile using a DSML as input. The methodology requires the manual definition of an intermediate metamodel that captures the abstract syntax. Despite the potential of these approaches, they usually involve non-trivial human-driven tasks, e.g., a UML profile skeleton [15] or an intermediate metamodel [7,8]. In contrast, our approach builds on top of standard Ecore metamodels (which are usually available in MBSE). Furthermore, our approach supports the customisation of UML profiles and the corresponding Papyrus editor.

Our work also subsumes research that focuses on bridging the gap between MOF-based metamodels (e.g., Ecore) and UML profiles. In [1], the authors propose a methodology that consumes a UML profile and its corresponding Ecore metamodel, and uses M2M transformation and model weaving to transform UML models to Ecore models, and vice versa. The methodology proposed in [23] simplifies the specification of mappings between a profile and its corresponding Ecore metamodel using a dedicated bridging language. Along the same path, the approach in [6] employs an integration metamodel to facilitate the interchange of modelling information between Ecore-based models and UML models. Compared to this research, AMIGO automatically generates UML profiles (like [6,23]), but requires only a single annotated Ecore metamodel and does not need any mediator languages [23] or integration metamodels [6]. Also, the transformation of

models from UML profiles to Ecore is only a small part of our generic approach (Sect. 4.6) that generates not only a fully-fledged UML profile but also a distributable custom graphical editor.

In terms of graphical modelling, our approach is related to EuGENia [14], which transforms annotated Ecore metamodels to GMF (Graphical Modelling Framework) models (i.e. GMF graph definition model, tooling model, mapping model and generation model) to automatically generate graphical model editors. Sirius [20], a tool based also on GMF, enables users to define a diagram definition model and use this model to generate a graphical editor. Unlike EuGENia, Sirius does not require additions to the original Ecore metamodel.

7 Conclusions and Future Work

In this paper we presented AMIGO, an MDE-based approach that uses annotated Ecore metamodels to automatically generate UML profiles and supporting distributable Papyrus editors. We evaluated AMIGO using Adocus Archimate for Papyrus and nine other metamodels from [21] showing that AMIGO reduces significantly the effort required to develop these artifacts. Our future plans for AMIGO involve providing better support for compartments since although in the current version users can create compartments using the available compartment relationships in UML (e.g., Package-Class, etc.), the visual result is not appealing. More specifically, the compartment where containing elements are placed is distinct and lies above the compartment that hosts the shape. As a result, the contained elements are drawn above the custom shape and not inside it. Also, we will extend AMIGO with support for the automatic generation of OCL constraints for opposite references and more connectors (e.g., UML Dependencies). We also plan to support the execution of validation scripts against the Ecore file to check that the annotation provided in the Ecore file are correct, and if not, to produce meaningful error messages. Finally, we plan to execute usability tests with real users to evaluate AMIGO against the native Papyrus approach.

Acknowledgments. This work was partially supported by Innovate UK and the UK aerospace industry through the SECT-AIR project, by the EU through the DEIS project (#732242) and by the Defence Science and Technology Laboratory through the project "Technical Obsolescence Management Strategies for Safety-Related Software for Airborne Systems".

A Annotations and Parameters

The following are all the currently supported parameters for the annotations.

A.1 @Diagram

- name: The name of the created diagrams as it appears on the diagram creation menus of Papyrus. [required]
- icon: The icon that will appear next to the name on the diagram creation menus of Papyrus. [optional]

A.2 @Node

- base: The name of the UML meta-element that this stereotype should extend. [required]
- shape: The shape that should be used to represent the node on the diagram. [required]
- icon: The icon that will appear next to the name of the stereotype in the custom palette. [optional]

A.3 @Edge

- base: The name of the UML meta-element that this stereotype should extend. [required]
- icon: The icon that will appear next to the name of the stereotype in the custom palette. [optional]
- lineStyle: The style of the line (possible values: solid, dashed, dotted, hidden, double). [optional]
- source (*for EClasses only*): The name of the EReference of the EClass that denotes the type of the source node for the edge. [required]
- target (*for EClasses only*): The name of the EReference of the EClass that denotes the type of the target node for the edge. [required]

References

1. Abouzahra, A., Bézivin, J., Del Fabro, M.D., Jouault, F.: A practical approach to bridging domain specific languages with UML profiles. In: Proceedings of the Best Practices for Model Driven Software Development at OOPSLA, vol. 5 (2005)
2. Bergmayr, A., Grossniklaus, M., Wimmer, M., Kappel, G.: JUMP—from Java annotations to UML profiles. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) MODELS 2014. LNCS, vol. 8767, pp. 552–568. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11653-2_34
3. Erickson, J., Siau, K.: Theoretical and practical complexity of modeling methods. Commun. ACM **50**(8), 46–51 (2007)
4. Friedenthal, S., Moore, A., Steiner, R.: A practical guide to SysML: the systems modeling language (2014)
5. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to UML profiles. UML Model Eng. **2** (2004)
6. Giachetti, G., Marin, B., Pastor, O.: Using UML profiles to interchange DSML and UML models. In: Third International Conference on Research Challenges in Information Science, pp. 385–394 (2009)
7. Giachetti, G., Marín, B., Pastor, O.: Using UML as a domain-specific modeling language: a proposal for automatic generation of UML profiles. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 110–124. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02144-2_13
8. Giachetti, G., Valverde, F., Pastor, O.: Improving automatic UML2 profile generation for MDA industrial development. In: Song, I.-Y. (ed.) ER 2008. LNCS, vol. 5232, pp. 113–122. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87991-6_16

9. Object Management Group: Modeling And Analysis Of Real-Time Embedded Systems (2011). http://www.omg.org/spec/MARTE/1.1/
10. Object Management Group: Unified Modeling Language, June 2015. http://www.omg.org/spec/UML/
11. Haren, V.: Archimate 2.0 specification (2012)
12. Iacob, M.E., Jonkers, H., Lankhorst, M.M., Proper, H.A.: ArchiMate 1.0 Specification. Van Haren Publishing, Zaltbommel (2009)
13. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69927-9_4
14. Kolovos, D.S., García-Domínguez, A., Rose, L.M., Paige, R.F.: Eugenia: towards disciplined and automated development of GMF-based graphical model editors. Softw. Syst. Model. 1–27 (2015)
15. Lagarde, F., Espinoza, H., Terrier, F., André, C., Gérard, S.: Leveraging patterns on domain models to improve UML profile definition. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 116–130. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78743-3_10
16. Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., Terrier, F.: Papyrus UML: an open source toolset for MDA. In: Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009), pp. 1–4 (2009)
17. Pardillo, J.: A systematic review on the definition of UML profiles. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 407–422. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_28
18. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The Epsilon generation language. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 1–16. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69100-6_1
19. Selic, B.: A systematic approach to domain-specific language design using UML. In: 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2007), pp. 2–9 (2007)
20. Viyović, V., Maksimović, M., Perisić, B.: Sirius: a rapid development of DSM graphical editor. In: IEEE 18th International Conference on Intelligent Engineering Systems INES 2014, pp. 233–238. IEEE (2014)
21. Williams, J.R.: A novel representation for search-based model-driven engineering. Ph.D. thesis, University of York (2013)
22. Williams, J.R., Zolotas, A., Matragkas, N.D., Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: What do metamodels really look like? EESSMOD@ MoDELS **1078**, 55–60 (2013)
23. Wimmer, M.: A semi-automatic approach for bridging DSMLS with UML. Int. J. Web Inf. Syst. **5**(3), 372–404 (2009)

Optimising OCL Synthesized Code

Jesús Sánchez Cuadrado(✉)

Universidad de Murcia, Murcia, Spain
jesusc@um.es

Abstract. OCL is a important element of many Model-Driven Engineering tools, used for different purposes like writing integrity constraints, as navigation language in model transformation languages or to define transformation specifications. There are refactorings approaches for manually written OCL code, but there is not any tool for the simplification of OCL expressions which have been automatically synthesized (e.g., by a repair system). These generated expressions tend to be complex and unreadable due to the nature of the generative process. However, to be useful this code should be as simple and resemble manually written code as much as possible.

In this work we contribute a set of refactorings intended to optimise OCL expressions, notably covering cases likely to arise in generated OCL code. We also contribute the implementation of these refactorings, built as a generic transformation component using bentō, a transformation reuse tool for ATL, so that it is possible to specialise the component for any OCL variant based on Ecore. We describe the design and implementation of the component and evaluate it by simplifying a large amount of OCL expressions generated automatically showing promising results. Moreover, we derive implementations for ATL, EMF/OCL and SimpleOCL.

Keywords: Model-Driven Engineering · Model transformations
OCL · Refactoring

1 Introduction

OCL [25] is used in Model-Driven Engineering (MDE) in a wide range of scenarios, including the definition of integrity constraints for meta-models and UML models, as a navigation language in model transformation languages and as input for model finders, among others. The most usual scenario is that OCL constraints are written by developers who can choose their preferred style, and thus tend to write concise and readable code. An utterly different scenario is the automatic generation of OCL constraints. In this setting, the style of the generated constraints is frequently sub-optimal, in the sense that it may contain repetitive expressions, unnecessary constructs (e.g., too many let expressions), trivial expressions (e.g., $false = false$), etc. This is so since synthesis tools typically use templates (or sketches in program synthesis [28]) whose holes are filled by automatic procedures.

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 28–45, 2018.
https://doi.org/10.1007/978-3-319-92997-2_3

Faced with the task of writing a non-trivial OCL synthesizer, the implementor could try to design it in a way that favour the generation of concise and simple expressions, but this introduces additional complexity at the core of the synthesizer which can be hard to manage. An alternative is to generate the OCL constraints in the easiest way from the synthesizer's implementation point of view, and then have a separate simplifying process to handle this task. In this work we propose a catalogue of simplifications for OCL expressions, especially targeted to OCL code generated automatically. The simplifications range from well-known rewritings for integers and booleans to more specific ones related to type comparisons (i.e., oclIsKindOf). This work fills the gap between refactorings and conversions targeted to manually written code [5,6] and the initial work by Giese and Larsson [19] about OCL simplifications.

On the other hand, there are several OCL implementations like EMF/OCL [17], SimpleOCL [31], the embedding of OCL in the ATL language [21], etc. Hence, committing to a single variant would limit the practical applicability of the catalogue. To overcome this issue we have implemented it as a generic transformation component using bentō [8]. Bentō is a transformation reuse tool for ATL, which allows the development of generic transformations that are later specialized to concrete meta-models. In this paper we describe the design and implementation of this component and the main elements of the catalogue. Finally, we have evaluated the catalogue by applying it to a large amount of OCL expressions and specializing it for ATL, EMF/OCL and SimpleOCL in order to show its reusability.

Altogether, this work presents the following contributions. (1) A new set of simplification refactorings for OCL, which complements the ones proposed in [19] and reuses some of ones described in [6,26]. (2) A design based on the notion of generic transformation [8] which allows mapping one definition of the refactorings to several variants of OCL. (3) A working implementation (BeautyOCL) implemented with bentō[1], for which the ATL/OCL specialization has been integrated in anATLyzer[2], our IDE for ATL model transformations. The catalogue can be easily extended with new simplifications and specializations by submitting pull requests to the available GitHub repository.[3]

Organization. Section 2 presents related work, and Sect. 3 motivates the work through a running example. Section 4 introduces the framework used to develop the catalogue of simplifications, and Sect. 5 describes the catalogue. The work is evaluated in Sect. 6, and Sect. 7 concludes.

2 Related Work

The closest work to ours was proposed by Giese and Larsson [19]. The motivation was to simplify constraints generated for UML diagrams in the context of

[1] http://github.com/jesusc/bento.
[2] http://anatlyzer.github.io.
[3] http://github.com/jesusc/beautyocl.

design patterns. Simplifications for primitive types and collections are proposed by means of examples. More complex cases including conditionals, let expressions and the treatment of oclIsKindOf expressions are not handled. We depart from this work and propose a more extensive catalogue. Moreover, we have developed the catalogue using a reusable approach, with the aim of fostering its usage.

Correa et al. investigated the impact of poor OCL constructs on understandability [7], finding that refactored expressions are more understandable. The experiments were carried out on hand-written expressions, thus, it is likely that refactorings for expressions generated automatically have an even bigger impact on understandability. In [32], a catalogue of refactorings for ATL transformations is presented. Some of them are applicable to OCL, but they do not target simplifications. Moreover, the authors point out the possibility of implementing the refactorings in a language independent way, which is now achieved with our framework. The work of Correa and Werner presents a set of refactorings for OCL [6]. Some of them are of interest for our case, particularly refactorings for verbose expressions, while others are particularly useful for hand-written OCL expressions. A complementary work with additional refactorings is presented in [26]. Cabot and Teniente [5] proposes a set of transformations to derive equivalent OCL expressions. Some of these transformations are simplifications, but they generally focus on equivalent ways of writing a given OCL expression. Similarly, a set of optimizations patterns to improve the performance of OCL expressions in ATL programas is presented in [15]. Another source of related works is expression simplification rules developed with program transformation systems [22].

Regarding the applicability of our approach, it is targeted to complement tools which generate or transform OCL constraints. Some of them are based on filling in a pre-defined template from a given model [1,19,29]. Other works modify OCL expressions as a response to meta-model evolution [20]. These approaches could be benefited by our implementation. Nevertheless, given that our target is automatically generated code, it is specially well suited to complement approaches related to the notion of program synthesis and program repair. This is so since they tend to generate "alien code" [23] which may be problematic when humans need to maintain the generated code. To the best of our knowledge, there are only a few systems of this kind in the MDE and OCL ecosystem, like our work in quick fixing ATL transformations [11] and the generation or pre-conditions [12,14,24]. Hence, we believe that this work will be also valuable to complement OCL synthesis tools likely to appear in the future.

3 Motivation and Running Example

In this section we present the running example which will be used throughout the paper. We also use it to motivate the need for our catalogue of simplification refactorings. In previous work we implemented a large set of quick fixes for ATL transformations [11], with which it is possible to fix many types of non-trivial problems. This is integrated in the anATLyzer IDE [13] allowing users to automatically generate and integrate pieces of OCL code that fixes problems in their

transformations. From the usability point view the main concern of our tool was that the generated OCL expressions where often accidentally complex due to the automatic procedure used to generate them. In practice, this means that users may not use the quick fix feature because the OCL expressions which are automatically produced are unnecessarily too complex, difficulting its understanding. This problem is not exclusive of our approach, but it is acknowledged in other works [19, 23].

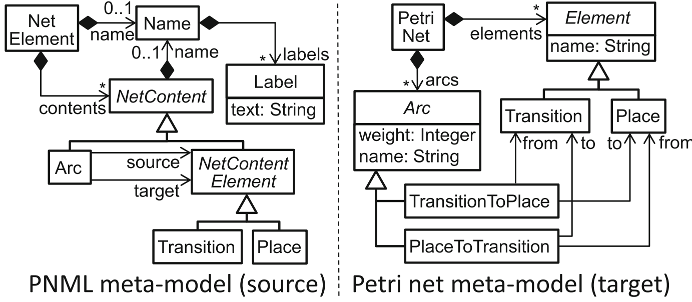


Fig. 1. Source and target meta-models of the running example.

To illustrate this issue we will use an excerpt of the PNML2PetriNet transformation, from the *Grafcet to PetriNet* scenario in the ATL Zoo[4], slightly modified to show interesting cases. Figure 1 shows the source and target meta-models of the transformation and Listing 1 shows an excerpt of the transformation.

```
1  rule PetriNet {
2    from n : PNML!NetElement
3    to p : PetriNet!PetriNet (
4      elements ← n.contents,
5      arcs ← n.contents→select(e | e.oclIsKindOf
         (PNML!Arc))
6    )
7  }
8
9  rule Place {
10   from n : PNML!Place
11   to  p : PetriNet!Place ( ... )
12 }
13
14 rule Transition {
15   from n : PNML!Transition
16   to  p : PetriNet!Transition ( ... )
17 }
18
19 rule PlaceToTransition {
20   from n : PNML!Arc (
21     n.source.oclIsKindOf(PNML!Place) and
22     n.target.oclIsKindOf(PNML!Transition)
23   )
24   to p : PetriNet!PlaceToTransition (
25     "from" ← n.source,
26     "to" ← n.target
27   )
28 }
29
30 rule TransitionToPlace {
31   from n : PNML!Arc (
32     −− The developer forgets to add n.source.
         oclIsKindOf(PNML!Transition)
33     n.target.oclIsKindOf(PNML!Place)
34   )
35   to p : PetriNet!TransitionToPlace (
36     "from" ← n.source, −− Problem here, n.
         source could be a Place
37     "to" ← n.target
38   )
39 }
```

Listing 1. Excerpt of the PNML2PetriNet ATL transformation.

[4] http://www.eclipse.org/atl/atlTransformations/.

Consider the bug introduced in line 32 due to a missing check in the filter which enables the assignment of a Place object to a property of type Transition. A valid fix would be to extend the rule filter with not n.source.oclIsKindOf(PNML!Place). This is, in fact, what anATLyzer generates by default since it just uses the typing of the from ← n.source binding (line 36) to deduce a valid fix. However, a simpler and more idiomatic expression would be n.source.oclIsKindOf(PNML!Transition).

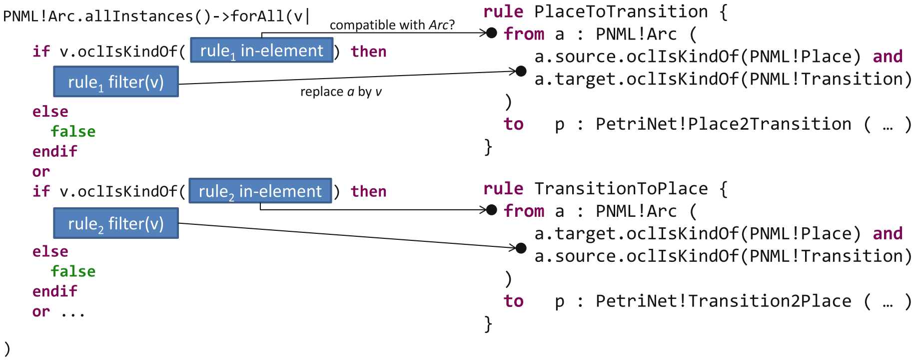


Fig. 2. Schema for constraint generation.

Once fixed, we could be interested in generating a meta-model constraint for PNML to rule out invalid arcs (e.g., an arc whose source and target references point both to places (or both to transitions)). The implementation of quick fixes in anATLyzer will generate a constraint like the one shown in Listing 2. The constraint is generated in the most general way, not taking into account the possible optimizations that could be made.

```
1 Arc.allInstances()→forAll(v1 |
2   if v1.oclIsKindOf(Arc) then
3     v1.source.oclIsKindOf(Transition) and v1.target.oclIsKindOf(Place)
4   else false endif or
5   if v1.oclIsKindOf(Arc) then
6     v1.source.oclIsKindOf(Place) and v1.target.oclIsKindOf(Transition)
7   else false endif
```

Listing 2. Automatically generated invariant to rule out invalid arcs in a Petri net

In general, a synthesizer uses a template and tries to fill the holes using some automated procedure. Figure 2 shows the schema to generate pre-conditions used in anATLyzer. To generate a constraint that will be attached to the PNML meta-model, our system would identify all rules dealing with Arc elements, that is, any rule whose input pattern has Arc or one of its subtypes (if any). Then, it would merge rule filters replacing occurrences of the a variable defined in the input pattern of the rules with the iterator variable v. Please note that this schema based on “if-then-else” is cumbersome, but it is necessary because there is no short-circuit in OCL and therefore a simple and expression would not work in

the general case. Hence, our goal is to simplify these kind of expressions into more idiomatic code, as shown in the Listing 3.

```
1 Arc.allInstances()→forAll(v1 |
2   (v1.source.oclIsKindOf(Transition) and v1.target.oclIsKindOf(Place)) or
3   (v1.source.oclIsKindOf(Place) and v1.target.oclIsKindOf(Transition))
```

Listing 3. Simplified invariant to rule out invalid arcs in a Petri net

In the rest of this paper we present our approach to beautify OCL code, in particular targeting automatically synthesized OCL code. As we will see, it is expected that such code has unnecessary complexity, contains repetitive expressions and it is many times difficult to read. The next section describes the framework and the following presents the current catalogue.

4 Framework

This section describes the design of the reusable component to simplify OCL expressions. We have designed the catalogue of simplifications as a set of reusable transformations using the notion of concept-based transformation components [8]. Our aim is to deal with the fact that there are several implementations of OCL which could be benefited from automatic simplifications. In the EMF ecosystem, we can find the standard OCL distribution (EMF/OCL), SimpleOCL, OCL embedded in the ATL language, the OCL variant of Epsilon, etc. These implementations are incompatible among each other, due to a number of reasons, including different representations of the abstract syntax tree, questions related to the integration of OCL in another language, different OCL versions, different supported and unsupported features (e.g., closure operation is not supported in ATL), access to typing information, etc.

4.1 Overview

Our framework is based on the notion of *concept* and generic transformation component. A concept is a description of the structural requirements that a meta-model needs to fulfil to allow the instantiation of the component with a concrete meta-model (e.g., a particular OCL implementation in this case). A generic transformation component consists of a transformation template and one or more concepts. To instantiate the component for a specific meta-model, a binding describing the correspondences between the meta-model and the concept is written, which in turn induces an adaptation of the template to make it compatible with the meta-model. Please note that the approach of rewriting the original transformation is more adequate than using a pivot meta-model plus a transformation since the OCL expressions need to be modified in-place.

Figure 3 shows the architecture of the solution, which is technically implemented using the facilities provided by bentō to develop reusable transformation components [10]. In this design the simplification component has a set of small transformations, each one targeting only one kind of simplification. On

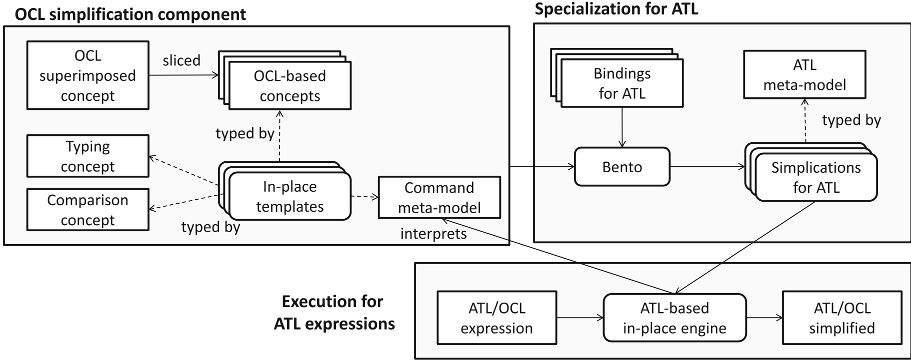


Fig. 3. Architecture of the generic component and its application to simplify ATL/OCL expressions.

the contrary to previous approaches which assumed one concept per transformation [8,27], in this work all transformations share a common OCL-based concept plus two additional concepts to enable parameterized access to type information and expression comparison facilities (see Sect. 4.2). The output is a set of rewriting commands, which will be interpreted by a custom in-place engine. Given a specific OCL implementation for which we want to reuse the simplification component, we must implement a binding between the concrete OCL meta-model and the OCL concept meta-model. The binding establishes the correspondences between the concrete language meta-model (ATL/OCL in the figure) and the OCL concept. The bentō tool takes the binding and the component and derives a new simplification component specialised for ATL. This is fed into the in-place engine to apply the simplifications to concrete ATL expressions.

4.2 Transformation Templates

We use ATL as our implementation language to develop the transformation templates. The in-place mode of ATL is quite limited, and it is not adequate to perform the rewritings required to implement our catalogue. Thus, we extended the in-place capabilities of ATL by creating a simple command meta-model to represent rewriting actions, which is later interpreted by a custom in-place engine. The rationale of choosing ATL despite of its limitations for in-place transformations is due to practical matters. First, we wanted to reuse the infrastructure provided by bentō, which currently supports ATL as the language to develop templates. Secondly, given the motivation of integrating the simplifications within anATLyzer it seems logical to use ATL to avoid extra dependencies. Finally, Henshin [2] was also considered but developing rewritings like the ones of this work is not very natural either, since one needs to specify every possible container type of an expression that is going to be replaced, so that the replacement action can be "statically" computed. Other languages like VIATRA [30] or EOL [18] have action languages which are imperative which complicates its integration with

bentō (i.e., it is not a declarative language and thus is difficult to write a HOT for it, which is the main mechanism used by bentō to adapt transformations).

Listing 4 shows a simplification rule written in ATL. An if expression like `if true then thenExp else elseExp endif` is rewritten to `thenExp`. The execution of the rule creates a `Replace` command which indicates which element (`source`) needs to be substituted by which element (`target`).

```
1   helper context OCL!OclExpression def: isTrue() : Boolean = false;
2   helper context OCL!BooleanExp    def: isTrue() : Boolean = self.booleanSymbol;
3
4   rule removeIf {
5     from o : OCL!IfExp ( o.condition.isTrue() )
6     to a : ACT!Replace
7     do {
8       a.source ← o;
9       a.target ← o.thenExpression;
10    }
11  }
```

Listing 4. Simplification rule

After the execution of the transformation our in-place transformation engine interprets and applies replacement commands over the source model. If there are no applicable actions, another transformation of the catalogue is tried. Thus, the in-place engine works by executing transformations and evaluating commands using an iterative, as-long-as-possible algorithm. We support commands for replacing elements, cloning and modifying pieces of abstract syntax tree and setting properties. This simple approach is enough for our implementation needs. All the transformations are executed in a pre-defined order, and termination has to guaranteed by ensuring that the generated commands only reduce the given expression. In this sense, it is possible to extend anATLyzer to enforce this property, which is part of our future work.

4.3 Concept Design

The transformation template is a regular ATL transformation, typed against Ecore meta-models which act as transformation concepts. A key element in a generic component is the design of such concepts. Our framework requires three concepts, which are depicted in Fig. 4. The *OCL concept* represents the elements of the OCL language which will be subject to simplifications. The *Typing* concept provides a mechanism to access typing information for OCL expressions, whereas the *Comparison* concept provides a way to determine if two expressions are equal. These latter two concepts are hybrid concepts, as defined in [16], since they provide hook methods which will be implemented by each specialization.

OCL Concept. A concept should contain only the elements required by the transformation template. This is intended to facilitate its binding when it is going to be reused and to remove unnecessary complexity from the transformation template implementation. However, if we strictly use this approach to implement the catalogue, we would have many transformations whose concepts have

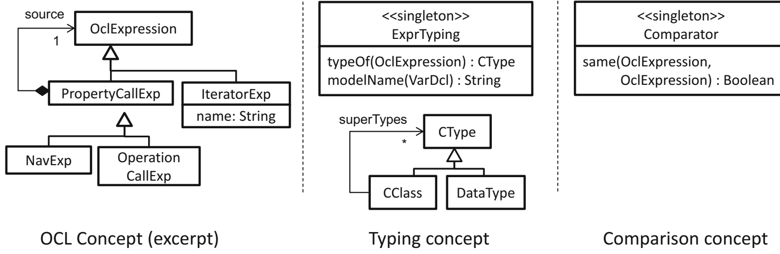


Fig. 4. Concepts used in the simplification component.

many shared elements. For example, all simplification transformations which use operators would need to define a new OperatorCallExp class. It is thus impractical to build each concept separately. Moreover, it would require to have as many bindings as reused transformations. Therefore, we have designed a superimposed concept which contains all the elements required by the transformations of the catalogue. From this concept we automatically extract the minimal concept of each transformation using the approach described in [9], so that each individual rewriting could be used in isolation if needed. Please note that the superimposed OCL concept (i.e., it merges all the concepts used by the individual rewritings) do not necessarily need to be exactly like the OCL specification, but it may have less elements which are not handled by the simplifications (e.g., the property name in a navigation expression is irrelevant, while the name of an iterator is important). The OCL concept currently implemented contains only 20 classes and 27 features. This is much smaller than the 85 classes of the ATL meta-model and the 54 classes of the EMF/OCL meta-model.

Typing Concept. There are a number of transformations in the catalogue which require access to the types of the abstract syntax of the OCL expression. One alternative would be to extend the OCL concept with elements to represent typing information. However, this approach is not flexible enough since it assumes that concrete OCL meta-models have their expressions annotated with types. An alternative design is to have a separate concept with operations to retrieve the typing information. Each concrete binding is in charge of providing access the typing information computed by underlying OCL type checker. This design is, to some extent, similar to the idea of mirrors [3].

Comparison Concept. The comparison concept is also a hybrid concept, but it addresses the problem of comparing two OCL expressions to determine if they are equivalent. The concept does not prescribe any mechanism to compare the expressions, but the implementations may decide to use simple approaches (e.g., comparing string serializations) or more complex ones (e.g., clone detection). The only requirement is that it must be reliable, in the sense that it cannot be heuristic.

5 Catalogue

This section describes through examples the most relevant simplifications currently implemented in the catalogue. The catalogue has been created based on the author's experience building anATLyzer, but it can be easily extended as new needs arise from other tools.

5.1 Literal Simplifications

This set of simplifications replaces operations over primitive values by their results. For instance, an operation like $1 < 0$ is replaced by $false$ by applying a constant folding simplification. This category also covers the simplification of collection expressions like Set { Set { 1 } }→flatten() ⇒ Set{1}.

It is worth noting that this kind of expressions will be rarely written by a developer, but are likely to appear in synthesized OCL code, hence the need for the catalogue in this setting.

5.2 Iterators

This set of simplifications deals with iterator expressions which can be removed or whose result can be computed at compile time. The following listing shows the three simplifications implemented up to now. The simplifications for select also apply to reject just by swapping the behaviour of *True select* and *False select*.

Original

```
-- Unnecessary collect
Place.allInstances()→collect(p | p)→select(p | ...)
-- True select
Place.allInstances()→select(p | true)→collect(p | ...)
-- True forAll
Place.allInstances()→forAll(p | true)
```

Simplified

```
-- Unnecessary collect
Place.allInstances()→select(p | ...)
-- True select
Place.allInstances()→collect(p | ...)
-- True forAll
true
```

5.3 Noisy Let Expressions

Let expressions are useful when a large expression is used many times, otherwise it tends to introduce unnecessary noise. This simplification takes into account the size of the assigned expression and the number of usages in order to remove such let expressions. The following listing shows an example.

Original

```
let src = arc.source in
let tgt = arc.target in
  src.oclIsKindOf(PNML!Place) and
  tgt.oclIsKindOf(PNML!Transition)
```

Simplified

```
arc.source.oclIsKindOf(PNML!Place) and
arc.target.oclIsKindOf(PNML!Transition)
```

The main concern with this simplification is that it may break well-crafted code, when the developer intended to organize a set of logical steps into let variables. Thus, the simplification should only be applied for synthesized code which is known to generate repetitive let expressions.

5.4 Type Comparison Simplifications

These simplifications are aimed at removing unnecessary type comparisons using *oclIsKindOf*/ *oclIsTypeOf* or to simplify a complex chain of type comparisons into a simpler one.

Remove If-Then Type Comparison. An example of this simplification has been shown in Fig. 2 and Listing 3. If the condition *cond* of an if expression is a single type comparison in the form *expr.oclIsKindOf(T)* we check if *typeOf(expr)* = *T*. In such case, we can safely replace the whole expression with *true*, which may enable other simplifications (see for instance *if-else elimination* below).

Full Subclass Checking to Supertype. This simplification takes a chain of *or* expressions in which each subexpression checks the type over the same variable and tries to simplify it to a unique type check over a common supertype.

For example, the listing below (left) is intended to rule out arcs from the contents reference. This simplification recognizes that all subtypes of NetContentElement are checked, and the simpler n.oclIsKindOf(PNML!NetContentElement) can be used instead.

Original

```
aPetriNet.contents→select(n |
  n.oclIsKindOf(PNML!Place) or
  n.oclIsKindOf(PNML!Transition))
```

Simplified

```
aPetriNet.contents→select(n |
   n.oclIsKindOf(PNML!NetContentElement))
```

The application condition of this simplification is relatively complex to implement, hence the advantage of implementing it in a reusable module. A binary operator must be composed by only "or" sub-expressions and each subexpression must apply an oclIsKindOf operator to the same source expression. Then, we extract the set of types used as arguments of the oclIsKindOf operations (*types*). From this set we obtain the most general common supertype (*sup*) of all of these classes (if any), with the constraint that all subclasses of such supertype are "covered" by the classes in *types*, that is the following OCL constraint must be satisfied: sup.allSubclasses→forAll(sub | types→forAll(c | c = sub or c.superTypes→includes(sub))).

In the example, the most general supertype satisfying this constraint is NetContentElement. This is so because its set of subtypes is completely covered by Transition and Place. In contrast, NetContent is not a valid result because Arc is not in the set of types compared by the expression. One concern with this simplification is that for some cases explicitly checking the subtypes could be more readable than the simplified code, since it evokes more clearly the vocabulary of the transformation meta-model. An alternative is to parameterize the simplification with a threshold indicating the minimum number of "oclIsKindOf checks" that need to exists in the original code to trigger it.

5.5 Unshort-Circuiting

OCL does not have short circuit for boolean expressions. Thus, automatic synthesis procedures need to take special care to produce safe boolean expressions.

For instance, the expression arc.source.oclIsKindOf(PNML!Place) and arc.source.tokens is unsafe because the tokens feature will be accessed regardless of the result of the first type comparison (i.e., if arc.source is a Transition). Hence, a runtime error will be raised. The usual solution is to write nested ifs, one for each boolean sub-expression, which typically leads to unreadable code. In the case of synthesized code the situation is exacerbated since it is likely that the synthesizer implementation always generate nested ifs to stay on the safe side.

For example, the following listing (left) shows a piece of code in which short circuit evaluation is not actually necessary because the name feature is defined in a superclass of Place. Therefore, it can be simplified as shown in the right part of the listing.

Original

```
if arc.source.oclIsKindOf(PNML!Place) then
  if arc.source.name <> OclUndefined then
    'plc' + arc.source.name
  else
    'no−name'
  endif
else
  'no−name'
endif
```

Simplified

```
if arc.source.oclIsKindOf(PNML!Place) and
   arc.source.name <> OclUndefined then
   'plc' + arc.source.name
else
  'no−name'
endif
```

Please note that this simplification makes use of the Comparison concept to be able reason more accurately about what can be simplified and the Typing concept to check typing correctness. Another variants of this simplification can also be implemented, for instance for checking OclUndefined conditions.

5.6 Conditionals

Remove Dead If/Else Branch. This is the most basic simplification of conditionals. Given a true or a false literal in the condition, the corresponding then or else parts are used to replace the conditional in the AST. For instance, if true then 'a' else 'b' endif can be simplified to 'a'.

Remove Equals Condition and Then Expression. A simple but useful simplification is recognizing that the condition and the "then" branch (or else branch) of an if expression are the same, and thus they always yield the same result.

Original

```
if place.tokens→size() = 1 then
  place.tokens→size() = 1
else
  false
endif
```

Simplified

```
place.tokens→size() = 1
−− If the else branch of the original expression
−− is true, then whole expression can be
−− replaced by true
```

If Fusion. This simplification takes a binary operation between the results of two if expressions whose conditions are the same. In this case, it is safe to inline the then and else branches of the second expressions in the first one, as in the following example:

Original

```
if elem.oclIsKindOf(PN!Place) then
  elem.tokens→size() > 1
else false endif
and
if elem.oclIsKindOf(PN!Place) then
  not elem.name.oclIsUndefined()
else true endif
```

Simplified

```
if elem.oclIsKindOf(PN!Place) then
  elem.tokens→size() > 1 and
  not elem.name.oclIsUndefined()
else
  false and true
endif
```

The simplified version is more concise, and at the same time enables more simplification opportunities.

6 Evaluation

This section reports the evaluation of our approach. We have evaluated whether the simplifications are able to reduce the complexity of expressions synthesized automatically (usefulness) and to what extent it is possible to reuse the catalogue (reusability) for different OCL dialects, reflecting on the advantages and limitations of the approach.

6.1 Usefulness

We have applied the simplifications of the catalogue to two different kinds of automatically generated OCL constraints, both for the ATL variant of OCL. The first experiment consisted on simplifying OCL preconditions generated from target invariants of model transformations as described in [14]. We simplified 24 constraints coming from invariants defined in three transformations used by existing literature HSM2FSM, ER2REL and Factories2PetriNets. The second experiment applied the simplifications to the quick fixes generated by anATLyzer for the 100 transformations of the ATL Zoo, focussing on those quick fixes which generate rule filters, binding filters or pre-conditions since they are the most interesting in terms of complexity of the generated expressions. Table 1 summarizes the results of the experiments. The complete data, and the scripts and instructions to reproduce the experiments are available at the following URL: http://sanchezcuadrado.es/exp/beautyocl-ecmfa18.

For the preconditions, a total of 440 simplifications were applied to 24 expressions. In average, 18.3 simplifications were applied for each expression, however the median was 5 simplifications. This is because some expressions were particularly large and involved more simplifications. For instance, two of the expressions had more than 3000 nodes, which enabled the application of more than 150 simplifications for each one. In the quick fixes experiment a total of 6562 simplifications were applied to 1729 expressions. We express the simplification power of the catalogue (shown in the "% nodes removed by simplifications" row) by counting the number of nodes of the AST before and after the simplifications. In both experiments the obtained reduction is similar, around 20% in average and 16% in the median.

Table 1. Summary of the results of the experiments.

	Pre-conditions				Quick fixes			
	#Simp.	%	Avg.	Median	#Simp.	%	Avg.	Median
Literals	6	1.4%	-	-	2397	36.5%	-	-
Iterators	6	1.4%	-	-	712	10.6%	-	-
Noisy let	0	0.0%	-	-	177	2.7%	-	-
Type comparison	38	8.6%	-	-	31	0.5%	-	-
Unshort-circuiting	362	82.3%	-	-	63	0.9%	-	-
Conditionals	28	6.6%	-	-	3182	48.5%	-	-
Total simplifications	404	100%	18.3	5	6562	100%	3.8	2
% of nodes removed by simplifications			19.3%	16.8%			33.5%	15.8%

Regarding which simplification categories are more useful, the results are disparate. Some simplifications occur much more often in one experiment than in the other. For instance, simplifications for literals and conditionals are very useful for quick fixes, whereas unshort-circuiting is more useful for pre-conditions. This suggests that simplifications are to some extent specific to the kind of generated code and the method used to generate such code.

At first glance some of the simplifications are quite simple, others are most complex (e.g., those based on the typing and comparison concepts). Combining all of them, the user gets a much better experience. For instance, the following listing shows the situation before and after the use of BeautyOCL. The simplifications applied has been the following: (1) replacing the oclIsKindOf operation by true, then (2) replacing the if expression by its condition and finally (3) simplifying the remaining `expr and expr` by `expr` where `expr = not i.hasLiteralValue.oclIsUndefined()`. As can be observed the result is much more readable. In other evaluated expressions the results are not so "beauty", but the user would expect an even simpler expression. Nevertheless, the results are very promising, and it is expected to have very good results as the catalogue grows.

Original

```
if not i.hasLiteralValue.oclIsUndefined() then -- #2
  i.hasLiteralValue.oclIsKindOf(RDM!Literal) -- #1
else
  false
endif
and not i.hasLiteralValue.oclIsUndefined() -- #3
```

Simplified

```
not i.hasLiteralValue.oclIsUndefined()
```

The catalogue instantiated for ATL has been integrated into anATLyzer through a dedicated extension point, so that the generated quick fixes are automatically simplified. Moreover, a quick assist to let the user simplify a piece of expression on demand is also available. A screencast demonstrating this feature in more detail is available at https://anatlyzer.github.io/screencasts/.

Regarding threats to validity, the main threat to the internal validity of these experiments is that we have only used code synthesized by AnATLyzer. The main reason is the lack of availability of similar tools for other OCL variants. Another issue is that we use the number of nodes to measure the improvement of an expression after simplifications. This metric can be misleading sometimes. For instance, the removal of let expressions generates a simpler expression, but it can introduce a few more nodes. A controlled experiment with final users is required to effectively assess this question. A threat to the external validity is the number of OCL variants reused. Variants like Epsilon or USE are not considered due to not using Ecore meta-models. This is so because our simplifications work at the abstract syntax level, specified with Ecore. Please note that many of them are complex transformations (e.g., use type information or compare sub-expressions) which cannot be addressed with text-based transformations.

6.2 Reusability

The catalogue of simplifications has been designed with reusability in mind in order to be able to easily instantiate the catalogue for a specific OCL variant. To assess to what extent this is possible we have instantiated the component for ATL/OCL, EMF/OCL and SimpleOCL.

The catalogue was first tested and debugged by writing a binding to ATL. The binding was relatively straightforward. The binding for EMF/OCL was also simple except for one important issue. The designed concept expects that an OperatorExp has a name to identify the concrete operator. However, in EMF/OCL an operation is identified by a pointer to an EOperation defined in the standard OCL meta-model. Our binding for the target model (i.e., to support the creation of operator expressions) is not powerful enough to handle this natively. The solution to overcome this has been to extend the typing concept with a "setOperation" so that it is possible to programmatically find and assign the proper EOperation if needed. For SimpleOCL the main limitation is that it does not compute any typing information, and thus we could not reuse those simplifications making use of the typing concept. This means that the instantiated catalogue for SimpleOCL needs to be smaller.

Regarding the size of the implementations, the ATL transformation templates consists of 791 SLOCs, whereas the bindings for ATL, EMF/OCL and SimpleOCL are 38, 49 and 48 SLOCs respectively. The bindings are relatively simple mappings specifications. These figures provide some evidence of the advantage of building transformations as reusable components.

Altogether, the catalogue has proved useful to optimise OCL expressions in terms of their size, thus having simpler and perhaps more beautiful expressions. The effort invested in the creation of the catalogue is amortized by allowing multiple instantiations. Moreover, this work is also non-trivial case study of the application of genericity techniques to model transformations, which can be a baseline to improve these techniques.

7 Conclusions

In this paper we have presented a catalogue of OCL simplifications for OCL expressions, which targets code which has been automatically generated. This catalogue has been implemented as a generic transformation component, with the aim of making it applicable to any OCL variant based on Ecore. The current implementation fully supports ATL and has also been partially instantiated for EMF/OCL and SimpleOCL. The evaluation shows that the proposed simplifications are useful and they can generally reduce the size of the expressions around 30%. As future works we plan to add new simplifications to the catalogue in order to be able to reduce generated expressions by anATLyzer even more. Also, we would like to extend bentō to allow using rewriting languages like Stratego [4] to develop the transformation templates for the generic transformation components. Another line of work is to reflect on how to optimise other kinds of MDE artefacts generated automatically, like models or meta-models.

Acknowledgements. Work funded by the Spanish MINECO TIN2015-73968-JIN (AEI/FEDER/UE).

References

1. Ackermann, J., Turowski, K.: A library of OCL specification patterns for behavioral specification of software components. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 255–269. Springer, Heidelberg (2006). https://doi.org/10.1007/11767138_18
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_9
3. Bracha, G., Ungar, D.: OOPSLA 2004: mirrors: design principles for meta-level facilities of object-oriented programming languages. ACM SIGPLAN Not. **50**(8), 35–48 (2015)
4. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. **72**(1–2), 52–70 (2008)
5. Cabot, J., Teniente, E.: Transformation techniques for OCL constraints. Sci. Comput. Program. **68**(3), 179–195 (2007)
6. Correa, A., Werner, C.: Refactoring object constraint language specifications. Softw. Syst. Model. **6**(2), 113–138 (2007)
7. Correa, A., Werner, C., Barros, M.: An empirical study of the impact of OCL smells and refactorings on the understandability of OCL specifications. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 76–90. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_6
8. Cuadrado, J.S., Guerra, E., de Lara, J.: A component model for model transformations. IEEE Trans. Softw. Eng. **40**(11), 1042–1060 (2014)

9. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Reverse engineering of model transformations for reusability. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 186–201. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_14
10. Cuadrado, J.S., Guerra, E., de Lara, J.: Reusable model transformation components with bentō. In: Kolovos, D., Wimmer, M. (eds.) ICMT 2015. LNCS, vol. 9152, pp. 59–65. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21155-8_5
11. Cuadrado, J.S., Guerra, E., de Lara, J.: Quick fixing ATL transformations with speculative analysis. Softw. Syst. Model. 1–35 (2016)
12. Cuadrado, J.S., Guerra, E., de Lara, J.: Static analysis of model transformations. IEEE Trans. Softw. Eng. **43**(9), 868–897 (2017)
13. Cuadrado, J.S., Guerra, E., de Lara, J.: AnATLyzer: an advanced IDE for ATL model transformations. In: 40th International Conference on Software Engineering (ICSE). ACM/IEEE (2018)
14. Cuadrado, J.S., Guerra, E., de Lara, J., Clarisó, R., Cabot, J.: Translating target to source constraints in model-to-model transformations. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 12–22. IEEE (2017)
15. Sánchez Cuadrado, J., Jouault, F., García Molina, J., Bézivin, J.: Optimization patterns for OCL-based model transformations. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 273–284. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01648-6_29
16. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. Softw. Syst. Mod. **12**(3), 453–474 (2013)
17. Eclipse Modelling Framework. https://www.eclipse.org/modeling/emf/
18. Epsilon. http://www.eclipse.org/gmt/epsilon
19. Giese, M., Larsson, D.: Simplifying transformations of OCL constraints. In: Briand, L., Williams, C. (eds.) MODELS 2005. LNCS, vol. 3713, pp. 309–323. Springer, Heidelberg (2005). https://doi.org/10.1007/11557432_23
20. Hassam, K., Sadou, S., Le Gloahec, V., Fleurquin, R.: Assistance system for OCL constraints adaptation during metamodel evolution. In: 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), pp. 151–160. IEEE (2011)
21. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008). http://www.emn.fr/z-info/atlanmod/index.php/Main_Page. Accessed Nov 2010
22. Loveman, D.B.: Program improvement by source-to-source transformation. J. ACM (JACM) **24**(1), 121–145 (1977)
23. Monperrus, M.: A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In: Proceedings of the 36th International Conference on Software Engineering, pp. 234–242. ACM (2014)
24. Mottu, J.-M., Simula, S.S., Cadavid, J., Baudry, B.: Discovering model transformation pre-conditions using automatically generated test models. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 88–99. IEEE (2015)
25. OMG. Object Constraint Language (OCL) (2014). http://www.omg.org/spec/OCL/2.4/PDF
26. Reimann, J., Wilke, C., Demuth, B., Muck, M., Aßmann, U.: Tool supported OCL refactoring catalogue. In: Proceedings of the 12th Workshop on OCL and Textual Modelling, Innsbruck, Austria, pp. 7–12, 30 September 2012

27. Rose, L., Guerra, E., De Lara, J., Etien, A., Kolovos, D., Paige, R.: Genericity for model management operations. Softw. Syst. Model. **12**(1), 201–219 (2013)
28. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. ACM Sigplan Not. **41**(11), 404–415 (2006)
29. Tibermacine, C., Sadou, S., Dony, C., Fabresse, L.: Component-based specification of software architecture constraints. In: Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering, pp. 31–40. ACM (2011)
30. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. Softw. Syst. Model. **15**(3), 609–629 (2016)
31. Wagelaar, D.: Simpleocl. https://github.com/dwagelaar/simpleocl
32. Wimmer, M., Perez, S.M., Jouault, F., Cabot, J.: A catalogue of refactorings for model-to-model transformations. J. Object Technol. **11**(2), 1–40 (2012)

Expressing Measurement Uncertainty in OCL/UML Datatypes

Manuel F. Bertoa[1], Nathalie Moreno[1], Gala Barquero[1], Loli Burgueño[1], Javier Troya[2], and Antonio Vallecillo[1](✉)

[1] Universidad de Málaga, Málaga, Spain
{bertoa,moreno,gala,loli,av}@lcc.uma.es
[2] Universidad de Sevilla, Sevilla, Spain
jtroya@us.es

Abstract. Uncertainty is an inherent property of any measure or estimation performed in any physical setting, and therefore it needs to be considered when modeling systems that manage real data. Although several modeling languages permit the representation of measurement uncertainty for describing certain system attributes, these aspects are not normally incorporated into their type systems. Thus, operating with uncertain values and propagating uncertainty are normally cumbersome processes, difficult to achieve at the model level. This paper proposes an extension of OCL and UML datatypes to incorporate data uncertainty coming from physical measurements or user estimations into the models, along with the set of operations defined for the values of these types.

1 Introduction

It has been claimed that the expressiveness of a model is at least as important as the formality of its expression [19]. This expressiveness is determined by the suitability of the language for describing the concepts of the problem domain or for implementing the design. While in software engineering there exists a variety of modeling languages tailored at addressing different problems, they may not be well suited for capturing some key aspects of the real world [3,17,27], and in particular for managing data uncertainty in a natural manner. In this respect, the emergence of Cyber-Physical Systems (CPS) [3] and the Internet of Things (IoT), as examples of systems that have to interact with the physical world, has made evident the need to faithfully represent some extra-functional properties of the modeled systems and their elements, as well as to overcome current limitations of existing modeling languages and tools.

One aspect of particular relevance is related to the *uncertainty* of the attribute values of the modeled elements, specially when dealing with certain *quality characteristics* such as precision, performance or accuracy. Data uncertainty can come from different reasons, including variability of input variables, numerical errors or approximations of some parameters, observation errors, measurement errors, or simply lack of knowledge of the true behavior of the system or its underlying physics [12]. On other occasions estimations are needed

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 46–62, 2018.
https://doi.org/10.1007/978-3-319-92997-2_4

because the exact values cannot be obtained since the associated properties are not directly measurable or accessible, values are too costly to measure, or simply because they are unknown.

In a previous paper [28] we presented an extension of the OCL/UML datatype `Real` to deal with measurement uncertainty of numerical values, by incorporating their associated uncertainty [12,13]. However, we soon realized that this was not enough: data uncertainty rapidly extends to all OCL/UML datatypes since it is not just a matter of propagating the uncertainty through the arithmetical operations, but also of dealing with the uncertainty when we compare two uncertain numbers, or need to make a decision about a collection of elements. This requires the definition of uncertain Booleans—values that are true or false with a given probability (level of confidence). Similarly, integers should also be endowed with uncertainty, e.g. when they are used to represent timestamps in milliseconds, and we need to deal with imprecise clocks. This extends to collections too (e.g., a `forAll` statement in a set of uncertain values), and to datatypes operations.

This paper shows how measurement uncertainty can be incorporated into OCL [21] primitive data types and their collections (and hence into UML [23], since both languages share the same primitive types), by defining super-types for them, as well as the set of operations defined on the values of these types. Both analytical and approximate algorithms have been developed to implement these operations. We provide a Java library and a native implementation in USE [9,10].

This paper is structured as follows. First, Sect. 2 briefly introduces the concepts related to measurement uncertainty that will be used throughout the paper. Then, Sect. 3 describes our proposal and the algebra of operations on uncertain values and the implementations we have developed for these operations. Section 4 illustrates some usage scenarios and applications of the proposal. Section 5 compares our work to similar proposals. Finally, we conclude the paper in Sect. 6 with an outlook on future work.

2 Background

Uncertainty is the quality or state that involves imperfect and/or unknown information. It applies to predictions of future events, estimations, physical measurements, or unknown properties of a system [12].

Measurement uncertainty is the special kind of uncertainty that normally affects model elements that represent properties of physical elements. It is defined by the ISO VIM [14] as "a parameter, associated with the result of a measurement, that characterizes the dispersion of the values that could reasonably be attributed to the measurand."

The Guide to the Expression of Uncertainty in Measurement (GUM) [12] defines measurement uncertainty for `Real` numbers representing values of attributes of physical entities, and states that they cannot be complete without an expression of their uncertainty. Such an uncertainty is given by a *confidence interval*, which can be expressed in terms of the *standard uncertainty*—i.e., the

standard deviation of the measurements for such value. Therefore, a real number x becomes a pair (x, u), also noted $x \pm u$, that represents a random variable X whose average is x and its standard deviation is u. For example, if X follows a normal distribution $N(x, u)$, we know that 68.3% of the values of X will be in the interval $[x - u, x + u]$.

The GUM framework also identifies two ways of evaluating the uncertainty of a measurement, depending on whether the knowledge about the quantity X is inferred from repeated measured values ("Type A evaluation of uncertainty"), or scientific judgment or other information concerning the possible values of the quantity ("Type B evaluation of uncertainty").

In Type A evaluation of uncertainty, if $X = \{x_1, \ldots, x_n\}$ is the set of measured values, then the estimated value x is taken as the mean of these values, and the associated uncertainty u as their *experimental standard deviation*, i.e., $u^2 = \frac{1}{(n-1)} \sum_{i=1}^{n} (x_i - x)^2$ [12]. In Type B evaluation, uncertainty can also be characterized by standard deviations, evaluated from assumed probability distributions based on experience or other information. For example, if we know or assume that the values of X follow a Normal distribution, $N(x, \sigma)$, then we take $u = \sigma$. And if we can only assume a uniform or rectangular distribution of the possible values of X, then x is taken as the midpoint of the interval, $x = (a+b)/2$, and its associated variance as $u^2 = (b-a)^2/12$, and hence $u = (b-a)/(2\sqrt{3})$ [12].

In addition to the measure or estimation of individual attributes, in general we need to combine them to produce an aggregated measure, or to calculate a derived attribute. For example, to compute the area of a rectangle we need to consider its height and its width, combining them by multiplication. The individual uncertainties of the input quantities need to be combined too, to produce the uncertainty of the result. This is known as the *propagation of uncertainty*, or *uncertainty analysis*.

Uncertainty can also apply to Boolean values. For example, in order to implement equality and comparison of numerical values with uncertainty, the traditional values of `true` and `false` returned by boolean operators are no longer enough. They now need to return numbers between 0 and 1 instead, representing the probabilities that one uncertain value is equal, less or greater than other [20]. This leads to the definition of *Uncertain Booleans*, which are Boolean values accompanied by the level of confidence that we assign to them. This is a proper supertype of Boolean and its associated operations. Note that this approach should not be confused with *fuzzy logic*: although both probability and fuzzy logic represent degrees of subjective belief, fuzzy set theory uses the concept of fuzzy set membership, i.e., how much an observation belongs to a vaguely defined set, whilst probability theory uses the concept of subjective probability, i.e., the likelihood of an event or condition [16].

3 Extension of OCL and UML DataTypes

Our goal is to extend the OCL and UML languages by declaring new types able to express uncertainty. The benefits are twofold. First, uncertainty can be expressed

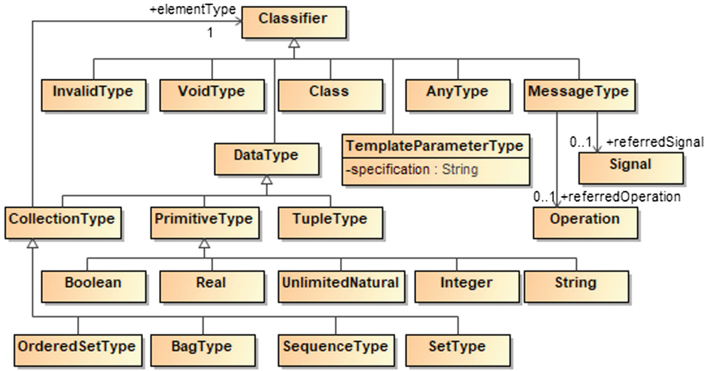


Fig. 1. OCL types, from [21].

in models, i.e., our approach allows the user to define and manipulate uncertainty in a high-level and platform-independent way. Second, information at the model level can be transferred to standard algorithms and tools, so that these can also manage uncertainty by dealing with complex types in their computations.

We propose to extend the OCL types, which are shown in Fig. 1, with uncertainty information. Of course, not all of them need such information, such as types `oclInvalid`, `oclAny`, or `oclVoid`. Other types, such as `Class` and `Tuple`, are user-defined and composed of other heterogeneous types that will convey such information, so there is no need to extend them at this level. Similarly for `TemplateParameter` types, which refer to generic types. Therefore, we need to cover the primitive types (`Real`, `Integer`, `Boolean`, `String`, and `UnlimitedNatural`), collections (`Set`, `Bag`, `OrderedSet`, and `Sequence`) and messages. In this paper we focus on the primitive types, excluding `String`, and on collections. Uncertainty in Strings, Messages and Enumerations—which are datatypes both in OCL and UML—is of different nature, and therefore their extension is left for future work.

3.1 Extension Strategy

In order to extend the OCL/UML primitive types, we apply *subtyping* [18]. We say that type A is a *subtype* of type B (noted $A <: B$), if all elements of A belong to B, and the behavior of operations of B, when applied to elements of A, is the same as those of A [1], i.e., they respect behavioral subtyping [18]. If $A <: B$, then we say that B is a supertype of A.

For instance, `Integer` is a subtype of `Real` because every `Integer` number can be seen as a `Real` number whose decimal part is zero. Besides, `Real` operations, when applied to `Integer` numbers, behave as those of type `Integer`.

Then, for extending a primitive OCL datatype `T` we will define a supertype that incorporates information about the uncertainty of the values of `T`, and

Table 1. New OCL primitive types and their operations.

Type	Operations
`UReal`	$+$, $-$, $*$, $/$, `abs()`, `neg()`, `power()`, `sqrt()`, `inv()`, `floor()`, `round()`, $<$, $\leq$, $>$, $\geq$, $=$, $<>$, `uEquals()`, `uDistinct()`, `min()`, `max()`, `toString()`, `toInteger()`, `toReal()`, `toUInteger()`
`UInteger`	$+$, $-$, $*$, `div`, $/$, `abs()`, `neg()`, `power()`, `sqrt()`, `inv()`, `mod()`, $<$, $\leq$, $>$, $\geq$, $=$, $<>$, `uEquals()`, `uDistinct()`, `min()`, `max()` `toString()`, `toInteger()`, `toUReal()`, `toUInteger()`
`UUnlimitedNatural`	$+$, $*$, `div`, $/$, `mod`, $<$, $\leq$, $>$, $\geq$, $=$, $<>$, `uEquals()`, `uDistinct()`, `min()`, `max()`, `toString()`, `toInteger()`, `toUReal()`, `toUInteger()`
`UBoolean`	`not`, `and`, `or`, `xor`, `implies`, `equivalent`, $=$, $<>$, `equalsC()`, `uEquals()`, `uDistinct()`, `toString()`, `toBoolean()`, `toBooleanC()`

defines the operations for the extended type, which are also applicable to the base type—i.e., the subtype. This uncertainty information will vary depending on whether the values of the base type are numbers (types `Real`, `Integer` and `UnlimitedNatural`) or boolean values. In the first case, the uncertainty information will record *measurement uncertainty*, and will be expressed as specified in the GUM [12]. Thus, numbers of the extended types will be pairs (x, u), with u the associated uncertainty (cf. Sects. 3.2–3.4). Operations will respect the subtyping relationship, ensuring safe-substitutability. In the case of booleans, the uncertainty will be given by means of a real number between 0 and 1 that represents the assigned confidence (cf. Sect. 3.5).

Table 1 shows the newly defined types and their operations. Besides, the subtyping relationships (<:) among the numeric datatypes—both standard and extended—are shown below:

$$\begin{array}{ccccc} \texttt{UnlimitedNatural}\backslash\{*\} & <: & \texttt{Integer} & <: & \texttt{Real} \\ \wedge & & \wedge & & \wedge \\ \texttt{UUnlimitedNatural}\backslash\{*\} & <: & \texttt{UInteger} & <: & \texttt{UReal} \end{array}$$

In addition, `Boolean` <: `UBoolean`, completing the relationships. To extend collections we will specify them using the corresponding extended operations of their element types. The following sections describe these extensions in detail.

3.2 Extending Type Real

To represent real values with measurement uncertainty, we make use of type `UReal` and the algebra of operations defined on the values of that type, which we presented in our previous work [28]. Basically, the values of `UReal` are pairs of `Real` numbers $X = (x, u)$. They determine the expected value (x) and associated standard uncertainty (u) of a quantity X, as defined in Sect. 2. Real numbers x are naturally injected into type `UReal`, corresponding to pairs $(x, 0)$.

We have specified in OCL, and also implemented in Java, all the operations on the values of type `UReal`, to allow modelers to use them for defining derived attributes and for specifying operations and invariants in OCL and UML models. Furthermore, to validate our proposal we have also extended the tool USE by implementing the new types as native ones—see Sect. 3.7.

As an example, the following listing shows the specification of two of the `UReal` operations[1]:

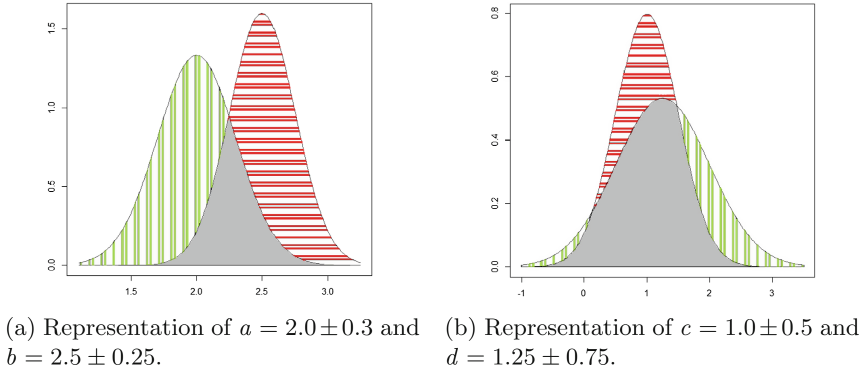
```
context UReal::add(r : UReal) : UReal
post: result.x = self.x + r.x and
      result.u = (self.u*self.u + r.u*r.u).sqrt()
context UReal::mult(r : UReal) : UReal
post: result.x = (self.x*r.x) and
      result.u = (r.u*r.u*self.x*self.x + self.u*self.u*r.x*r.x).sqrt()
```

In addition to the traditional comparison operations between uncertain reals ($<$, $\leq$, $>$, etc.), which return a Boolean value, comparisons between real numbers with uncertainty should return uncertain booleans. To illustrate this need, consider the graphical representation of two pairs of uncertain reals shown in Fig. 2. We can see that there is indeed an overlap (represented by the gray area): it constitutes the probability that the two values are equal.

Then, given two `UReal` values x and y we define three real numbers (l, e, g) that represent, respectively, the probability of x being less, equal or greater than y. Of course, it is always the case that $l+e+g = 1$. For example, the triplet that we obtain for values a and b (Fig. 2a) is the following: $(0.893, 0.106, 1.11 \cdot 10^{-16})$. This means that $a < b$ with probability 0.893; $a = b$ with probability 0.106, and $a > b$ with a probability $1.11 \cdot 10^{-16}$. Similarly, the triplet for c and d (Fig. 2b) is: $(0.152, 0.754, 0.094)$. Note that these 3 numbers correspond to the three areas in which the curve that represents the first of the values can be divided (this is clearer in Fig. 2b).

All this has been specified in OCL using an auxiliary operation on type `UReal` called `calculate(r:UReal)` that returns a tuple with the triplet. With it, the specification of comparison operations between `UReal` numbers is as follows (`lt` and `gt` mean *lower/greater than*, `le` and `ge` mean *lower/greater or equal than*, b and c conform the `UBoolean` type, as explained in Sect. 3.5).

[1] Operations on basic datatypes normally use infix notation (e.g., $x + y$, $a < b$, P `and` Q). This is the notation that we already support in our USE implementation for the newly defined types (`UReal`, `UBoolean`, etc.), see Sect. 3.7. However, other languages that we have used to implement these new types (e.g., Java) do not support infix notation. Therefore, in the following we will use either an infix or prefix notation (`x.add(y)`, `a.lt(b)`, `P.and(Q)`) for the operations of these types, depending on the context and on the particular language used.



(a) Representation of $a = 2.0 \pm 0.3$ and $b = 2.5 \pm 0.25$.

(b) Representation of $c = 1.0 \pm 0.5$ and $d = 1.25 \pm 0.75$.

Fig. 2. Graphical representation of `UReal` values.

```
context UReal::lt(r :UReal) :UBoolean
  post: (result.b) and (result.c = self.calculate(r).l)
context UReal::le(r :UReal) :UBoolean
  post: (result.b) and (result.c=let x:Tuple(l:Real,e:Real,g:Real)=
                             self.calculate(r) in x.l + x.e)
context UReal::gt(r :UReal) :UBoolean
  post: (result.b) and (result.c=self.calculate(r).g)
context UReal::ge(r :UReal) :UBoolean
  post: (result.b) and (result.c=let x:Tuple(l:Real,e:Real,g:Real)=
                             self.calculate(r) in x.g + x.e)
context UReal::uEquals (r :UReal) :UBoolean
  post: (result.b) and (result.c = self.calculate(r).e)
context UReal::uDistinct(r :UReal) :UBoolean
  post: (result.b) and (result.c = 1.0 - self.uEquals(r))
```

The complete OCL specifications of these types and operations, and their implementation in SOIL [4]—an OCL extension that permits the execution of OCL specifications for simulation purposes—is available from [2], together with the two implementations in Java that we provide, depending on whether we assume values are independent and normally distributed—and therefore a closed form expression can be used for the calculations—or using Monte-Carlo simulations in case variables follow arbitrary distributions.

Table 1 shows the set of operations defined for type `UReal`, including conversion operations to other OCL datatypes (both standard and extended).

3.3 Extending Type `Integer`

Type `UInteger` is the supertype of OCL type `Integer` that defines measurement uncertainty. This is needed, for instance, when representing timestamps of events, which are normally expressed in milliseconds, and may have some uncertainty due to lack of clock accuracy.

This extension is straightforward. Every `UInteger` element is of the form (n, u) with n an `Integer` value and u a `Real` value that represents the uncertainty. The injection of any `Integer` value n into type `UInteger` is naturally

defined by $(n, 0)$. In turn, the behavior of UInteger operations is defined by lifting the operation to type UReal, and then projecting the corresponding result, if needed. This, together with the subtyping relationship Integer $<:$ Real existing in OCL, ensures the proper subtyping relationship between Integer and UInteger.

3.4 Extending Type UnlimitedNatural

An OCL UnlimitedNatural is either a non-negative Integer or a special *unlimited* value (*) that represents the upper value of a multiplicity specification [21].

First, we have that UnlimitedNatural$\backslash\{*\}$ $<:$ Integer, that is, excluding value *, unlimited naturals are just non-negative integers. This special value * cannot be used in any arithmetic operation with unlimited naturals, but only with comparison (including max and min) operations. Although subtraction is not defined in OCL for unlimited naturals, it can be naturally defined as a partial operation, and hence lifted to type Integer (and hence to Real).

The extension of UnlimitedNatural to UUnlimitedNatural consists in adding a new component to every unlimited natural value, with the expression of its uncertainty. The uncertainty of special value * will always be 0.

Operations on UUnlimitedNatural values not involving special value * are defined by lifting them to type UInteger. Comparison operations need to consider the particular case of special value * (internally represented by "−1"), lifting the operation to the supertype if this value is not involved. For illustration purposes, the following listing shows the OCL specifications of the comparison operations between UUnlimitedNatural values.

```
uEquals(r : UUnlimitedNatural) : UBoolean
  post: result = if (self.x<>-1) and (r.x<>-1) then
                        self.toUInteger().uEquals(r.toUInteger())
                 else (self.x=-1) and (r.x=-1)
                 endif
lt(r : UUnlimitedNatural) : UBoolean
  post: if (self.x<>-1) and (r.x<>-1) then
             result=self.toUInteger().lt(r.toUInteger())
        else (result.b = ((self.x<>-1) or (r.x=-1))) and (result.c=1.0)
        endif
le(r : UUnlimitedNatural) : UBoolean
  post: result=self.lt(r).or(self.equals(r))

gt(r : UUnlimitedNatural) : UBoolean
  post: result = not self.le(r)

ge(r : UUnlimitedNatural) : UBoolean
  post: result = not self.lt(r)

max(r : UUnlimitedNatural) : UUnlimitedNatural
  post: result = if (self.x=-1) then self
          else if (r.x=-1) then r
             else if r.lt(self).toBoolean() then self else r endif
             endif
          endif
min(r : UUnlimitedNatural) : UUnlimitedNatural
  post: result = if (self.x=-1) then r
          else if (r.x=-1) then self
             else if r.lt(self).toBoolean() then self else r endif
             endif
          endif
```

3.5 Extending Type `Boolean`

Type `UBoolean` is the supertype for type `Boolean` that adds uncertainty to its values. In this case, the uncertainty does not refer to measurement uncertainty, but to confidence. Thus, a `UBoolean` value is a pair (b, c) where b is a boolean value $(true, false)$ and c is a real number in the range $[0 \ldots 1]$, representing the confidence that b is certain. Boolean values `true` and `false` are injected into the supertype as $(\texttt{true}, 1)$ and $(\texttt{false}, 1)$, respectively.

A property of this representation is that $(b, c) = (\neg b, 1 - c)$ for every boolean value b. Then, in its internal representation we will use a canonical form, always taking b the value of *true* and c the corresponding confidence. Using this canonical form, a true value with 95% confidence is represented as $(\texttt{true}, 0.95)$ and a false value with 95% confidence as $(\texttt{true}, 0.05)$.

The operations supported by type `UBoolean` extend those of type `Boolean`, as defined by OCL [21]. We have defined the basic (`not`, `and` and `or`) and secondary operations (`implies`, `equivalent` and `xor`) of the traditional Boolean algebra, extending them with uncertainty. Assuming all values are independent, the following listing shows the specification of all the `UBoolean` type operations.

```
not() : UBoolean
  post: (result.b) and
        (result.c = if self.b then 1-self.c else self.c endif)

and(b : UBoolean) : UBoolean
  post: let C : Real = (self.c * b.c) in (result.b) and
        (result.c=if (self.b and b.b) then C else (1-C) endif)

or(b : UBoolean) : UBoolean
  post: let C : Real = (self.c + b.c - (self.c * b.c)) in
        (result.b) and
        (result.c = if (self.b or b.b) then C else (1-C) endif)

implies(b : UBoolean) : UBoolean
  post: let C : Real = (self.c + b.c - (self.c * b.c)) in
        (result.b) and
        (result.c = if (self.b implies b.b) then C else (1-C) endif)

equivalent(b : UBoolean) : UBoolean
  post: let C : Real = (self.c + b.c - (self.c * b.c)) in
        (result.b) and
        (result.c = if (self.b implies b.b) and (b.b implies self.b)
                    then C else (1-C) endif)

xor(b : UBoolean) : UBoolean
  post: result = self.uEquivalent(b).not()

equals(b : UBoolean) : Boolean = (self.b=b.b) and (self.c=b.c) or
                                 (self.b=not b.b) and (self.c=1-b.c)

equalsC(b : UBoolean, c: Real) : Boolean =
      (self.b=b.b) and ((self.c-b.c).abs()<=1-c)

distinct(b : UBoolean) : Boolean = not (self.equals(b))

toBoolean() : Boolean =
    if (self.c>=0.5) then (self.b) else (not self.b) endif

toBooleanC(c:Real):Boolean =
    if (self.c>=c) then (self.b) else (not self.b) endif
```

We have kept '=' (`equals()`) and '<>' (`distinct()`) operations with their usual semantics, that is, two `UBoolean` elements are the same if their boolean and confidence values match. We have also extended the `equals()` operation with the possibility of indicating a confidence threshold that both `UBoolean` values are equal. Other identity operations (`uEquals()`, `uDistinct()`) compare two `UBoolean` values, returning another `UBoolean`. Finally, some conversion operations allow `UBoolean` values to be converted into `Boolean` values, either approximately, if the confidence is greater than or equal to 0.5, or by indicating a threshold for the confidence.

We have also specified an alternative implementation of these operations, in case no assumption can be made about the independence of the variables in a boolean expression. It is based on the Monte-Carlo simulation method proposed in [13] for Type-A measurement uncertainty in real numbers, adapted to boolean values. Basically, every `UBoolean` value contains a sequence of `Boolean` values that represent the sample obtained when measuring that value. Operations are performed on the samples, and then b and c become just derived values. An excerpt of such specification, showing only the first two operations, is shown in the listing below. Note that an additional invariant, at the end of the listing, requests that all samples should be of the same size.

```
class UBoolean_A
   -- canonical form: triplets (sample[],true,c), with:
   --     sample: the set of measured values obtained for self
   --     c: the confidence that self is true
attributes
   sample : Sequence(Boolean)
   b : Boolean derive: true
   c : Real derive: self.sample->count(true)/self.sample->size()
not() : UBoolean_A
  post: (Sequence{1..self.sample->size}->forAll(i|
           result.sample->at(i)=not self.sample->at(i)))
and(b : UBoolean_A) : UBoolean_A
  post: (Sequence{1..self.sample->size}->forAll(i|
        result.sample->at(i)=(self.sample->at(i) and b.sample->at(i))))
...
context UBoolean_A inv SameSampleSize:
  UBoolean_A.allInstances->forAll(u1,u2|u1.sample->size=u2.sample->size)
```

Similar specifications (and their corresponding implementations in Java) are also available for the rest of the extended types.

3.6 Extending OCL Collections

OCL collections can be easily extended based on the extended operators for primitive datatypes. The following listing shows the specification of all collection operations. Those that return a `UBoolean` value incorporate a '`u`' at the start of their name, to distinguish them from their boolean versions:

```
source->uForAll(e | P(e)) : UBoolean
::= source->iterate(e, acc:UBoolean=UBoolean(true,1) | acc.and(P(e)))

source->uExists(e | P(e)) : UBoolean
::= source->iterate(e, acc:UBoolean=UBoolean(true,0) | acc.or(P(e)))
```

```
source->uIncludes(e) : UBoolean
::= source->iterate(v, acc:UBoolean=UBoolean(true,0) |
        if v.uEquals(e).c > acc.c then v.uEquals(e) else acc endif)

source->uIncludesAll(collection) : UBoolean
::= collection->uForAll(e | source->uIncludes(e))

source->uExcludes(e) : UBoolean
::= source->uForAll(v | v.uEquals(e).not())

source->uExcludesAll(collection) : UBoolean
::= collection->uForAll(e | source->uExcludes(e))

source->uSelect(P():UBoolean) : collection
::= source->iterate(v, acc:collection=collection{} |
        if P(v).toBoolean() then acc->including(v) else acc endif)

source->uSelect(P():UBoolean, c :Real) : collection
::= source->iterate(v, acc:collection=collection{} |
        if P(v).toBooleanC(c) then acc->including(v) else acc endif)

source->uReject(P():UBoolean):collection
::= source->iterate(v, acc:collection=collection{} |
        if P(v).toBoolean() then acc->excluding(v) else acc endif)

source->uReject(P():UBoolean, c:Real) : collection
::= source->iterate(v, acc:collection=collection{} |
    if P(v).toBooleanC(c) then acc->excluding(v) else acc endif)

source->uCount(e) : Integer
::= source->iterate(v, acc:Integer=0 |
        if v.uEquals(e).toBoolean() then acc + 1 else acc endif)

source->uCountC(e,c) : Integer
::= source->iterate(v, acc:Integer=0 |
        if v.uEquals(e).toBooleanC(c) then acc + 1 else acc endif)

source->uOne(P():UBoolean) : Boolean
::= source->uSelect(e | P(e))->size()=1

source->uOneC(P():UBoolean, c:Real) : Boolean
::= source->uSelect(e | P(e).toBooleanC(c) )->size()=1

source->uIsUnique(P():UBoolean) : UBoolean
::= source->uForAll(e|source->uForAll(v|e<>v implies
                                             P(e).uEquals(P(v).not())))
source->sum() : UReal
::= source->iterate(v, acc:UReal=UReal(0,0) | acc.add(v))
```

3.7 Implementation in USE

USE [9] is a modeling tool that allows the validation of OCL and UML models by means of executing the UML models and checking its OCL constraints. The tool is open-source and distributed under a GNU General Public License. To validate our proposal we have extended the OCL/UML language in USE by adding the previously described uncertain types as basic primitive data types, as well as their native operations, so they become available to any OCL/UML modeler. An example of how the new types can be effectively used is illustrated in Sect. 4. The extended tool can be downloaded from our website[2].

[2] http://atenea.lcc.uma.es/downloads/uncertainOCLTypes/use-5.0.0_extended.zip.

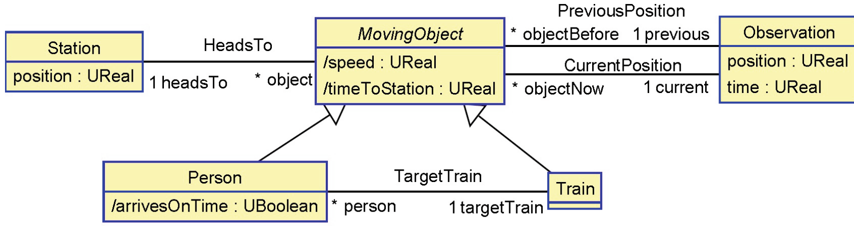


Fig. 3. UML class diagram for train example.

4 Applications

To illustrate our proposal, let us consider the system described by the metamodel shown in Fig. 3. It is composed of people, trains and stations. Both persons and trains move towards stations. For simplicity, we assume they all move in one single direction. Monitors observe their movements, and record their last two positions and the time in which they were observed. The speed is automatically calculated from this information, as well as the expected time to arrive at the station. For a person it is also important to know if she will be able to catch her target train, i.e., reach the station at least 3 s before the train does. All these calculations can be specified by means of OCL expressions:

```
context MovingObject::speed:Real
  derive: (self.current.position-self.previous.position) /
          (self.current.time-self.previous.time)

context MovingObject::timeToStation: Real
  derive: (self.headsTo.position-self.current.position) / self.speed

context Person::arrivesOnTime:Boolean
  derive: (self.timeToStation + 3) <= self.targetTrain.timeToStation
```

Figure 4 shows a UML object diagram with an example of such a system, using conventional UML datatypes `Real` and `Boolean`.

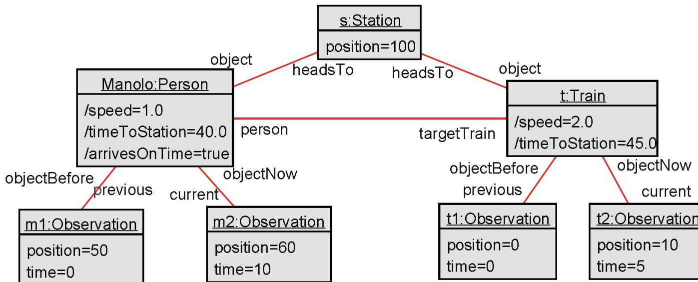


Fig. 4. UML object diagram with `Real` and `Boolean` types.

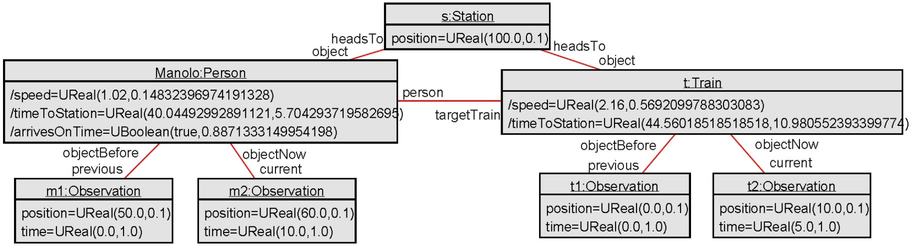


Fig. 5. UML object diagram with uncertain types.

Note, however, that in practice all these attributes and operations are subject to uncertainty: positions and times are never 100% precise, and this imprecision is propagated to derived values and estimated times. For example, suppose our positioning system is correct up to one centimeter, and our clock has a precision of 1 s. This can be captured in our model by simply updating the types of `Real` and `Boolean` variables to `UReal` and `UBoolean`, respectively (this was already showed in Fig. 3).

Figure 5 shows an object diagram with uncertain variables. Those variables take into account the measurement uncertainty in the observations, and propagate it through the computations. We can see how the expected train and user arrival times at the station are $T = 44.560 \pm 10.581$ and $M = 40.045 \pm 5.704$, respectively, and therefore their difference is $T - M = -4.515 \pm 12.374$. Using a `UBoolean` comparison operation, $M \leq T = (true, 0.887)$, which means that the user will be able to arrive on time to catch the train with a probability of 0.887. This is much more realistic than the first model, which probably was too naïve to be of real use. Something worth noticing is that we only had to change the types of the variables; all the OCL expressions that were used to compute the values of derived attributes remained exactly the same.

5 Related Work

The need to represent and manipulate physical values in software models is emerging, in particular units or real-time properties of cyber-physical systems [27]. For example, given that timing values are by nature uncertain (they are very often estimates and/or measured by means of monitoring), the real-time community is used to represent probability distributions and intervals for timing properties, and their influence is evident in the MARTE Profile [22] and SysML [24]. However, neither MARTE nor SysML support operations for performing calculations with these values, they remain at the descriptive level.

Similarly, in [31], the authors propose a conceptual model, called *Uncertum*, which is supported by a UML profile (UUP, the UML Uncertainty Profile)

that enables including uncertainty in test models. Uncertum is based on the U-Model [32], extending it for testing purposes. UUP is a very complete profile that covers all different kinds of uncertainties, in particular measurement uncertainty. Again, their focus, testing, is slightly different from ours, and they only need to represent uncertainty but not to perform operations with it, and therefore they also remain at a descriptive level.

Other works on Business Process Models (e.g., [15]) also consider uncertainty when modeling the arrival time of clients, the availability of some resources or the duration of some tasks. These works use probabilistic mass functions for modeling the values of the corresponding attributes. We have preferred to use the way defined by the GUM [12,13]. Apart from being simpler and widely adopted by other engineering disciplines, it has the main benefit of permitting operations on variables that do not follow any particular probabilistic distribution.

The work in [30] defines an XML-based modeling language for measurement uncertainty evaluation based on the GUM, and a simulation framework for it. This work can be in principle considered closely related to our proposal, but the fact that it is not integrated with the type system of a mainstream modeling language (such as OCL or UML), and its low-level syntax (based on plain XML) hindered its usability. Similarly, the work in [11] defines a datatype that incorporates measurement uncertainty and provides some libraries to perform computations with its values. The integration of these works with OCL/UML models is not straightforward, and therefore their adoption and usage by UML modelers might be limited. To the best of our knowledge these works are more closely related to the mathematical libraries and tools already existing [29] for propagating measurement uncertainty and operating with uncertain values, than to our work.

Other works deal with model uncertainty, but focusing on aspects different from the ones we have described here. For instance, on the uncertainty on the models themselves and on the best models to use depending on the system properties that we want to capture [19]. Other works deal with the uncertainty of the design decisions, of the modeling process, or of the domain being modeled [5–8,26]. We depart from them since we are concerned with the uncertainty of the values of the quantities being measured, which is a different problem.

Finally, the OMG defined the Structured Metrics Meta-model (SMM) [25], which is part of the Architecture Driven Modernization (ADM) effort, and aims at representing measurement information related to software, its operation and design. The SMM is a specification for the definition of measures and the representation of their measurement results, including uncertainty, independently of the representation of the measured entities. In this sense, our proposal can be considered as a refinement of the SMM metamodel, particularizing it to the domain of OCL and UML datatypes.

6 Conclusion and Future Work

In this paper we have focused on representing and managing measurement uncertainty in OCL and UML software models, something required in order

to precisely capture and manipulate some of the essential quality properties of any physical system. We have extended the OCL datatypes and their related operations with uncertainty information. OCL and Java libraries have also been developed to implement the type and its operations in MDE settings. Our implementation is available on [2].

This work opens several interesting lines of research that we would like to explore next. First, we would like to analyze how uncertainty could be added to those OCL datatypes not covered here, namely Strings and Enumerations. As mentioned earlier, the nature of their uncertainty seems to be rather different from the rest. Second, we would like to provide mappings from our high-level OCL/UML specifications to other specification and simulation languages and tools, in particular Modelica and Simulink. The objective is to achieve a stepwise refinement heterogeneous specification and simulation process, whereby high-level specifications (and hence more lightweight) can be progressively refined into more concrete, complete (and more complex) specifications. Finally, we would like to further validate our proposal with different kinds of examples, checking the expressiveness and applicability of this type of specifications.

Acknowledgements. This work has been partially supported by the Spanish Government under Grant TIN2014-52034-R. We would like to thank Martin Gogolla for his help and support during the development of the USE tool extension, and to the reviewers for their constructive comments and very valuable suggestions.

References

1. America, P.: Inheritance and subtyping in a parallel object-oriented language. In: Bézivin, J., Hullot, J.-M., Cointe, P., Lieberman, H. (eds.) ECOOP 1987. LNCS, vol. 276, pp. 234–242. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47891-4_22
2. Bertoa, M.F., Moreno, N., Barquero, G., Burgueño, L., Troya, J., Vallecillo, A.: Uncertain OCL Datatypes, April 2018. http://atenea.lcc.uma.es/projects/UncertainOCLTypes.html
3. Broy, M.: Challenges in modeling cyber-physical systems. In: Proceedings of the ISPN 2013, pp. 5–6. IEEE (2013)
4. Büttner, F., Gogolla, M.: On OCL-based imperative languages. Sci. Comput. Program. **92**, 162–178 (2014)
5. Eramo, R., Pierantonio, A., Rosa, G.: Managing uncertainty in bidirectional model transformations. In: Proceedings of SLE 2015, pp. 49–58. ACM (2015)
6. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 214–238. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_9
7. Famelis, M., Salay, R., Chechik, M.: Partial models: towards modeling and reasoning with uncertainty. In: Proceedings of ICSE 2012, pp. 573–583. IEEE Press (2012)
8. Garlan, D.: Software engineering in an uncertain world. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010), pp. 125–128. ACM (2010)

9. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. Sci. Comp. Prog. **69**, 27–34 (2007)
10. Gogolla, M., Hilken, F.: Model validation and verification options in a contemporary UML and OCL analysis tool. In: Oberweis, A., Reussner, R. (eds.) Proceedings of the Modellierung (MODELLIERUNG 2016). LNI, vol. 254, pp. 203–218. GI (Gesellschaft für Informatik), Karlsruhe (2016)
11. Hall, B.D.: Component interfaces that support measurement uncertainty. Comput. Stand. Interfaces **28**(3), 306–310 (2006)
12. JCGM 100:2008: Evaluation of measurement data - Guide to the expression of uncertainty in measurement (GUM). Joint Committee for Guides in Metrology (2008). http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf
13. JCGM 101:2008: Evaluation of measurement data - Supplement 1 to the "Guide to the expression of uncertainty in measurement" - Propagation of distributions using a Monte Carlo method. Joint Committee for Guides in Metrology (2008). http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf
14. JCGM 200:2012: International Vocabulary of Metrology - Basic and general concepts and associated terms (VIM), 3rd edn. Joint Committee for Guides in Metrology (2012). http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf
15. Jiménez-Ramírez, A., Weber, B., Barba, I., del Valle, C.: Generating optimized configurable business process models in scenarios subject to uncertainty. Inf. Softw. Technol. **57**, 571–594 (2015)
16. Kosko, B.: Fuzziness vs. probability. Int. J. Gen. Syst. **17**(2–3), 211–240 (1990)
17. Lee, E.A.: Cyber physical systems: design challenges. In: Proceedings of ISORC 2008, pp. 363–369. IEEE (2008)
18. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994)
19. Littlewood, B., Neil, M., Ostrolenk, G.: The role of models in managing the uncertainty of software-intensive systems. Reliab. Eng. Syst. Saf. **50**(1), 87–95 (1995)
20. Mayerhofer, T., Wimmer, M., Burgueño, L., Vallecillo, A.: Specifying quantities in software models (2018, submitted). Technical report: http://atenea.lcc.uma.es/index.php/Main_Page/Resources/DataUncertainty
21. Object Management Group: Object Constraint Language (OCL) Specification. Version 2.2, February 2010. OMG Document formal/2010-02-01
22. Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1, June 2011. OMG Document formal/2011-06-02
23. Object Management Group: Unified Modeling Language (UML) Specification. Version 2.5, March 2015. OMG Document formal/2015-03-01
24. Object Management Group: OMG Systems Modeling Language (SysML), Version 1.4, January 2016. OMG Document formal/2016-01-05
25. Object Management Group: Structured Metrics Metamodel (SMM) Specification. Version 1.1.1, April 2016. OMG Document formal/16-04-04
26. Salay, R., Chechik, M., Horkoff, J., Sandro, A.: Managing requirements uncertainty with partial models. Requir. Eng. **18**(2), 107–128 (2013)
27. Selic, B.: Beyond mere logic - a vision of modeling languages for the 21st century. In: Proceeding of MODELSWARD 2015 and PECCS 2015, p. IS–5. SciTePress (2015)

28. Vallecillo, A., Morcillo, C., Orue, P.: Expressing measurement uncertainty in software models. In: Proceedings of the 10th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 1–10 (2016)
29. Wikipedia: List of uncertainty propagation software. https://en.wikipedia.org/wiki/List_of_uncertainty_propagation_software. Accessed 13 Apr 2018
30. Wolf, M.: A modeling language for measurement uncertainty evaluation. Ph.D. thesis, ETH Zurich (2009)
31. Zhang, M., Ali, S., Yue, T., Norgren, R., Okariz, O.: Uncertainty-wise cyber-physical system test modeling. Softw. Syst. Model. (2017). https://doi.org/10.1007/s10270-017-0609-6
32. Zhang, M., Selic, B., Ali, S., Yue, T., Okariz, O., Norgren, R.: Understanding uncertainty in cyber-physical systems: a conceptual model. In: Wąsowski, A., Lönn, H. (eds.) ECMFA 2016. LNCS, vol. 9764, pp. 247–264. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42061-5_16

On the Influence of Metamodel Design to Analyses and Transformations

Georg Hinkel[1(✉)] and Erik Burger[2]

[1] Software Engineering Department, FZI Research Center of Information Technologies, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
hinkel@fzi.de

[2] Software Design and Quality Group (SDQ), Karlsruhe Institute for Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany
burger@kit.edu

Abstract. Metamodels are a central artifact of model-driven engineering. As they determine the structure of instance models, they are a foundation for other model-driven artifacts such as model transformations, code generators or model analyses. Therefore, the quality of metamodels is important for any model-driven process. However, the implications of metamodel design to other artifacts such as model analyses or model transformations has barely been looked at through empirical research. In this paper, we present an empirical study where we analyzed equivalent model analyses and transformations for 19 different metamodels of the same domain. The results indicate that metamodel design has a strong influence to model analysis in terms of code metrics but only little influence on model transformations targeting this metamodel.

1 Introduction

To aid the increased complexity of modern (software) systems, model-driven engineering (MDE) helps by raising the level of abstraction. Models can be used for analysis to conclude insights on the represented system or for transformation into other artifacts, such as models of other metamodels or code.

In the Neurorobotics-platform developed in the scope of the *Human Brain Project* (HBP), these dependent artifacts include not only editors, but also an entire simulation platform where the connection between robots and neural networks is described in models [1,2]. As the HBP is designed for a total duration of ten years, it is likely that the metamodel will degrade unless extra effort is spent for its refactorings [3,4]. For such refactorings, we aim to measure their success and potentially automate them.

A central artifact for the specification of model analyses or model transformations is the metamodel [5] as it defines the abstract syntax used by instances. As a consequence, metamodel design has a strong impact on the design of model analyses and model transformations, particularly because metamodel evolution usually implies an evolution of these artifacts [6]. Therefore, ensuring the quality of metamodels is very important.

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 63–79, 2018.
https://doi.org/10.1007/978-3-319-92997-2_5

As metamodels are formalizations of the domain they are describing, many design decisions for the creation of the metamodel are already determined by the domain. Nevertheless, many decisions are left to the metamodel developer, opening a design space of metamodels, even for a fixed domain. The goal of our research is to evaluate the metamodels of this design space for their quality. This quality may be dependent on the particular domain, but we think there are also notions of it that are domain-independent such as subtleties in the inheritance and composition hierarchies.

There have been surprisingly few research activities on the actual implications of metamodel design to other artifacts: Does a concise metamodel imply concise analyses? Does a modular metamodel support modularity of model transformations? Does an understandable metamodel help to understand analyses? Often, it is difficult to answer these questions, because especially the quality of metamodels is not well captured by metrics. Further, as model analyses and transformations are laborious to create, they are hardly created for multiple semantically equivalent metamodels. This makes it very hard to reason on the influence of metamodeling design decisions to these artifacts: We usually do not know how transformations or analyses would look like if the metamodel was designed differently. Existing empirical studies neglect the different intention of transformations [7] or only look at correlations within the metamodel [8–10].

On the other hand, an automated quantitative measurement of quality is only the second step: For many purposes, it suffices to know whether the intuitive perception of quality correlates with code metrics of model transformations or analyses: To what degree matches the intuition of what is good or bad in a metamodel to what the metrics for later developed artifacts tell us.

In this paper, we present an empirical study to analyze and quantify the design impact of 19 different metamodels of the same domain to 38 equivalent model analyses (two for each metamodel) and 19 equivalent transformations that each transform into one of the metamodels from a common source model. We have reported correlations between the perceived quality or metric values of the metamodels and metric values of model analyses or transformations.

The remainder of this paper is structured as follows: Sect. 2 explains the experiment setup in more detail. Section 3 presents the resulting correlations. Section 4 discusses insights drawn from the results. Section 5 discusses threats to validity. Section 6 presents related work. Lastly, Sect. 7 describes future work before Sect. 8 concludes the paper.

2 Method

The method for our study is sketched in Fig. 1.

At first, a group of participants create metamodels on a common domain description. Then, we propose to let the participants review each others created metamodels. In parallel, transformation developers and analysis developers may develop artifacts based on the created metamodels. Here, it is very important that analyses and transformations are created consistently across metamodels,

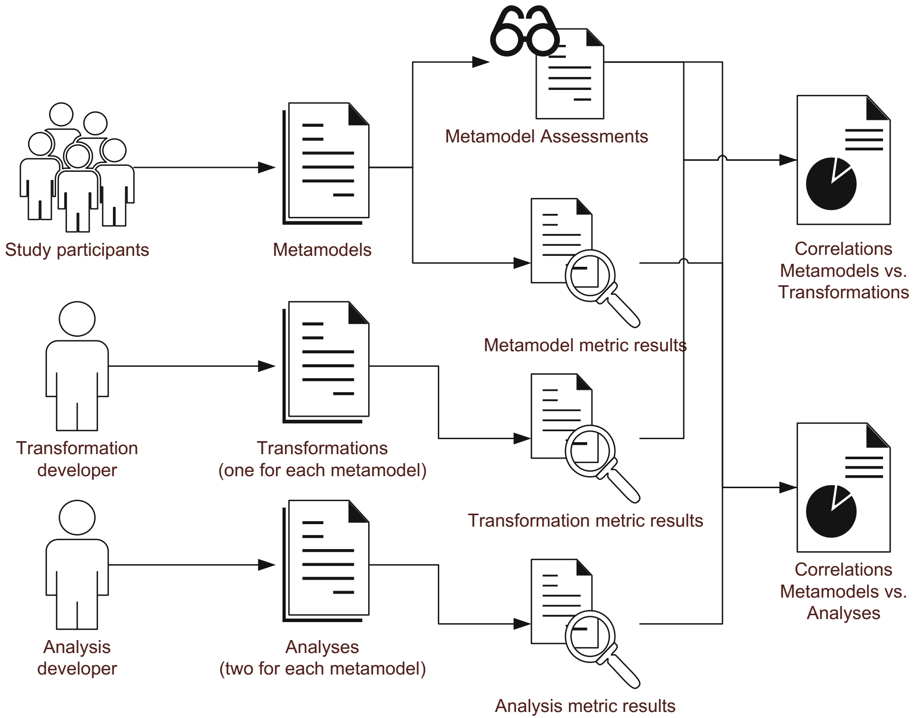


Fig. 1. Method overview for the empirical study

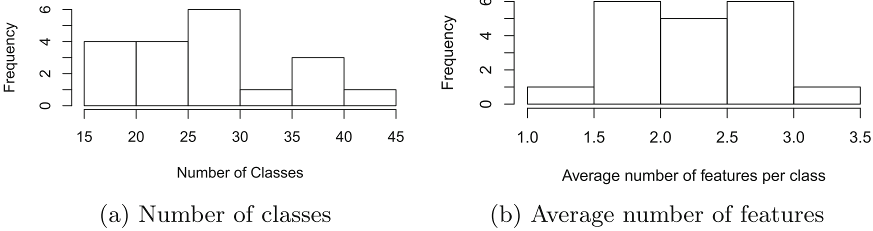
which is why all transformations and all analyses should be written by the same developers, ideally. Then, from each artifact, metamodels, analyses and transformations, several metrics are computed that are correlated in a last step. In the remainder of this section, we discuss each step in more detail.

2.1 Data Sources

In this section, the data sources of this experiment setup are described in detail.

Metamodels. The 19 metamodels we use originate from students attending a practical course on model-driven engineering across multiple years. The students were working in groups of two to create an Ecore-metamodel based on a textual domain description of component-based software architectures. This domain description includes creating a component repository, assembling software architectures from components and deploying software architectures to resources. The domain description is inspired by Palladio [11] that uses similar models to predict non-functional properties of such systems. Palladio is taught in other courses so that several students already knew the underlying concepts. In the experiment,

we asked them to create Ecore metamodels for these concepts. For that, the students used the tools provided by Eclipse.



(a) Number of classes (b) Average number of features

Fig. 2. Characteristics of the created metamodels

Due to space limitations, we cannot describe the 19 metamodels in detail. To still give an impression on the created metamodels, we depicted a histogram of the number of classes in Fig. 2a. The metamodels contained between 15 and 45 classes with a peek between 25 and 30 classes. Further, we depicted the average number of features per class in Fig. 2b, i.e. the number of attributes and references, including those defined in base classes.

We did not publish the metamodels as they originate from homework assignments of a practical course and unfortunately, we have not for a consent to publish them.

Peer-Reviews. We captured the human perception of metamodel quality for the metamodels in a controlled setting: We gave a questionnaire to participants of the last edition of that practical course and asked the students to review 7 metamodels of their colleagues, making sure that nobody had to review their own metamodel. The questionnaire that we used is publicly available online[1].

The review of the metamodels was done by students in a controlled experiment setting, though we allowed the students to collaborate on evaluating metamodels assigned to them. However, the low level of experience with modeling techniques means that the peer reviews rather reflect an intuitive feeling on the metamodels rather than expert opinions.

Similar to previous experiments [9,10], we asked students to evaluate the complexity, understandability, conciseness, modularity, consistency, completeness and changeability of the metamodels. Further, students were asked to estimate how easy it would be to create instances of this metamodel and how easy it would be to specify model transformations. Lastly, they should evaluate the overall quality. Complexity is asked as the degree in which a metamodel is not complex. Therefore, a high value for complexity means that the metamodel is simple. As the values obtained this way are basically intuitions, they are highly subjective.

[1] https://sdqweb.ipd.kit.edu/wiki/Metamodel_Quality.

```
Root =>
  (from assemblyConnector in Root.Systems
      .FirstOrDefault().AssemblyConnectors
   where !assemblyConnector.From.Provided
      .Contains(assemblyConnector.ConnectedInterface) ||
     !assemblyConnector.To.Required
      .Contains(assemblyConnector.ConnectedInterface)
   select assemblyConnector).Distinct()
```

Listing 1. An exemplary analysis whether components are correctly connected

Analyses. Based on the metamodels, we created two model analyses and a model transformation for each metamodel. The analyses compute violations to the following validity rules:

Correct Connection of Assemblies. Assemblies, i.e. usages of components in a software architecture, must be connected on the same interface. This interface is required by the requiring component and provided by the providing component.

Correct Allocation of Assemblies. Assemblies that are connected in the software architecture must be deployed either to the same resource container or there must be a link between the resource containers. Such a connection exists because the component of one assembly requires an interface provided by the component of the other assembly.

We are aware that validation constraints are only one particular kind of analyses. In the domain that we studied, they seem to us as a good compromise between simplicity (the analyses have to be implemented for each metamodel) and non-trivial complexity.

All model analyses are implemented C# in the query syntax, based on metamodel representations using NMF [12]. As argued previously by Akehurst et al. [13], C# is equivalent to OCL. Therefore, we expect that the results for analyses written in the C# query syntax should generalize to OCL. However, the usage of C# allows us to use the code metric implementations from Visual Studio to correlate them to metamodel properties.

As two examples, we depicted exemplary analyses in Listings 1 and 2 (for MM15).

Both model analysis tasks are rather simple and therefore solvable in a few lines of code. The solutions for the query whether assemblies are correctly contained ranged from 6 to 25 lines of code. The solutions for the allocation query ranged from 17 to 62 lines of code.

All analyses and transformations were created by two students in order to reduce the influence of the individual analysis developer. After all analyses and transformations were written, all of them were refactored to minimize learning effects.

```
Root =>
  (from connector in Root.Systems
      .FirstOrDefault().AssemblyConnectors
   from requiringAllocation in Root.Allocations
      .FirstOrDefault().AllocationContexts
   where requiringAllocation.AllocatedContext == connector.From
   from providingAllocation in Root.Allocations
      .FirstOrDefault().AllocationContexts
   where providingAllocation.AllocatedContext == connector.To &&
      providingAllocation.Container != requiringAllocation.Container &&
      !Root.Environments.FirstOrDefault().Links
        .Any(link =>
          link.Containers.Contains(providingAllocation.Container) &&
          link.Containers.Contains(requiringAllocation.Container))
   select connector).Distinct()
```

Listing 2. An exemplary analysis whether components are correctly allocated

Transformations. For every metamodel, we also implemented a model transformation from the much more detailed Palladio Component Model (PCM, the metamodel used in Palladio) to that metamodel. PCM also describes component repositories, software architecture and deployment of component-based software architectures. However, PCM is a complex metamodel with more than 200 classes, as it supports the architecture-based quality prediction of component-based systems. If a developer simply wants to take a look at the software architecture and deployment, a transformation to a simpler metamodel is appropriate.

As we always create a transformation from PCM, we vary the target metamodel of the model transformation, not the source. The reason is that we assume that the impact of metamodel design to model transformations from the metamodel into another domain is quite similar to the impact that metamodel design has on model analyses.

All model transformations have been implemented using ATL [14], one of the most commonly used textual model transformation languages [15].

Because the created metamodels are mostly projections of PCM, many transformation rules are essentially simple copy rules with filters. For example, we depicted the transformation rule to transform assembly contexts from PCM to MM15 in Listing 3.

2.2 Metrics

For a quantitative analysis, we capture metrics for each of the involved artifacts – metamodels, analyses and transformations.

Metamodels. Although many studies have reported metrics for metamodels, very few of them are empirically validated (cf. Sect. 6), making the selection of metrics to some degree arbitrary. In this work, we use the adapted Sarkar metrics for inheritance-based coupling (IC), association-based coupling (AC), association-based coupling from opposite references (AC_{op}), association-based

```
rule AssemblyContext2AssemblyContext {
 from
  pcmAssemblyContext : PCM!AssemblyContext
 to
  assemblyContext : MM15!AssemblyContext(
   name <- pcmAssemblyContext.entityName,
   instantiatedComponent <- pcmAssemblyContext.
        encapsulatedComponent__AssemblyContext,
   provided <- pcmAssemblyContext.encapsulatedComponent__AssemblyContext
     .providedRoles_InterfaceProvidingEntity
     ->select(role | role.oclIsKindOf(PCM!OperationProvidedRole))
     ->collect(role | role.providedInterface__OperationProvidedRole),
   required <- pcmAssemblyContext.encapsulatedComponent__AssemblyContext
     .requiredRoles_InterfaceRequiringEntity
     ->select(role | role.oclIsKindOf(PCM!OperationRequiredRole))
     ->collect(role | role.requiredInterface__OperationRequiredRole)
  )
}
```

Listing 3. An exemplary transformation rule from PCM

coupling of composite references (AC_{cmp}) and the class uniformity (CU) [10][2] and some basic metrics such as the number of classes (TNC), the number of features (NF, attributes and references of a class, including inherited ones) and the depth of inheritance (DIT).

Analyses. For the analyses, we rely on widely accepted metrics: cyclomatic complexity, class coupling and lines of code. Additionally, we use the *Maintainability index* [16], a composite metric based on the Halstead effort, the lines of code and the cyclomatic complexity. We use the implementations of these metrics that ship with Visual Studio as the analyses are specified in C#[3].

Transformations. For the ATL transformations, we use the metrics defined by van Amstel [17]. However, some of these metrics rather measure the coding style of the transformation developer such as the number of unused rules. Though these metrics are useful for transformation development, we do not expect a correlation of this metric to metamodel design. Therefore, we concentrate only on the following metrics:

- The number of transformation rules
- The number of rule inheritance trees
- The mean number of bindings per rule
- The mean cyclomatic complexity of helpers.

These metrics are chosen because a correlation of them with metamodel quality seems reasonable and the metrics are sufficiently generic. We did not account

[2] In [10], we also introduced an adapted version of module uniformity (MU) but we discarded this metric as it showed major weaknesses.

[3] In contrast to [16], Visual Studio rescales the maintainability index to fit into the value range of 0 to 100.

for metrics such as the number of abstract transformation rules or the depth of rule inheritance trees simply because the model transformations that we used as inputs hardly used abstract transformation rules and therefore, correlations are rather random. However, we included the number of inheritance trees such where transformation rules that share a common base rule are counted as one.

2.3 Analysis

For the analysis, we rely on a statistical tool: Pearson correlation indices. Because these correlation indices require the random variables to be normal distributed, we check this using Quartile-Quartile-plots (QQ-plots). QQ-plots print the quartiles of a sample distribution against the quartiles of a normal distribution. If the points appear on a line, the assumption of a normal distribution for the sample data is reasonable.

Due to the low number of data points in our experiments, we do not control for family-wise error rates. As a consequence, our results are not statistically clear but only indicate trends. More empirical studies are necessary to confirm the results of this paper to be sure on a good level of confidence.

3 Results

We divide the presentation of results into four parts: Correlations of model transformation metrics and model analysis metrics to metamodel perception and metamodel metrics. Due to space limitations, we only depict selected correlations. However, the raw metric values, scripts used to obtain the correlations and spreadsheets with all correlations are available online[4].

3.1 Correlations Between Metamodel Perception and Model Transformation Metrics

The correlations between the perceived quality evaluations from the students and metrics for the model transformations are depicted in Table 1.

Very interesting is the fact that there are no significant correlations for Transformation Creation with any of the considered transformation metrics (there are also no significant correlations with any of the metrics not reported in Table 1). We have two interpretations for this fact. Either, the impact of metamodel design to model transformations is not well captured by the selected metrics or it is not easy to estimate how good a metamodel is suited for the creation of model transformations.

The strongest correlation between perceived metamodel quality and metrics of the model transformations is the connection between complexity and the mean cyclomatic complexity of helpers: A simple metamodel is correlated with a low cyclomatic complexity of helpers, can be expressed with slightly less transformation rules that contain more bindings.

[4] https://sdqweb.ipd.kit.edu/mediawiki-sdq-extern/images/a/a0/ECMFA2018Results.zip.

Table 1. Correlations between model transformation metrics and perceived metamodel quality

	# Transformation rules	# Rule inheritance trees	Mean # bindings	Mean helper CC
Complexity	−0.16	−0.10	0.39	−0.43
Understandability	0.20	0.27	0.38	−0.06
Conciseness	0.16	0.14	0.25	−0.17
Modularity	0.34	0.44	0.03	0.08
Consistency	0.07	0.04	0.13	0.12
Completeness	−0.36	−0.20	0.17	0.33
Correctness	−0.08	−0.02	−0.05	0.26
Changeability	0.07	0.14	0.13	0.10
Instance creation	0.00	−0.01	0.11	−0.13
Transformation Creation	0.24	0.31	−0.16	−0.11
Overall quality	0.08	0.19	0.08	0.14

3.2 Correlations Between Metamodel Perception and Model Analysis Metrics

The correlations between metamodel perception and model analysis metrics are depicted in Table 2. In this table, we have printed correlations with an absolute value higher than 0.5 in bold. For 19 observations, the threshold for a significant (i.e. significantly larger than 0) correlation is 0.53 for a one-sided test at confidence level 99% and 0.58 for the two-sided test. As mentioned above, we do not take the family-wise error-rate into account and therefore, the results only indicate trends.

We can see strong correlations (meaning that $|\varrho| > 0.5$) from the metamodel correctness to all of the selected code metrics for both analyses with the only exception for class coupling in the allocation analysis. The reason for this correlation is that metamodels perceived as incorrect often lack important navigation links. Therefore, analyses based on these metamodels often use naming conventions instead of references.

Furthermore, we have strong correlations between the overall quality and the maintainability index. This correlation is very interesting as the maintainability index is a very aggregated metric, similar to the overall quality of a metamodel being an aggregate of the quality attributes. The fact that there is a correlation between the two of them confirms the hypothesis that better metamodels make model analysis easier, even though this connection is hard to grasp, at least in terms of other quality attributes or code metrics, besides correctness.

Table 2. Correlations between model analysis metrics and perceived metamodel quality. The analyses are abbreviated with *Conn* for the analysis for *Correct Connection of Assemblies* and *Alloc* for the analysis *Correct Allocation of Assemblies*.

Analysis	Class coupling		Lines of code		Maintainability index		Cyclomatic complexity	
	Alloc.	Conn.	Alloc.	Conn.	Alloc.	Conn.	Alloc.	Conn.
Complexity	0.04	0.03	0.06	−0.17	0.09	0.10	−0.22	−0.16
Understandability	−0.05	0.03	0.07	−0.04	0.11	0.08	−0.23	−0.03
Conciseness	0.20	−0.10	0.02	−0.22	0.07	0.27	−0.14	−0.21
Modularity	−0.14	−0.03	0.03	−0.01	0.09	0.07	−0.14	−0.01
Consistency	−0.25	−0.27	−0.34	−0.29	0.40	0.30	−0.36	−0.28
Completeness	−0.46	−0.08	−0.18	−0.14	0.28	0.12	−0.28	−0.13
Correctness	−0.32	**−0.54**	**−0.54**	**−0.58**	**0.59**	**0.62**	**0.51**	**−0.57**
Changeability	−0.23	−0.16	−0.24	−0.20	0.35	0.18	−0.35	−0.19
Instance creation	−0.08	−0.05	−0.11	−0.23	0.21	0.20	−0.25	−0.22
Transf. creation	−0.33	−0.42	−0.13	−0.36	0.33	0.36	−0.41	−0.36
Overall quality	−0.37	0.43	−0.43	−0.48	**0.52**	**0.51**	**−0.50**	−0.47

3.3 Correlations Between Metamodel Metrics and Model Transformation Metrics

The correlations between the metrics from metamodels and model transformations are depicted in Table 3. Again, we have printed correlations with $|\varrho| > 0.5$ in bold.

Table 3. Correlations between model transformation metrics and metamodel metrics

	# Transf. rules	Mean # bindings	Mean helper CC
TNC	−0.07	−0.12	**0.82**
DIT	0.04	0.17	0.36
NF	0.09	−0.16	**−0.83**
AC	−0.40	−0.11	**0.50**
AC_{cmp}	−0.31	−0.29	0.43
AC_{op}	−0.28	−0.05	**0.51**
CU	−0.24	0.01	**−0.51**
IC	−0.41	−0.28	−0.03

Very interestingly, the cyclomatic complexity of the helpers correlates strongly with the total number of classes, but strongly negative with the number of features. This is due to the different implementations of parameter types

in the metamodels: While some metamodels implemented these types as enumerations (allowed by the domain description) while other implemented every enumeration literal as a type. Because these types do not have any features, also the average number of feature is lower for these metamodels. In the transformation, the helper implementation for the enumerations is a pre-initialized map from which the corresponding enumeration entry can be obtained. This causes a very low cyclomatic complexity whereas the implementation for metamodels with explicit type classes requires a more elaborated control flow.

Another interesting, though not so strong correlation is between the Sarkar coupling metrics AC or IC and the number of transformation rules, especially because the correlation is negative: A metamodel with a higher coupling can be transformed to with fewer rules. The inheritance-based coupling IC also correlates with coefficient 0.43 with the average number of output pattern elements of a transformation rule. Indeed, some of the metamodels are tightly coupled such that the conceptual separation between a component repository, a system and its deployment becomes meaningless as the classes are very intertwined. In such a case, a single transformation rule handles the transformation of all these concepts together.

However, fewer but larger rules also means that the connection between input and output model elements gets blurred. Moreover, rules also act as a unit for reuse, in particular for superimposition. This point of view also explains the positive, yet not so strong correlation of perceived metamodel modularity and the number of transformation rules and moreover the number of rule inheritance trees.

3.4 Correlations Between Metamodel Metrics and Model Analysis Metrics

The correlations between metamodel metrics and model analysis metrics are depicted in Table 4.

The only correlation with $|\varrho| > 0.5$ between metamodel metrics and metrics of the model analyses is the negative correlation between the DIT of the metamodel and lines of code in the analysis: A more extensive usage of inheritance goes along with a reduction of the lines of code. This correlation is reasonable because the increased usage of inheritance enables a unified transformation of model elements. However, the correlation is only strong in the allocation analysis and much weaker in the connection analysis.

4 Discussion

The fact that the manual metamodel assessments were done by students instead of more trained experts in modeling technologies means that we essentially measure how intuitive the quality of model analyses and transformations can be guessed from the metamodel. The fact that we did not see very strong and significant correlations in many cases is therefore not surprising.

Table 4. Correlations between model analysis metrics and metamodel metrics. The analyses are abbreviated like in Table 2.

Analysis	Class coupling		Lines of code		Maintainability index		Cyclomatic complexity	
	Alloc.	Conn.	Alloc.	Conn.	Alloc.	Conn.	Alloc.	Conn.
TNC	0.00	0.28	−0.11	0.28	−0.02	−0.21	0.29	0.18
DIT	−0.23	−0.20	**−0.55**	−0.28	0.47	0.33	−0.27	−0.27
NF	−0.32	−0.41	−0.05	−0.32	0.14	0.23	−0.34	−0.24
AC	−0.14	0.24	−0.23	0.14	0.24	−0.21	0.17	−0.17
AC_{cmp}	−0.08	0.34	−0.23	0.19	0.16	−0.28	0.21	−0.05
AC_{op}	0.11	0.35	−0.15	0.10	0.12	−0.14	0.12	−0.05
CU	0.05	−0.18	−0.01	−0.23	0.06	0.14	−0.23	−0.15
IC	0.32	0.30	0.46	0.27	−0.45	−0.32	0.28	0.33

Furthermore, the correlations that we found often had explanations that are very specific to the domain: The fact that complexity influenced helpers more than transformation rules[5] is presumably specific to our case of a transformation between similar component models.

As shown for object-oriented design already [18], a higher DIT value eventually correlates with a fault probability, the fact that the DIT correlated with shorter (in terms of lines of code) analyses is therefore probably due to the experiment setup.

However, there are also connections that we think are to some degree independent of our scenario:

- The fact that a more accurate (correct) metamodel simplifies navigation through the model which in turn improves all code metrics of an analysis is a connection that we think holds independent from our study. However, we will have to repeat the study for other domains and other types of analysis to verify this.
- Using an enumeration in case there is no additional information need simplifies the analysis. In addition to the correlations, we think that a key advantage here is that using an enumeration makes it very explicit that constants are used, meanwhile this is not clear for classes that have no features: It is unclear whether they should be treated as singletons or not.
- A high coupling in the metamodel makes model transformation rules in transformations targeting that metamodel more complex as the target model elements are very intertwined. Whether the Sarkar metrics are a suitable choice to control this coupling will have to be double-checked.

[5] The metric set by van Amstel does not include a metric to measure the complexity of model transformation rules, so we might have seen results if we had a proper metric.

Another very interesting finding is that while there is no model transformation metric that correlates strongly with the *perception* of any high-level quality attribute, we have a strong correlation even between the most abstract overall metamodel quality and very abstract code metrics such as the Maintainability Index. This may indicate that whereas code metrics do a relatively good job in capturing the complexity of code, this is not as much the case for model transformations. Thus, we require more elaborate model transformation metrics to better capture and thus predict the quality of a model transformation.

However, there is also another interpretation to this correlation, namely that the quality of a metamodel simply has more implications to artifacts that consume instances of it (such as model analyses) than to artifacts that produce these instances (such as model transformations targeting this metamodel).

Interestingly, if we have a look at the correlations between metamodel *metrics* and metrics for model transformations and model analyses, we see a different picture: While there is only one stronger correlation of the metamodels to code, we see plenty of them to model transformation metrics. We think that this is because the metrics of model transformations and metamodels are somehow closer together. A possible reason for this is that existing metamodel metrics are as bad in characterizing metamodel quality as model transformation metrics are for characterizing model transformation quality.

The results (the correlations) from this study will have to be reproduced at least in another domain and for different types of analysis for the results to have a more general applicability. Independently from the obtained correlations, they do not imply causality. Therefore, the presented paper only is a starting point to study the connections between metamodels and other artifacts in more depth.

5 Threats to Validity

As usual, we divide the threats to validity into internal and external validity.

5.1 Internal Validity

The participants in the study did not know how we wanted to analyze the peer-reviews from the metamodels, nor did they know what analyses or transformations we wanted to create based on these metamodels. Therefore, we can exclude a subject effect. For the correlation of metrics, such a subject effect is also clear.

We collected the metamodel reviews in a single session such that we can exclude an influence of histories, maturation or mortality. For the creation of the metamodels, these threats do not apply. The assignment of students to their review assignments was done by random such that we can exclude a subject effect from ourselves.

However, not all metamodels have been evaluated by all of the students as evaluating a metamodel is also quite time-consuming. Further, we may have faced an instrumentation or sequencing effect as the students evaluated the metamodels sequentially. However, we allowed the students to revoke edit their reviews after they had started reviewing other metamodels to mitigate such an effect.

Furthermore, the internal validity only affects the correlations of perceived metamodel quality to the other metrics.

5.2 External Validity

All metamodels that we have analyzed come from the same domain. Further, all analyses are in principle only two different analyses that essentially both simple validation constraints. Therefore, we cannot claim that the results are generalizable (cf. Sect. 4) for all domains and all analyses, but we think that the results may be a good indicator for future research.

6 Related Work

To the best of our knowledge, the study by Di Rocco et al. [7] was the only one yet to connect model transformations with metamodel metrics by means of empirical research. However, their study ignores the different purpose of the considered transformations. Furthermore, the relation of the analyzed metrics to quality of metamodels or model transformations is unclear as the study was only limited to correlations between metrics.

Most related work in the context of metamodel quality consists of adoptions of metrics for UML class diagrams and object-oriented design. In prior work, we have shown that this adoption is sometimes misleading as in the case of the Sarkar metrics [10]. Others work very well [19]. However, these studies did not consider model transformations or analysis.

Di Rocco et al. applied metrics onto a large set of metamodels [8]. Besides size metrics, they also feature the number of isolated metaclasses and the number of concrete immediately featureless metaclasses. Based on the characteristics they draw conclusions about general characteristics of metamodels. However, to the best of our knowledge, they did not correlate the metric results to any quality attributes.

A more elaborate analysis of related work in the context of measuring metamodel quality can be found in our previous work [10,19]. We do not repeat it here due to space limitations.

7 Future Work

In the current form of the experiment, we were only able to relate the quality of metamodels to static properties of model analyses and transformations. However, we are particularly interested also in the dynamic properties such as runtime for an example model and response times to incremental updates. For this, we only need to run the transformations and analyses incrementally. For this, we plan to use NMF Expressions[6] to incrementalize the analyses and NMF Synchronizations to run the transformations incrementally [21].

[6] See [20] for a usage example.

Furthermore, though our results look promising, we require more data points, in particular for more domains. Therefore, we aim to repeat the study in a different domain.

Lastly, it would be interesting to what degree the results of this study were different if we used expert opinions instead of student assessments for the metamodel quality.

8 Conclusion

In this paper, we have presented an empirical study to evaluate the influence of metamodel design to other artifacts, in particular model analyses and model transformations. We see this study as a starting point for empirical research in this direction in order to gain a better characterization of this connection and to use insights from such empirical study to improve the metamodel design process.

Besides metamodel metrics, we used intuitive metamodel assessments done by students to calculate correlations. Here, we see that the perception hardly correlates with model transformation metrics, but there are strong correlations with code metrics.

From the correlations between metamodel metrics and metrics of model analyses or model transformations, we were able to detect several correlations that we think apply independent of our experiment setup: More correct metamodels makes model analysis easier and a high coupling of the metamodels causes fewer but more complex rules in model transformations.

In general, the correlations between perceived metamodel quality and code metrics for the model analyses were stronger than the correlations between perceived metamodel quality and transformations targeting this metamodel. On the contrary, metamodel metrics seem to correlate stronger to model transformation metrics than they do to code metrics of the model analyses.

Because we had not enough data points to perform a family-wise error correction, these results have to be seen as preliminary and we look forward to repeat this study and confirm the results in future research.

Acknowledgements. We would like to thank all students that participated in our study as well as Frederik Petersen and Lennart Henseler who helped us creating the model transformations and analyses.

This research has received funding from the European Union Horizon 2020 Future and Emerging Technologies Programme (H2020-EU.1.2.FET) under grant agreement no. 720270 (Human Brain Project SGA-I).

References

1. Hinkel, G., Groenda, H., Vannucci, L., Denninger, O., Cauli, N., Ulbrich, S.: A domain-specific language (DSL) for integrating neuronal networks in robot control. In: 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-Based Software-Engineering (2015)
2. Hinkel, G., Groenda, H., Krach, S., Vannucci, L., Denninger, O., Cauli, N., Ulbrich, S., Roennau, A., Falotico, E., Gewaltig, M.-O., Knoll, A., Dillmann, R., Laschi, C., Reussner, R.: A framework for coupled simulations of robots and spiking neuronal networks. J. Intell. Robot. Syst. **85**, 71–91 (2016)

3. Lehman, M.M.: Programs, cities, students: limits to growth? (Inaugural Lecture - Imperial College of Science and Technology, 1974). University of London, Imperial College of Science and Technology (1974)
4. Lehman, M., Ramil, J., Wernick, P., Perry, D., Turski, W.: Metrics and laws of software evolution-the nineties view. In: Proceedings of the Fourth International Software Metrics Symposium, pp. 20–32 (1997)
5. Schmidt, D.C.: Model-driven engineering. IEEE Comput. **39**(2), 25 (2006)
6. Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 20–37. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_2
7. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Mining correlations of ATL model transformation and metamodel metrics. In: Proceedings of the Seventh International Workshop on Modeling in Software Engineering, pp. 54–59. IEEE Press (2015)
8. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Mining metrics for understanding metamodel characteristics. In: Proceedings of the 6th International Workshop on Modeling in Software Engineering, MiSE 2014, pp. 55–60. ACM (2014)
9. Hinkel, G., Kramer, M., Burger, E., Strittmatter, M., Happe, L.: An empirical study on the perception of metamodel quality. In: Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, pp. 145–152 (2016)
10. Hinkel, G., Strittmatter, M.: On using Sarkar metrics to evaluate the modularity of metamodels. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (2017)
11. Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K.: Modeling and Simulating Software Architectures - The Palladio Approach. MIT Press, Cambridge (2016). 408 pp.
12. Hinkel, G.: NMF: a modeling framework for the .NET platform. Technical report, Karlsruhe Institute of Technology (2016)
13. Akehurst, D.H., Howells, W.G.J., Scheidgen, M., McDonald- Maier, K.D.: C# 3.0 makes OCL redundant. In: Electronic Communications of the EASST, vol. 9 (2008)
14. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MODELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006). https://doi.org/10.1007/11663430_14
15. Troya, J., Vallecillo, A.: A rewriting logic semantics for ATL. J. Object Technol. **10**(5), 1–29 (2011)
16. Oman, P., Hagemeister, J.: Metrics for assessing a software system's maintainability. In: Proceedings of the Conference on Software Maintenance, pp. 337–344. IEEE (1992)
17. van Amstel, M., van den Brand, M.: Using metrics for assessing the quality of ATL model transformations. In: Proceedings of the Third International Workshop on Model Transformation with ATL (MtATL 2011), vol. 742, pp. 20–34 (2011)
18. Zhou, Y., Leung, H.: Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Trans. Softw. Eng. **32**(10), 771–789 (2006)
19. Hinkel, G., Strittmatter, M.: Predicting the perceived modularity of MOF-based metamodels. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (2018)

20. Hinkel, G., Happe, L.: An NMF solution to the TTC train benchmark case. In: Proceedings of the 8th Transformation Tool Contest, a Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, CEUR Workshop Proceedings, vol. 1524, pp. 142–146. CEURWS.org (2015)
21. Hinkel, G., Burger, E.: Change propagation and bidirectionality in internal transformation DSLs. Softw. Syst. Model. (2017)

Automatic Transformation Co-evolution Using Traceability Models and Graph Transformation

Adrian Rutle[1(✉)], Ludovico Iovino[2], Harald König[3], and Zinovy Diskin[4]

[1] Western Norway University of Applied Sciences, Bergen, Norway
aru@hvl.no
[2] Gran Sasso Science Institute, L'Aquila, Italy
ludovico.iovino@gssi.it
[3] FHDW Hannover, Hannover, Germany
harald.koenig@fhdw.de
[4] McMaster University, Hamilton, Canada
diskinz@mcmaster.ca

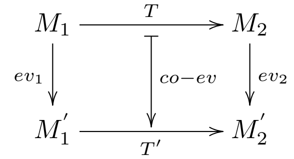
Abstract. In rule-based approaches, a model transformation definition tells how an instance of a source model should be transformed to an instance of a target model. As these models undergo changes, model transformations defined over these models may get out of sync. Restoring conformance between model transformations and the models is a complex and error prone task. In this paper, we propose a formal approach to automatically co-evolve model transformations according to the evolution of the models. The approach is based on encoding the model transformation definition as a traceability model and the evolution of the models as applications of graph transformation rules. These rules are used to obtain an evolved traceability model from the original traceability model. We will identify the criteria which need to be fulfilled in order to make this automatic co-evolution possible. We provide a tool support for this procedure, in which the evolved model transformation definition is derived from the evolved traceability model.

1 Introduction

In Model-driven software engineering (MDSE) models are used to represent certain aspects of software systems. Model transformations are used to encode model manipulations such as model translation, code-generation, behaviour definition, etc. These transformations are usually specified as a set of rules which tell how to transform instances of the source model to instances of the target model. Although the levels of definition (model/metamodel level) and application (instance/model level, resp.) of these transformations may vary in different contexts, we stick to the term "model transformation". That is, the model to which the transformation is applied might be located at any of the levels – M0, M1, M2, etc. – in a metamodelling hierarchy.

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 80–96, 2018.
https://doi.org/10.1007/978-3-319-92997-2_6

As model evolution is inevitable due to new regulations, evolution, refactoring, changes in requirements' specifications, etc., model transformations would also need to co-evolve in order to work properly on the new versions of the models [13,20,23,24,27,32]. The modeller can of course change the transformations manually by inspecting the model evolution. However, this procedure is based on individual skills and, like any other refactoring activity, presents intrinsic difficulties as it is error-prone and tedious if performed without automation [6]. Additionally, the ways a model transformation can co-evolve is not unique since its validity can be restored through different co-evolution strategies, i.e., different adaptations can be derived from the model evolution. Moreover, it is crucial that the modeller is able to specify the particular evolution strategy she has in mind in order to have a uniform and consistent co-evolution of the artefacts.



This diagram depicts the model evolution and model transformation co-evolution scenario. The models M_1 and M_2 represent the original source and target models which we start with, respectively. The mapping T from M_1 to M_2 represent the original model transformation definition. The model $M_1^{'}$ represents the new version of M_1, the same goes for $M_2^{'}$, while the mappings ev_1 and ev_2 represent the evolution of the original models. The mapping $co - ev$ represents the transformation co-evolution. In this paper, we consider the scenario where the source model M_1 is changed. The case where M_2 changes, or where both M_1 and M_2 change, are left for future work.

We encode the transformation definition T as traceability mapping [8] which we will represent as a traceability model. A traceability mapping is a set of traceability links between the models, which may be directly executed [12,25, 26], or used for automatic transformation rule generation [7]. For the formal semantics of traceability mappings we rely on Kleisli-mappings [9], which is a categorical construction consisting of two steps: first a derived instance is created by *querying* the source instance, then, the target instance is extracted as a *retyping* of the augmented instance. Although we focus our examples on rule based model transformations which are defined in ATL [16] due to its built-in support for traceability mapping creation, the general procedure can be applied to other transformation definition approaches. The expressiveness of the Kleisli-mappings depends on the language used in the querying step, hence picking the right query language would cover complex ATL (and other transformation languages) features.

The evolution ev_1 from M_1 to $M_1^{'}$ might have happened by editions in a model editor or by applying automatic model refactorings, however, the modification can be represented as a minimal (or normalised) sequence of rule applications which in turn can be extracted by inspecting the models (see e.g. [17,18]). In our implementation we use the language Edelta [1] for specifying and applying the evolution, but we formalise the approach by using graph transformation rules. Although our examples are refactoring evolutions from [29], the approach can be generalised to other evolutions as long as the necessary criteria are satisfied.

Before explaining the approach, we give an intuition on the necessary criteria for automatic co-evolution of transformation definition. First we have to check what might have happened in ev_1: If no element from M_1 is deleted by the evolution, the transformation will still be working. If we only rename elements, or change model elements which are not involved in the transformation, the rules can be retyped and the transformations will be working as before. However, if we have changed model elements by ev_1 over which we have typed some rules, the rules need to be changed (we say, co-evolved) accordingly. The change can be propagated to the transformation rule if it does not delete elements (renaming is allowed), which are used by the transformation rule. Adding elements to M_1 which affect negative context is not considered a co-evolution issue in this paper since the very purpose of defining the negative context in the first place is to block applying the rule in undesired situations.

In the next section we present our running example. Then in Sect. 3 we outline the formalisation of our technique, which is implemented in a prototype tool (see Sect. 4) for evaluation. In Sect. 5, we explore existing approaches for dealing with transformation evolution. Finally, in Sect. 6 we summarize the main ideas of the paper and outline our future research.

2 Motivating Example

This section explores a motivating example demonstrating the problem of coupled evolution of models and model transformations. The scenario involves two Ecore [33] models, Company (Fig. 1) and CRM (Fig. 2), and an ATL [16] transformation called Company2CRM (Listing 1).

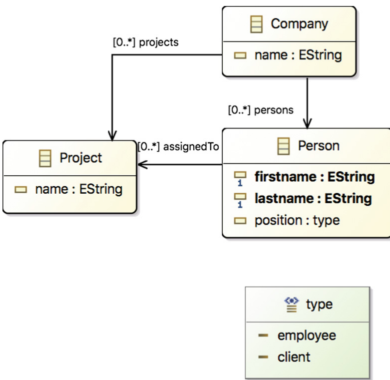


Fig. 1. Company model

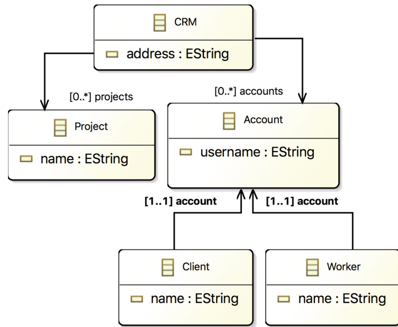


Fig. 2. CRM model

The Company model is used to define Companies that contain Persons identified with firstname, lastname and position. The position is specified with two different options, i.e., employee or client, specified in the model with an enumeration. Moreover each person can be assigned to multiple Projects. The CRM model supports the definition of software in the Customer Relationship Management domain. A CRM software system is usually composed of Accounts, that can be used to give access to Clients or Workers of the company.

The ATL transformation in Listing 1 is responsible for translating instances of the Company in Fig. 1 to instances of the CRM model in Fig. 2. The transformation is composed of four matched rules, the first one at lines 5–12 is responsible for creating a CRM instance for each Company, setting the web url, composed by an expression elaborating the company name. The rule Person2Worker creates an instance of Account and Worker if the Person instance has the position set to employee (line 14). On the contrary rule Person2Client generates an Account and a Client instance when the Person instance has the position set to client (line 25).

```
1  module Company2CRM;
2  create OUT: CRM from IN: Company;
3
4  rule Company2CRM{
5  from s: Company!Company
6  to t: CRM!CRM(
7  address <- 'www.'+s.name.toLower()
       +'.com',
8  accounts <- s.persons,
9  projects<-s.projects  )
10 }
11 rule Person2Worker{
12 from s: Company!Person(s.position
       =#employee)
13 to t: CRM!Account(
14 username <- s.firstname.toLower()+
       '.'+s.lastname.toLower() ),
15 t1: CRM!Worker(
16 account<-t,
17 name<-s.firstname+'␣'+s.lastname )
18 }
19 rule Person2Client{
20 from s: Company!Person(s.position
       =#client)
21 to t: CRM!Account(
22 username <- s.firstname.toLower()+
       '.'+s.lastname.toLower() ),
23 t1: CRM!Client(
24 account<-t ,
25 name<-s.firstname+'␣'+s.lastname )
26 }
27 rule Project2Project{
28 from s: Company!Project
29 to t: CRM!Project(
30 name<-s.name )
31 }
```

Listing 1. *Company2CRM*

```
1  module Company2CRM;
2  create OUT: CRM from IN: Company;
3
4  rule Evolved_Company2CRM{
5  from s: Company!Organisation
6  to t: CRM!CRM(
7  address <- 'www.'+s.name.toLower()
       +'.com',
8  accounts <- s.persons,
9  projects<-s.projects )
10 }
11 rule Evolved_Person2Worker{
12 from s: Company!Employee
13 to t: CRM!Account(
14 username <- s.firstname.toLower()+
       '.'+s.lastname.toLower() ),
15 t1: CRM!Worker(
16 account<-t,
17 name<-s.firstname+'␣'+s.lastname )
18 }
19 rule Evolved_Person2Client{
20 from s: Company!Client
21 to t: CRM!Account(
22 username <- s.firstname.toLower()+
       '.'+s.lastname.toLower() ),
23 t1: CRM!Client(
24 account<-t ,
25 name<-s.firstname+'␣'+s.lastname )
26 }
27 rule Project2Project{
28 from s: Company!Project
29 to t: CRM!Project(
30 name<-s.name )
31 }
```

Listing 2. *EvoCompany2CRM*

We consider now a model evolution scenario where the source of the transformation in Listing 1 has been evolved to the model in Fig. 3. We can describe the patterns applied on the model as follows. First, an *introduction of subclasses* [29] is applied: (i) the class Person becomes abstract, (ii) two new classes are added, i.e. Employee and Client, (iii) the attribute position with its enumeration is deleted as effect of introduction of subclasses.

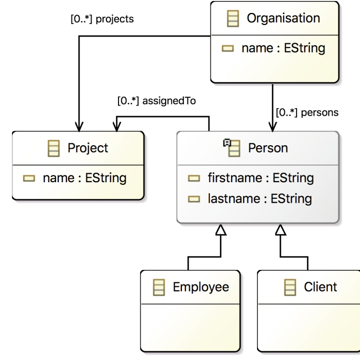


Fig. 3. Evolved source model

As effect of this evolution on the source model, the transformation Company2CRM in Listing 1 is corrupted. In particular, the lines 14 and 25 are responsible for the error *"Feature position does not exist on Person"*. Another refactoring pattern applied in this example is the *Rename class*, where the class Company is renamed to Organisation. This refactoring applied on the Company model triggers another inconsistency in the transformation execution, which is identified in line 6 where Company is highlighted as no longer existing in the source of the transformation.

In Listing 2, the expected migrated Company2CRM transformation is shown, namely EvoCompany2CRM. This repairing operation is complex and hence error prone if manually performed, especially if multiple refactoring patterns are applied to the models [32]. The repaired transformation parts in Listing 2 can be identified in lines 6 where the class Company in the source pattern of the transformation rule is renamed to Organisation; as well as lines 14–25, where the filtering conditions of the previous version have been replaced by the new input patterns, i.e., Employee and Client.

3 Formalization

In this section, we show how to formalize the induced co-evolution of model transformations due to evolved source models M_1. Note that we will not go into details of categorical constructions such as pushout, pullback, morphisms, etc., the interested reader may for example consult [11].

3.1 Model Transformations as Query-Retyping-Combination

A model transformation T is specified by a source model M_1 and a target model M_2 and correspondences between elements of M_1 and M_2 (a traceability mapping). The *execution* of the transformation takes an instance I_1 typed over M_1 as input and returns I_2 typed over M_2 (see top of Fig. 4) in two steps:

- Derive new data from existing elements in I_1 and augment I_1 with this data.
- Create I_2 as a substructure, modulo renaming, of the augmented I_1.

We pick up notations and terminology from [8,9], where the first step is called *query execution*, and the second step is called *retyping execution*. In the sample ATL transformation in Sect. 2, the crucial actions are the following (we disregard here transformation of names and the univalent mapping of Company to CRM and of Projects). For each Person p:

1. create a new account a,
2. create a new object x of type Worker/Client depending on whether p.position.emp = employee or p.position.cli = client (in Person2Worker/ Person2Client p.position = #employee is the concrete syntax for p.position.emp = employee), and
3. augment with a link from x to a.

This results in a relation between the source and target model as follows: Each person is in one-to-one correspondence to an account, each position value is in one-to-one-correspondence to either a worker or a client object. Hence these relations can be interpreted as *retypings*. In contrast to that, the account links are *true augmentations* of existing source data.

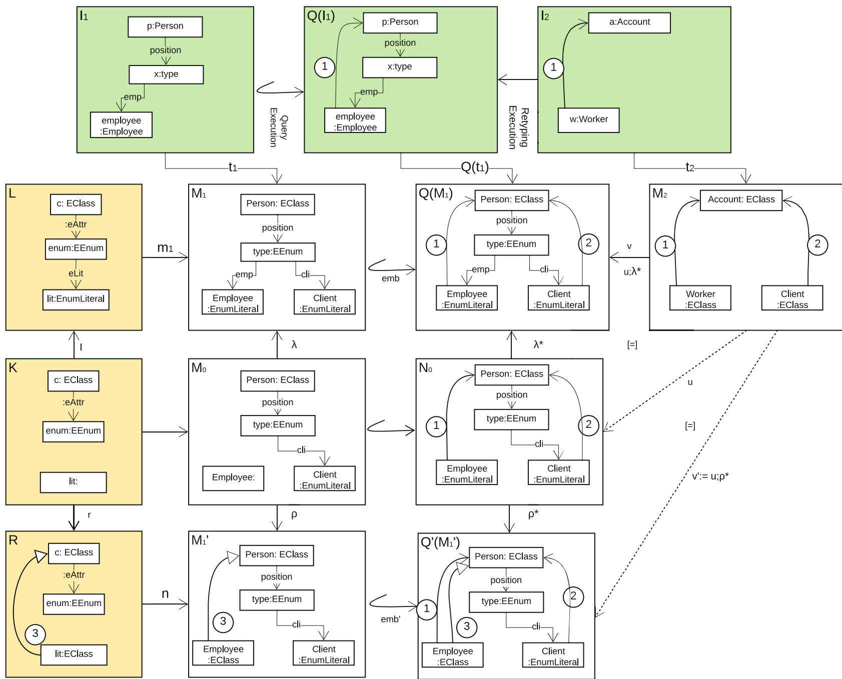


Fig. 4. Transformation co-evolution by rule application

Thus it makes sense to separate these two facts in terms of query and retyping specifications as shown in Fig. 4. The graph M_1 depicts the (relevant part of) model M_1, namely the fact that each person possesses a position with an enumeration type, which in turn references either of the literals Employee or Client. The query specification is an embedding *emb* of M_1 into the traceability model $Q(M_1)$, which specifies the above mentioned augmentations by two new associations labelled ① and ②. Note that in Fig. 4 class models are encoded as attributed graphs with node inheritance [3,11,14].

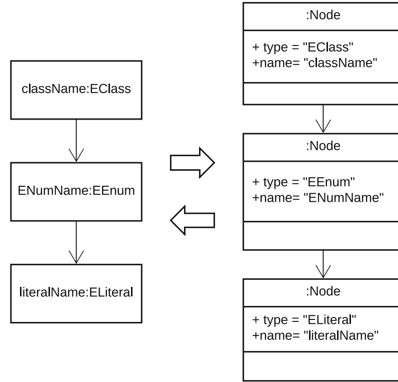


Fig. 5. Ecore to/from graph

For simplicity reasons we still call the nodes of our graphs for EClasses, EEnums, etc., however, we represent internally each node's type as an attribute of the node (see Fig. 5). This is to avoid violating constraints which are imposed by the Ecore metamodel, e.g. an ELiteral cannot refer to an EClass. Once our formal construction is tested with several examples, reverting back to a model which conforms to Ecore is as simple as deducing the types from the attribute which stores the type of the node.

Retyping in the above sense is specified by a graph morphism v in Fig. 4 (e.g. v(Account) = Person means that Account objects are retyped Persons). Also, v specifies that type enumerations will be excluded from the target. Upon execution of this transformation (see top row in Fig. 4), for an instance I_1 which is typed over M_1 by graph morphism t_1, the step *Query Execution* is an operation which takes t_1 as input and returns the augmentation of I_1 with relations ① = {(e:Employee, p:Person) | p.position.emp = e} and ② = {(c:Client, p:Person) | p.position.cli = c}. The *Retyping Execution* step accordingly performs retyping and exclusion by pulling back $Q(t_1)$ along v, i.e. by constructing the intersection over a common model taking mappings into account.

3.2 Transformations Co-evolution by Rule Application

As mentioned, a model evolution $M_1 \xrightarrow{ev_1} M_1'$ can be specified in a variety of ways. An appropriate choice is graph transformation rules [11]. From our sample evolution in Sect. 2 we disregard the renaming of Company to Organization, and focus only on the interesting evolution of introducing subclasses from enumerations. This evolution is performed by applying to M_1 the graph transformation rule $L \xleftarrow{l} K \xrightarrow{r} R$, where l is a monomorphism, in the left column of Fig. 4. We assume that no other class uses the type enumeration, such that a deletion rule (not shown) of a literal in this enumeration causes no side-effects.

The rule can be applied to a class with an attribute which stores a type (in the example the attribute position of Person). Since this attribute takes a certain

value from an enumeration, this value can be substituted by an appropriate subclass (subclassing is shown as the arrow labelled ③ in R). Now L can be matched with M_1 via morphism $m_1 : L \to M_1$. If $m_1(\mathsf{lit}) = \mathsf{Employee}$, using the double pushout (DPO) approach from [11] we get M_1' as shown in Fig. 4. This is achieved by deleting elements (matched with elements in) L but not in K, creating elements in R but not in K, while avoiding any dangling edges.

This rule can be applied at a second match m_2 (with $m_2(\mathsf{lit}) = \mathsf{Client}$). The application of this rule decreases the possible values of the position attribute. If all values are dereferenced, a second rule (not shown in Fig. 4) can be applied, which deletes the position attribute of Person together with the type enumeration. Note that the dangling condition in DPO rewriting prevents application of this rule as long as the matched attribute position possesses values. Any rule application as described above yields a span $M_1 \xleftarrow{\lambda} M_0 \xrightarrow{\rho} M_1'$ (see Fig. 6).

Note that the structure of M_2 is partitioned into renamed types of M_1 and true augmentations in $Q(M_1) - M_1$ (cf. Sect. 3.1). In the example, Person is renamed (new name is Account) causing persons to be retyped, whereas the enhancement with ① or ② causes true augmentation of instances I_1. Clearly, rebuilding instances typed over M_2 after the evolution is only possible if the application of the graph transformation rule does not delete types of M_1 that shall be renamed. Hence, we consider the following subset relation necessary to allow for successful rebuilding of M_2-instances:

$$v(M_2) \subseteq \lambda(M_0) \cup (Q(M_1) - M_1) \tag{1}$$

Let's assume that we have applied the rule in the left column in Fig. 4. Then it is possible to interpret (λ, ρ) again as a graph transformation rule and *emb* as a match of its left hand side M_1. If we compute the corresponding direct derivation, then we obtain two pushouts as in the left part of Figs. 4 and 6.

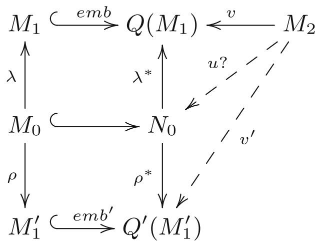


Fig. 6. Double pushout

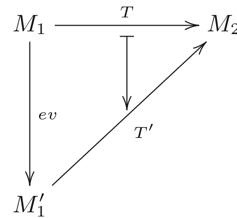


Fig. 7. Transformation co-evolution

Our goal is to construct an evolved transformation

$$T' := (M_1' \xhookrightarrow{emb'} Q'(M_1') \xleftarrow{v'} M_2)$$

Obviously this is possible, if in Fig. 6, there is a morphism $u : M_2 \to N_0$, such that $u; \lambda^* = v$. This equation means that $v' := u; \rho^*$ represents the new retyped

version of the new augmented structure $Q'(M_1')$ and that it acts in the same way as v on the parts that are preserved by the evolution. Simple formal arguments show that u exists, if $v(M_2) \subset \lambda^*(N_0)$. But this condition is fulfilled, if (1) holds, because $N_0 = M_0 \cup (\lambda^*)^{-1}(Q(M_1) - M_1)$ by the pushout property of the top left square in Fig. 6. We summarize this as a proposition:

Proposition 1. *If M_1 evolves to M_1' as described above and if condition (1) holds, then T can automatically be co-evolved to T', cf. Fig. 7.* □

This co-evolution is depicted in Fig. 6: Query specification is now an embedding emb' of M_1' into $Q'(M_1')$. Since there are two pushout squares, the augmentation still consists of the two new associations labelled ① and ② of $Q(M_1)$ in Fig. 4. Retyping (v') is the same as before, but v' now specifies that inheritance relations in M_1' are excluded from the target.

Note that the graph transformation rules can sequentially be applied, each application yielding a co-evolved transformation. *Sequentially* means to first rewrite only the ATL-code-part concerning the employee-position and in the second application the part for the client-position. The final deletion of position causes no further change to the new ATL code.

4 Co-evolution Using Traceability Models

We will now build upon the example in Sect. 2 and outline an implementation of the proposed formal approach to solve the co-evolution process in model transformations. Figure 8 represents the complete picture of the situation, where the transformation Company2CRM is represented as a traceability model between the source and target models. The traceability model Q(CompanyM) shows how the source model is augmented and the repeated labels on the arrows (e.g. ③) indicate the mappings from the target model CrmM. This follows the Query and Retyping Execution pattern explained in Sect. 3. In the bottom left side, we report the EvoCompanyM model which is the result of the evolution explained in Sect. 2, applied on CompanyM. This is achieved by applying the rule in Fig. 4 two times (App1R1 and App2R1), before applying the deletion rule (App1R2) which deletes the enumeration type and the position attribute. This evolution is implemented in the Edelta [1] language, which is explained in Subsect. 4.2. Finally, the model Q(EvoCompany) shows the automatically generated augmented model of the EvoCompany. This augmented model with the mappings from CRM (which are depicted by the labels on the arrows) represents the co-evolved transformation as a traceability model.

Figure 9 depicts the procedure for automatically deriving the *Evolved Traceability* model (labelled Ⓒ) from the *Transformation Traceability* model (labelled with Ⓐ) and the *Evolution Traceability* model (labelled Ⓑ). This procedure starts with a traceability model generated from ATL transformation rules using the AnATLyzer [2] tool. This tool analyses ATL transformations in order to spot possible inconsistencies and produces a detailed view of the traceability links between the models. Although the presented approach has been tested with ATL

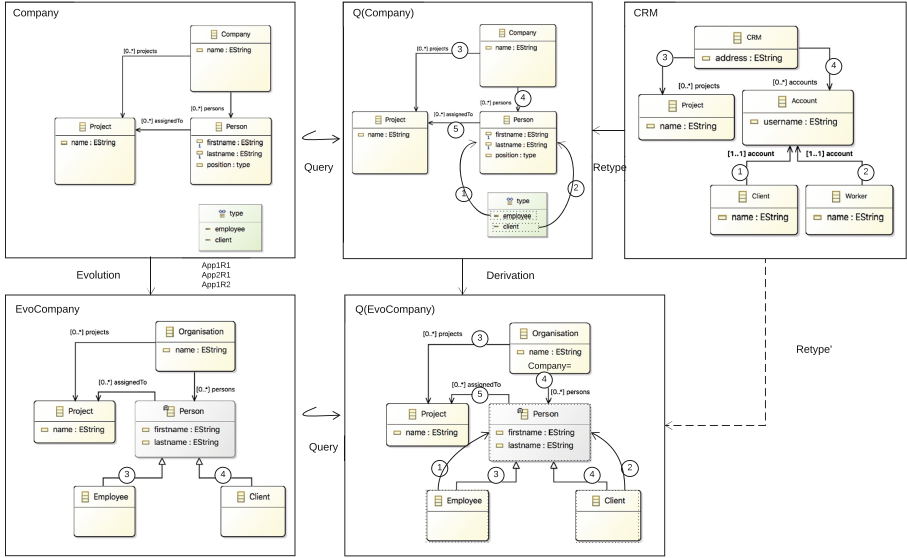


Fig. 8. Co-evolution scenario in the running example

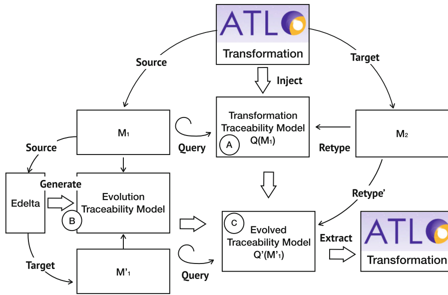


Fig. 9. Procedure

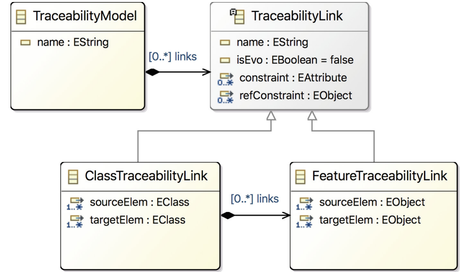


Fig. 10. Traceability metamodel

transformations, it can be extended to other transformation languages such as ETL [19]. The Evolution Traceability model embodies the traceability information about how the model M_1 is evolved into M_1'.

Our traceability model conforms to a traceability metamodel (inspired by [8]); an excerpt of which is shown in Fig. 10. The TraceabilityModel contains a specification of two different types of traceability links: (i) a ClassTraceabilityLink representing links between classes, and, (ii) a FeatureTraceabilityLink representing links between features belonging to these classes. We differentiate between the links representing a transformation rule and the links representing an evolution by the attribute isEvo. A traceability link can be specified also among features which may lead to complex problems that arise from the introduction of OCL expressions [22]. In the next subsections we detail the components of Fig. 9.

4.1 Representing Transformation as Traceability Model

In this section we explain how we represent traceability models with a particular instantiation of the traceability metamodel in Fig. 10. The procedure of generating the traceability model from the produced traceability links can be summarised as follows: for each rule of the transformation, the input and output patterns are linked and internally the features of the classes in the patterns are connected when the bindings are specified [16].

Figure 11 depicts the transformation traceability model containing the information extracted from the transformation shown in Listing 1. The central panel shows the Company2CRM transformation which is composed of 4 matched rules, where the highlighted TraceLink represents the Person2Worker matched rule. The left panel shows the source model of the Company2CRM transformation, where the Person class is automatically linked as input source pattern with a constraint on the position attribute that has to be set to employee in order to produce a Worker and an Account instance, which is highlighted in the right panel. This is expressed using the concepts constraint and refConstraint for identifying the element where the constraint is applied (e.g., the attribute position) and for expressing the condition (e.g., the literal to be employee), respectively.

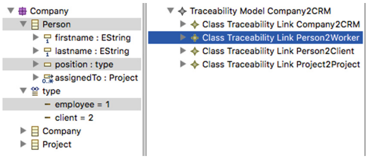


Fig. 11. Traceability model for Company2CRM

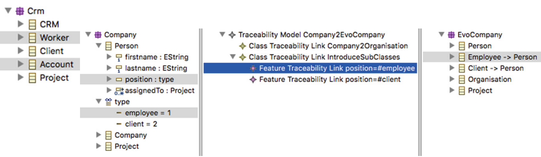


Fig. 12. Traceability model for Company2EvoCompany

In Fig. 12 the evolution example is represented using the same concepts from the traceability metamodel. In general, if the type of the link is specified with isEvo, then a traceability link declares how the concepts in the original model have been evolved. In fact, two patterns of refactorings have been reported in the central panel in which two evolution traceability links are shown. The highlighted one represents the internal part of the introduction of subclasses, where the position attribute, and the enumeration literal employee in the original model are linked to the Employee class in the evolved one (same link for Client). The evolution of the original model can be specified directly using our traceability tool (Fig. 12) or it can be automatically generated from the existing model evolution specification in Edelta [1], which is briefly clarified in the sequel.

4.2 Specifying Model Evolution with Edelta

Edelta is a domain specific language for specifying and applying generic model refactorings. The language allows the modeller to specify both atomic and complex changes. Edelta provides modellers with an extension mechanism enabling

the use of already developed refactorings which can be organized in generic libraries. An example is shown in Listing 3.

```
1 package it.gssi.refactorings
2 metamodel "ecore"
3 def renameMetaclass(EClass c, String newname){
4   c.name=newname;
5 }
6 def introduceSubclasses(EAttribute attr,EEnum attr_type, EClass
      containingclass){
7   containingclass.abstract=true;
8   val subclasses =attr_type ;
9   for(subc:subclasses.ELiterals){
10    containingclass.EPackage.EClassifiers+=newEClass(subc.literal.toString
          .toFirstUpper)
11      [
12      ESuperTypes+=containingclass;
13      ];
14      containingclass.EStructuralFeatures-=attr;
15   }
16 }
```

Listing 3. Edelta definition for generic renaming and introduction of subclasses

The listing shows the definition of the generic refactorings `renameMetaclass` and `introduceSubclasses` which are part of a catalogue of refactorings documented in [29]. Listing 4 reports an Edelta program, using the Edelta library `refactorings` (see line 3) originally defined in Listing 3. The evolution in the case study outlined in Sect. 2 is now specified in lines 4–8, where the refactoring definitions `renameMetaclass` and `introduceSubclasses` form Listing 3 are invoked. The model subject to the modifications is declared at line 1. Moreover, Edelta provides a construct, namely `ecoreref`, that is responsible for accessing the actual model elements on which the program is operating, e.g., in lines 5 and 8. The Edelta tool is able to generate the evolved model by running this Edelta program. However, in this paper we use the Edelta program for generating the evolution traceability model (labelled Ⓑ in Fig. 9).

```
1 metamodel "CompanyM"
2 metamodel "ecore"
3 use MMRefactorings as refactorings
4 //Renaming the metaclass Company to Organisation
5 refactorings.renameClass(ecoreref(Company),'Organisation')
6 // Introduce subclasses from the attribute position of the metaclass Person
7 changeEClass CompanyM.Person{
8   refactorings.introduceSubclasses(ecoreref(Person.position),ecoreref(
        CompanyMM.^type) as EEnum, it)
9 }
```

Listing 4. Edelta program for the running example

4.3 Deriving the Evolved Traceability Model

In this section, we explain the process of deriving the evolved traceability model (labelled Ⓒ in Fig. 9). The input of this process is the original *Transformation Traceability* model Ⓐ and the *Evolution Traceability* model Ⓑ, which are both represented as instances of the metamodel in Fig. 10. We create first an empty

Evolved Traceability model. Then we iterate through all the traceability links and look for links where the sources are the same.

That is, the source of the transformation traceability link is involved in an evolution. For example, the source of the transformation traceability link Person2Worker in Fig. 11 is also the source of the IntroduceSubclass evolution specification, expressed with a traceability link in Fig. 12. The *ResolvePattern* function is responsible for manipulating the traceability link according to the found evolution pattern. For example, the resulting traceability link highlighted in Fig. 13 is generated in such a way that the source of the current link is now Employee.superType = Person instead of Person.position = #employee.

The result of the execution of this process on the transformation Company2CRM is the evolved traceability model shown in Fig. 13. The highlighted traceability link shows that the source of the rule has been correctly fixed to Employee and the rule has been renamed as $evolved_Person2Worker$, to distinguish which rules have been co-evolved. The log, shown in Fig. 14, lists the steps of the procedure where the red lines show three traceability links found in the evolution specification—once *RenameClass* and twice *IntroduceSubclass*—; it reports also that these patterns have been found in the source model of the transformation, in particular in the affected transformation rules Company2CRM, Person2Client and Person2Worker. After the pattern resolution, it saves the evolved traceability model and concludes the procedure.

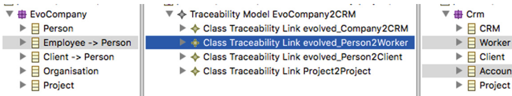


Fig. 13. Result of the co-evolution

```
Company2CRM traceability model loaded
Company2EvoCompany traceability model loaded
Reading: Company2CRM traceability model
Found pattern {REPLACE_CLASS=SRC} on Company2CRM
Found pattern {INTRODUCE_SUBCLASSES=SRC} on Person2Client
Found pattern {INTRODUCE_SUBCLASSES=SRC} on Person2Worker
Saving the evolved Traceability Model...
Saved.
```

Fig. 14. Log of co-evolution

From Traceability Model to ATL. The generated evolved traceability model can be extracted to create ATL transformation rules using a model-to-text transformation approach similar to [7]. The transformation generation translates the traceability model into a transformation model and with an higher-order transformation (HOT) the rules and bindings are produced. The generated ATL transformation will correspond to the one in Listing 2.

5 Related Works

Co-evolution is widely treated in literature in relation to different artefacts corrupted because of model evolution [15,27,28,30,31,34]. In this section, we explore some related approaches specific to co-evolution of model transformations. Each approach presents a different methodology, usually based on preprocessing the model changes and reacting to adapt the artefacts.

Meńdez et al. [27] introduce the notion of domain conformance: the relationship between a transformation and its domain model that is corrupted during the

evolution scenario. They proposed a transformation co-evolution process based on impact analysis and the suggestion of a set of actions to re-establish the affected relationship. This work is limited to the impact analysis and suggestion of co-evolution without a complete automation like what we obtain by the evolved traceability model.

Di Ruscio et al. [6] discuss the problem of co-adapting models, transformations, and tools. They propose a characterization of different aspects involved co-evolution problems with the goal of clarifying the difficulties and the basic requirements for possible unifying solutions. In [5,32] a feature-based approach is used for variability exploration and resolution in models and model transformation co-evolution. This approach uses feature models to assist the user in selecting a suitable co-evolution when multiple solutions are available. This approach relies on a specific DSL for migration called EMFMigrate that is able to automate the co-evolution process, driven by the user specification. In our approach we perform the resolution by an algorithm that can be modified in order to satisfy different users' needs and tools.

In [13] an approach to transformation co-evolution is proposed where the process is divided into two main stages: detection stage, where the changes to the domain model are detected and classified, and co-evolution stage, where the required actions for each type of change are performed. Kruse [21] present an operator-based approach to support co-evolution of models and model transformations. The approach allows the description of changes applied to a domain model and the automatic or semi-automatic resolution of the impact on related model transformations. Our co-evolution approach is different in the usage of traceability models which uniformly treats different transformation languages as long as traceability models can be created from the transformation rules.

The approach in [23] proposes a set of atomic model changes which are able to describe arbitrary model evolutions. It supports reusability and extensibility by means of change composition, and provides resolution actions for both models and transformations changes ensuring an intra-artefact consistent co-evolution of models and transformations. The resolution actions resemble our procedure, however, the formalisation with traceability models is more generic in the sense that it does not rely on a predefined set of model changes.

A formal treatment of the evolution of model transformations by model refactoring was proposed in [10] using graph transformations not only to specify the evolution, but also the model transformation itself. In this paper, evolution rule's left-hand sides govern the migration of the model transformation rules. Since both the evolution and the transformation are based on the same formal technique, precise conditions can be given for successful migration. However, their approach requires to construct a comprehensive type graph, which includes concepts from both the source and the target metamodel of the transformation, together with correspondence links between them. In our approach these correspondence links are not necessary. Moreover, the query-retyping-approach abstracts away from the concretely used transformation language and one is not forced to formalize the transformation in the same way as the evolution.

6 Conclusions

We have proposed a formal approach to automatically evolve model transformations to keep it in sync with the evolved model. The approach is based on encoding the model transformation definition as a traceability model and the evolution of the models as applications of graph transformation rules. By applying the rules, we obtain an evolved traceability model from the original one, that can in turn be converted into an evolved version of the transformation definitions. We have also implemented a prototype tool for co-evolving model transformations written in ATL.

We plan to test the approach with other transformation languages. Moreover, we will characterise the cases when the approach is applicable according to the definition of breaking-resolvable and breaking unresolvable changes [4]. Our approach has also some limitations which deserve further investigations: (i) determine which of the refactorings presented in [29] can be represented as graph transformation rules; (ii) further development of the prototype; (iii) extend the approach to deal with evolution of the target of the transformation definition.

References

1. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta: an approach for defining and applying reusable metamodel refactorings. In: ME workshop @MoDELS (2017)
2. Cuadrado, J.S., Guerra, E., de Lara, J.: Static analysis of model transformations. IEEE Trans. Softw. Eng. **43**(9), 868–897 (2017)
3. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. Theor. Comput. Sci. **376**(3), 139–163 (2007)
4. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Dealing with the coupled evolution of metamodels and model-to-text transformations. In: ME workshop @MoDELS, pp. 22–31 (2014)
5. Di Ruscio, D., Etzlstorfer, J., Iovino, L., Pierantonio, A., Schwinger, W.: Supporting variability exploration and resolution during model migration. In: Wąsowski, A., Lönn, H. (eds.) ECMFA 2016. LNCS, vol. 9764, pp. 231–246. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42061-5_15
6. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: Proceedings of the 2nd IWMCP 2011, pp. 30–38. ACM (2011)
7. Didonet Del Fabro, M., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. SoSyM **8**(3), 305–324 (2009)
8. Diskin, Z., Gómez, A., Cabot, J.: Traceability mappings as a fundamental instrument in model transformations. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 247–263. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_14
9. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and Kleisli categories. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 163–177. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28872-2_12

10. Ehrig, H., Ehrig, K., Ermel, C.: Refactoring of model transformations. In: ECE-ASST, vol. 18 (2009)
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. MTCSAES. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
12. Freund, M., Braune, A.: A generic transformation algorithm to simplify the development of mapping models. In: MoDELS, pp. 284–294. ACM (2016)
13. García, J., Diaz, O., Azanza, M.: Model transformation co-evolution: a semi-automatic approach. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 144–163. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36089-3_9
14. Hermann, F., Ehrig, H., Taentzer, G.: A typed attributed graph grammar with inheritance for the abstract syntax of UML class and sequence diagrams. ENTCS **211**, 261–269 (2008)
15. Herrmannsdoerfer, M.: COPE – a workbench for the coupled evolution of metamodels and models. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 286–295. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_18
16. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
17. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. In: Denney, E., Bultan, T., Zeller, A. (eds.) 28th International Conference on Automated Software Engineering, pp. 191–201. IEEE (2013)
18. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: Van Gorp, P., Engels, G. (eds.) ICMT 2016. LNCS, vol. 9765, pp. 173–188. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42064-6_12
19. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69927-9_4
20. Kruse, S.: On the use of operators for the co-evolution of metamodels and transformations. In: ME Workshop @MoDELS (2011)
21. Kruse, S.: Co-Evolution of Metamodels and Model Transformations. Books On Demand, Norderstedt (2015)
22. Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schoenboeck, J., Schwinger, W., Wimmer, M.: Systematic co-evolution of OCL expressions. In: 11th APCCM, 27–30 January 2015
23. Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., Schönböck, J.: Consistent co-evolution of models and transformations. In: Lethbridge, T., Cabot, J., Egyed, A. (eds.) MoDELS, pp. 116–125. IEEE Computer Society (2015)
24. Levendovszky, T., Balasubramanian, D., Narayanan, A., Karsai, G.: A novel approach to semi-automated evolution of DSML model transformation. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 23–41. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12107-4_4
25. Lopes, D., Hammoudi, S., Bézivin, J., Jouault, F.: Mapping specification in MDA: from theory to practice. In: Konstantas, D., Bourrières, J.-P., Léonard, M., Boudjlida, N. (eds.) Interoperability of Enterprise Software and Applications, pp. 253–264. Springer, Heidelberg (2006). https://doi.org/10.1007/1-84628-152-0_23
26. Marschall, F., Braun, P.: Model transformations for the MDA with BOTL. Technical report, University of Twente (2003)

27. Méndez, D., Etien, A., Muller, A., Casallas, R.: Towards transformation migration after metamodel evolution. In: ME Workshop @MoDELS (2010)
28. Meyers, B., Vangheluwe, H.: A framework for evolution of modelling languages. Sci. Comput. Program. **76**(12), 1223–1246 (2011)
29. MOLABO Research Group. The Metamodel Refactorings Catalog. University of L'Aquila - Gran Sasso Science Institute. http://www.metamodelrefactoring.org
30. Rose, L.M., Herrmannsdoerfer, M., Mazanek, S., Van Gorp, P., Buchwald, S., Horn, T., Kalnina, E., Koch, A., Lano, K., Schätz, B., Wimmer, M.: Graph and model transformation tools for model migration. SoSyM **13**(1), 323–359 (2014)
31. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model migration with epsilon flock. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 184–198. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13688-7_13
32. Di Ruscio, D., Etzlstorfer, J., Iovino, L., Pierantonio, A., Schwinger, W.: A feature-based approach for variability exploration and resolution in model transformation migration. In: Anjorin, A., Espinoza, H. (eds.) ECMFA 2017. LNCS, vol. 10376, pp. 71–89. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_5
33. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley, Boston (2008)
34. Wagelaar, D., Iovino, L., Di Ruscio, D., Pierantonio, A.: Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 192–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30476-7_13

Bidirectional Method Patterns for Language Editor Migration

Enes Yigitbas[1(✉)], Anthony Anjorin[1], Erhan Leblebici[2], and Marvin Grieger[1]

[1] Paderborn University, Paderborn, Germany
{enes.yigitbas,anthony.anjorin,marvin.grieger}@upb.de
[2] Technische Universität Darmstadt, Darmstadt, Germany
erhan.leblebici@es.tu-darmstadt.de

Abstract. Language editors play an important role in a Model-Driven Engineering context, as they enable the productive use of Domain Specific Languages (DSLs). To support language editor development, numerous language editor frameworks exist including extensible UML tools such as Enterprise Architect and textual language editor frameworks such as Xtext. When maintaining DSL-based software systems, language editor migration is an important task, which can be well supported with bidirectional transformation (bx) languages. There currently exists, however, no systematic guidelines describing why, when, and how bx languages can be leveraged for language editor migration. In this paper, therefore, we analyse the problem and solution domains for language editor migration, identifying and describing a set of reusable solution strategies that support assessing the potential and advantages of using bx languages in this context.

1 Introduction

Model-Driven Engineering (MDE) is a software engineering approach that considers models as first-class citizens to be used throughout all engineering disciplines and in any application domain. As *models* are suitable abstractions of a (software) system, a complete system will be typically defined by *multiple* models, each of which is an element of a Domain Specific Modelling Language (DS(M)L). A *modelling language* is a set of all possible models that conform to the modelling language's abstract syntax, are represented by a concrete syntax, and that satisfy a given semantics. The *abstract syntax* of a modelling language defines all concepts and their respective relations in the language. The *concrete syntax* of a modelling language refers to its notation, i.e., the manner in which users will view and specify models of the language [8].

As the MDE approach explicitly encourages the use of possibly numerous DSLs, *language editors* with which models in the concrete syntax(es) of the DSLs can be specified, play a central role. To use a DSL productively, end users require a suitable language editor that is comparable in functionality to established IDEs for general purpose languages. As a consequence, numerous

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 97–114, 2018.
https://doi.org/10.1007/978-3-319-92997-2_7

language editor frameworks exist to support the development and maintenance of language editors. These include *visual* language editor frameworks such as Enterprise Architect, GMF, Sirius, and Graphiti, as well as *textual* language editor frameworks such as Xtext and MPS. The former are used to develop *visual* language editors supporting a visual concrete syntax, while the latter are used to develop *textual* language editors supporting a textual concrete syntax.

As part of the maintenance of a DSL-based software system, language editor migration is a crucial and common task, especially as available language editor frameworks often evolve rapidly. The main drivers for language editor migration are basically the same as for software migration in general: technological obsolescence of the currently used language editor framework, inability to realise new emerging requirements, or loss of knowledge on how to efficiently evolve the existing language editor [4].

A *model transformation* translates a set of input models into a set of output models. For this paper, it suffices to consider *unidirectional* model transformations with exactly one input model, the *source model*, and one output model, the *target model*. We thus have unidirectional *forward* model transformations going from source to target, and *backward* model transformations going from target to source [25]. In some cases, a given pair of unidirectional forward and backward transformations are coupled in the sense that they cannot be changed independently of each other. This coupling can be formalised either via roundtripping laws [9], or by specifying an underlying consistency relation over pairs of source and target models that both the forward and backward model transformations must respect [28]. Let us refer to such a coupled pair of forward and backward transformations as a *bidirectional model transformation (bx)*.

While a bx can be realised (to a certain extent) by combining two separately specified forward and backward unidirectional model transformations, there are numerous *bx languages* with which a bx can be specified as a *single* program. Expected advantages include improved maintainability, and support for a wider range of *consistency management* operations such as model generation, consistency checking, and traceability link creation. Prominent examples of actively developed bx languages include grammar-based approaches such as Triple Graph Grammars (TGGs) [26], constraint-based approaches such as JTL [6], and combinator-based approaches such as BiGUL [19]. The interested reader is referred to existing surveys on bx languages [18,27] for further details. Bx is relevant for various application scenarios in diverse domains including databases (e.g., inverting queries and solving the view-update problem), supporting concurrent engineering by synchronising related software artefacts, and providing improved tools for program language analyses and compiler development [7].

Based on the accumulated experience of maintaining and migrating an MDE tool eMoflon (and its predecessor MOFLON) [22] for many years, we have recognised that bx languages can also be used to support the task of language editor migration. Intuitively, this is because two distinct concerns have to be addressed. On the one hand, existing models specified in the concrete syntax supported by

the legacy language editor must be *migrated* to the new language editor. On the other hand, a means of *generating* the abstract syntax from the concrete syntax of models specified with the new language editor must be established to use it. In this context, bidirectional model transformations address both concerns simultaneously. To fully leverage bx languages, however, an *a priori* understanding of the application scenario and the corresponding solution space is required. Without an upfront decision and a rough plan of how to apply bx languages, somehow extending an implemented unidirectional to a bidirectional model transformation as an *a posteriori* consideration can lead to an increase in effort and reduction in quality. There currently exist, however, no systematic guidelines on the usage of bx languages for language editor migration.

Our contribution in this paper is thus to analyse the problem and solution domains for language editor migration, identifying and describing a set of *bidirectional method patterns*, i.e., reusable solution strategies to support assessing the potential and advantages of leveraging bx languages in this context. Our analysis is presented using problem and solution domain description languages proposed in our previous work in the industry automation domain [3].

The rest of the paper is structured as follows: Sect. 2 presents our running example for the entire paper – a recent language editor migration project for a part of eMoflon[1], an MDE tool that provides languages for specifying unidirectional and bidirectional model transformations. The migration was performed for the bidirectional part, and involved the transition from a visual language editor implemented with Enterprise Architect (EA)[2], to a textual language editor implemented with Xtext[3]. Section 3 presents our analysis of the problem domain, while Sect. 4 identifies a series of different strategies (patterns) in the solution domain. To demonstrate the benefit of our identified patterns, Sect. 5 applies them to structure the discussion of advantages and trade-offs of the final strategy chosen and implemented in our running example. Related approaches are discussed in Sect. 6, while Sect. 7 concludes.

2 Migrating eMoflon from Enterprise Architect to Xtext

The meta-modelling and model transformation tool eMoflon is the current version of its predecessor MOFLON [1]. Development of MOFLON started in 2002, with a first stable release (1.0) in 2006 [2]. Since then the tool has been maintained and regularly migrated for diverse reasons ranging from standardisation efforts to a shift in research focus from code generation to (bidirectional) model transformation. For more details the interested reader is referred to [2].

An important change implemented with the shift from MOFLON to eMoflon in 2011 was the migration to EA, i.e., establishing a professional UML tool as a new language editor [2]. While this has been successful, especially in an industrial context, a series of drawbacks emerged over the years for both end

[1] www.emoflon.org.
[2] http://www.sparxsystems.com/.
[3] http://www.eclipse.org/Xtext/.

users and developers including (i) the extra hurdle of applying for academic and campus licences, (ii) having to master an additional complex tool environment, and (iii) spending substantial human resources as the required tool chain for EA is orthogonal to Eclipse plugin development. Establishing an Xtext-based textual language editor for eMoflon provides an adequate solution: students only need to install a single open-source Eclipse plugin and can use an editor that feels familiar from previous lectures on Java and Eclipse. As eMoflon developers are familiar with graph grammars (a generalisation of string grammars), they already posses the right skill set for Xtext development. Finally, tailoring of Xtext-based editors is in Java/EMF and thus requires no extra dependencies. Figure 1 depicts the migration of eMoflon from EA (baseline scenario above) to Xtext (envisioned scenario below) schematically as a component diagram.

The EA-based language editor takes its input in a visual concrete syntax ❶ specified as diagrams that the end user can manipulate. An example of such a diagram (a TGG rule) is depicted in the upper left corner of Fig. 1. These diagrams are persisted in a database that can be accessed via SQL queries. EA provides an internal metamodel for all diagrams and the corresponding abstract syntax ❷ can be extracted from this database. Diagrams are basically simple tree-like, rather generic structures, with cross-tree references realised via unique identifiers. An excerpt of this EA abstract syntax for the same TGG rule is depicted in the upper right corner of Fig. 1. Note that only the portion of the diagram in the visual concrete syntax ❶ with a grey background is depicted in the EA abstract syntax ❷. To map this to an abstract syntax ❸ that eMoflon (the backend) can process, an EA export was implemented. The corresponding model fragment for our TGG rule excerpt (again only the portion with a grey background) is depicted in the lower right corner of Fig. 1. Note the explicit connectivity and TGG-specific concepts such as "domains" that result from an interpretation of attributes in the generic, tree-like EA abstract syntax.

In the envisioned scenario, an Xtext-based editor takes as input a textual concrete syntax ❹ for TGGs. The corresponding textual fragment for our TGG rule excerpt is depicted in the lower left corner of Fig. 1. The entire TGG rule is depicted and the lines with a grey background correspond to the portion of the visual concrete syntax ❶ with a grey background. The output of the Xtext-based editor ❺ is an EMF-compatible model that must be transformed to the eMoflon abstract syntax.

The goal of this paper is to provide a systematic means of classifying and discussing relevant requirements of such a migration project and the corresponding consequences for feasible solutions.

3 Problem Domain: How Are the Editors to Be Used?

In this section, we analyse the problem domain for language editor migration by identifying and describing possible usage scenarios of both editors before and after the migration. We view each scenario as a *consistency restoration* task and formalise it using a description language for consistency management scenarios

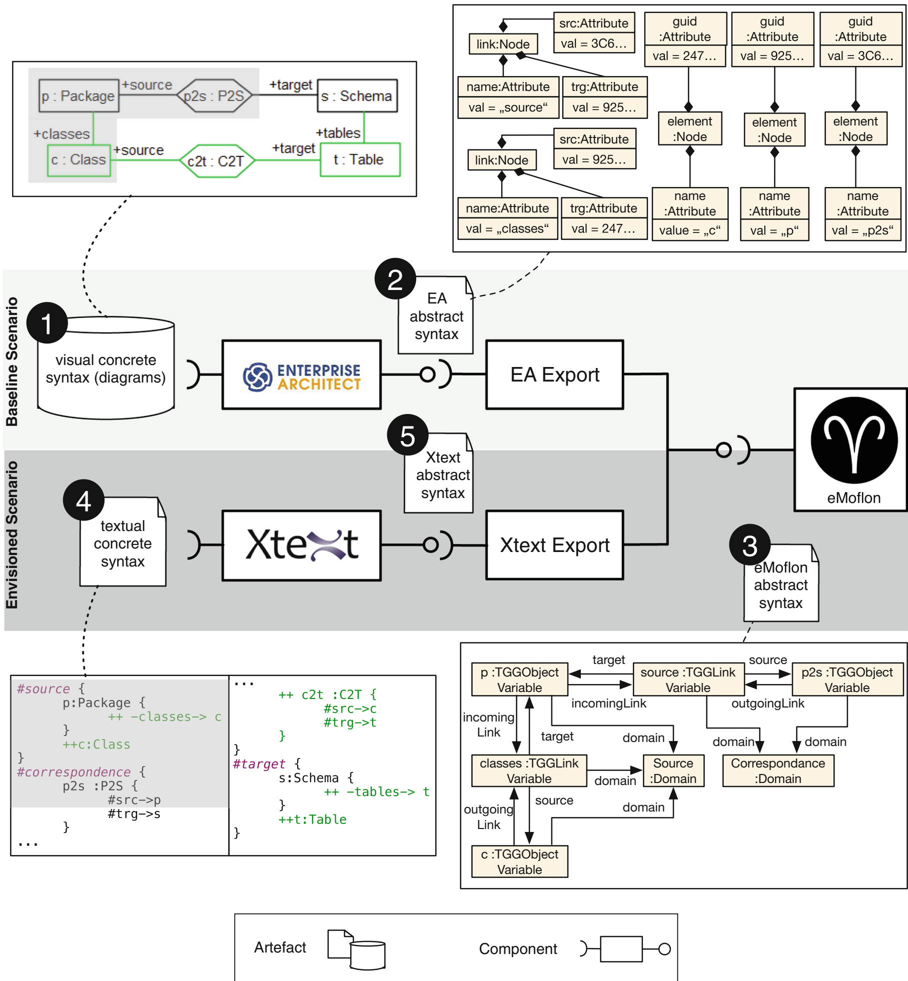


Fig. 1. Current and envisioned scenario.

proposed by Stevens [29], and extended in our previous work in the industry automation domain [3]. We refer interested readers to Anjorin et al. [3] for full details and present the minimum required for this paper in the following.

The first step is to specify a common *transformation context* consisting of types (depicted as curved rectangles) and consistency relations (depicted as edges between types). The transformation context for language editor migration is depicted in Fig. 2. The types are grouped into two layers (represented as swim lanes in the diagram): the artefact layer and the model layer. In this context, *artefacts* are arbitrary representations of data, while *models* are compatible with a chosen modelling framework. We identify three types: `Legacy` representing

artefacts used by the legacy editor, `New` representing artefacts used by the new editor, and `Model` representing the common representation on the model layer.

There are also three consistency relations: `LegacyToModel` consisting of consistent pairs of models and legacy artefacts, `ModelToNew` consisting of consistent pairs of models and new artefacts, and `LegacyToNew` representing a consistency relation that can be defined for pairs of legacy and new artefacts. Note that the (practical) means of defining the consistency relations is left open and can be via constraints, rules, consistency restoration programs, etc.

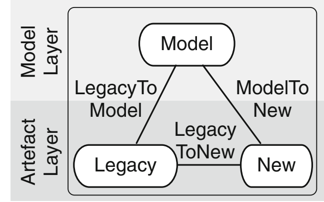


Fig. 2. Transformation context for editor migration

Example: For our running example, legacy artefacts are EA diagrams in a database, new artefacts are textual files, and (EMF) models are in the abstract syntax expected by the eMoflon backend.

Based on this transformation context, we now discuss relevant usage scenarios of both legacy and new editors, before and after the migration. There are basically two different scenarios: (i) a one-time migration is feasible, i.e., work with the legacy editor can be prohibited as soon as the migration is performed, and (ii) for some time after the migration (possibly indefinitely), working with the legacy editor still has to be supported in some way.

3.1 Complete Switch from Legacy to New Editor

Before discussing what kind of migration is required, it is helpful to specify how the legacy editor is currently being used. The different possibilities are represented as *resolution paths* consisting of a sequence of *transformation networks*. A transformation network consists of objects and links typed over the types and consistency relations in the transformation context, respectively. Using the same swim lanes as the transformation context, two resolution paths are depicted in Fig. 3. Resolution paths are denoted visually as "story boards" partitioned into transformation networks by white vertical lines. Each network is a snapshot consisting of all objects and links that are currently relevant. Version numbers are used to indicate what has been changed from one network to the next. Objects with a black fill are *authoritative* and must not be changed in the next step. *Inconsistent* links are depicted as bold dashed red lines.

We are now ready to read the resolution path depicted to the left of Fig. 3, representing *parsing* using the legacy editor: The path starts with a consistent network consisting of a model and legacy artefact both in version 1. In the next step, the legacy artefact is changed to version 2, making it inconsistent with the model. Consistency should only be restored by updating the model to version 2 in the last network. To the right of Fig. 3, a path representing *serialising* with

legacy editor starts with the same initial network. In this path, however, the model is updated to version 2, making it inconsistent with the legacy artefact. Consistency can thus only be restored by updating the legacy artefact.

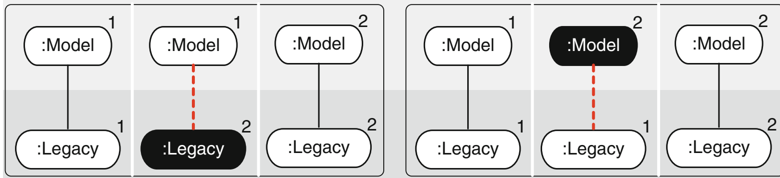


Fig. 3. Usage of legacy editor: parsing (left) and serialising (right)

Parsing is the basic usage scenario that every language editor must typically support, while serialising is typically optional but can be very useful, e.g., to reflect refactorings performed on model level to the artefact level.

Independently of how the legacy editor is being used, the requirement for a *one-time migration* is depicted in Fig. 4. The resolution path starts with a consistent legacy artefact and model, which are not to be changed in the process. In the next step, the new artefact should be created in such a way that the network stays consistent. In a final step, we discard the legacy artefact, i.e., in all subsequent networks it becomes irrelevant for consistency. Note that the second network can be used to capture requirements concerning the consistency between legacy and new artefacts, independent of their consistency to a common model. In practice this might be necessary due to layout, comments, file encoding, and other details that might not be present on the model layer but nonetheless be important for the migration. Finally, although we do not depict this explicitly, the same two resolution paths for parsing and serialising also describe how the new editor is to be used after a one-time migration.

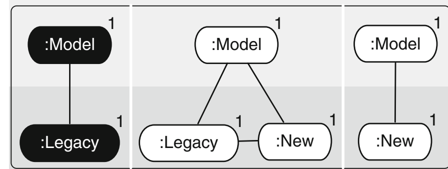


Fig. 4. One-time migration

Example: EA was used mainly for parsing as serialising required an incremental update of the layout of the visual concrete syntax (and we never had sufficient resources to implement and maintain this). The new Xtext editor, however, supports serialising as, in our experience at least, most users accept a standard formatter for a textual concrete syntax. This means that the layout of the *textual* concrete syntax did not have to be incrementally updated as the standard formatter could be used after regenerating the textual artefacts.

3.2 Concurrent Usage of Both Editors

Support for switching regularly between or working concurrently with both the legacy and the new language editor is sometimes an important requirement in practice. It might be impossible to force all stakeholders working on shared models to switch to the new editor at the same time, or it might be a strategic decision to support both editors for a period of time especially when introducing a new concrete syntax. We differentiate between concurrent usage (i) where users can request for and acquire locks for a model, and (ii) where locking is infeasible.

Two resolution paths for concurrent usage with locks are depicted in Fig. 5. In the first path to the left, the legacy artefact has been changed to version 2 making it inconsistent with both the model and the new artefact.

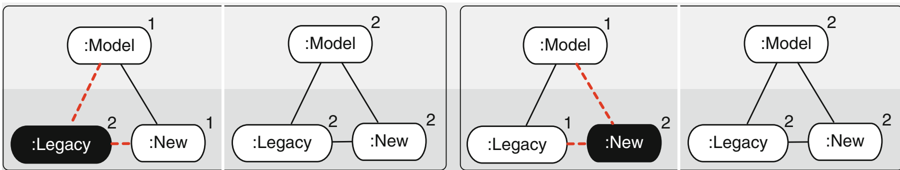


Fig. 5. Concurrent usage with locks

As it is possible to acquire a lock for the model, the changed legacy artefact is made authoritative and cannot be changed when restoring consistency in the following network. Consistency must thus be restored by updating both the corresponding model and the new artefact. In the second path to the right, the new artefact is changed and made authoritative. Consistency must thus be restored by updating both the corresponding model and the legacy artefact.

If locking is infeasible or undesirable, there is no way of ensuring that only changes to one artefact need to be reflected in the model, and the resolution path depicted in Fig. 6 must be supported. In the first network, both artefacts have been changed to version 2 and are inconsistent with the model and each other. As restoring consistency in this case can require changes to all objects (indicated by the version numbers in the final consistent network), no object in the first network is made authoritative.

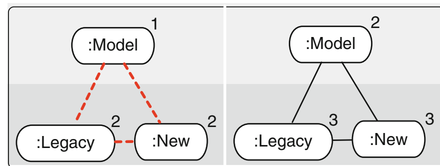


Fig. 6. Concurrent usage without locks

Example: We had a mix of (i) long-term users, who had been using eMoflon for some time already, were already familiar with the visual concrete syntax, and had often already obtained a permanent licence for EA, and (ii) short-term users who were typically students in our lectures, only had an EA licence for

some months, and often had a preference for textual languages. Our requirement was thus concurrent usage of both editors on shared models, and locking was acceptable.

4 Solution Domain: Patterns for Editor Migration

A *method* is used to systemise an endeavour by specifying the activities to perform, artefacts to create, roles to involve and tools to use [11]. In this section, we propose a set of methods on how to perform the transition to a new language editor, related to the scenario introduced in the previous section. To describe the methods in a structured and reusable way, we decompose them into a set of generic method building blocks. These blocks can be composed into a method and can be refined for a specific situation. This approach is based on Situational Method Engineering (SME) [14] principles, which is the discipline that encompasses all aspects of creating a method for a specific situation.

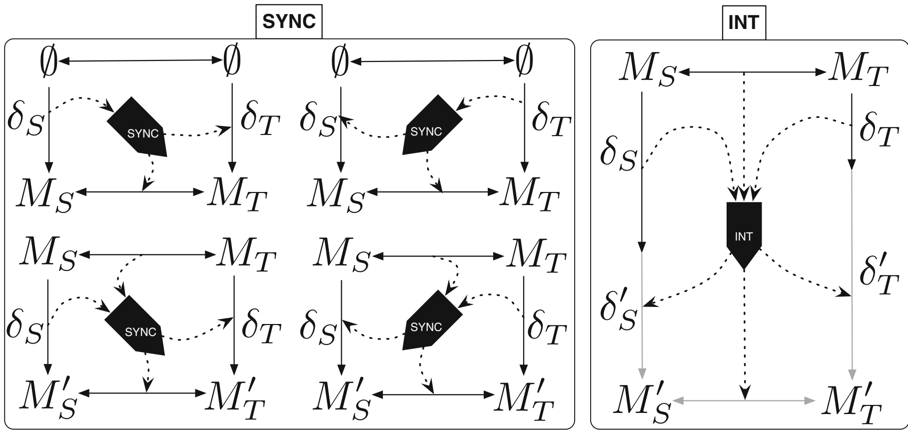


Fig. 7. Relevant method fragments from Anjorin et al. [3]

In previous work [3], we have proposed a set of atomic, reusable building blocks of methods, so called *method fragments*, and used them to describe different solution strategies for consistency management problems. A subset of relevant method fragments for this paper is depicted in Fig. 7. Our fragments consist of four fragments for model *synchronisation* (`SYNC`), and a single fragment for model *integration* (`INT`). Models are denoted by M with a subscript for the domain (S for source, T for target). Empty models are denoted by $\emptyset$. Changes (deltas) applied to a model to yield another model are denoted by vertical arrows with δ together with a domain subscript as label. Correspondences (traceability links) between consistent models in different domains are denoted by horizontal arrows. The activity performed in each fragment is denoted

by a black "wedge" connected to its input deltas/correspondences and output deltas/correspondences by dashed arrows.

The top-left SYNC fragment represents an *initial* forward transformation. The input is a source delta δ_S and the output is a target delta δ_T that produces a consistent target model M_T, and a correspondence between the source and target models. Analogously, the top-right SYNC fragment represents an *initial* backward transformation. The bottom-left and bottom-right SYNC fragments represent *incremental* versions of forward and backward synchronisation; a correspondence between existing models is required as an additional input, and the synchroniser is to restore consistency by incrementally changing the output model.

The single INT fragment to the right of Fig. 7 represents the activity of *model integration*. The input in this case consists of both a source and target delta, as well as a correspondence between existing and consistent source and target models. To restore consistency, an integrator has to resolve possible conflicts and determine output deltas δ'_S and δ'_T leading to consistent models M'_S and M'_T, respectively, and the correspondence between these models. We refer the interested reader to Anjorin et al. [3] for further details.

We are now ready to discuss different solution strategies, referred to as *method patterns*, for language editor migration, formed by composing these model fragments in a suitable manner. If a one-time migration (see Subsect. 3.1) suffices, the method patterns depicted in Fig. 8 are applicable. To adapt the fragments to language editor migration (see the transformation context in Sect. 3), legacy artefacts are denoted by $A_L, A'_L, \ldots$, new artefacts by $A_N, A'_N, \ldots$, and the common model for the backend by $M, M', \ldots$. To reduce diagram clutter, we omit the labels for deltas. Finally, a red circle with a "1" in the top-right corner of a fragment indicates that the fragment is only required as a one-time activity.

The simplest scenario is if the new editor is only to be used as a parser (see Subsect. 3.1). In this case, the method pattern MigrationToParser, depicted to the left of Fig. 8, suffices: the first activity is to forward transform legacy artefacts to models. This is *not* a one-time activity as the legacy editor was already in use as at least a parser (and perhaps a serialiser too). The next activity is to forward transform the models to new artefacts. As the new editor is only used as a parser, this is now a one-time activity that does not need to be repeated. The final activity is to backward transform new artefacts to models, representing the usage of the new editor after the migration. In general this final activity might have to be incremental, either for efficiency reasons or to enable editors that cover only a part of the model.

If the new editor is to be used as a serialiser too, the method pattern MigrationToParserSerialiser, depicted to the right of Fig. 8, is required: the difference compared to MigrationToParser is that there are now no one-time activities, and that an extra *incremental* forward transformation from model to new artefact is required. This additional fragment must be incremental in general to retain information in the new artefact (layout, comments, manual additions, etc.) that is not to be transferred to the model layer.

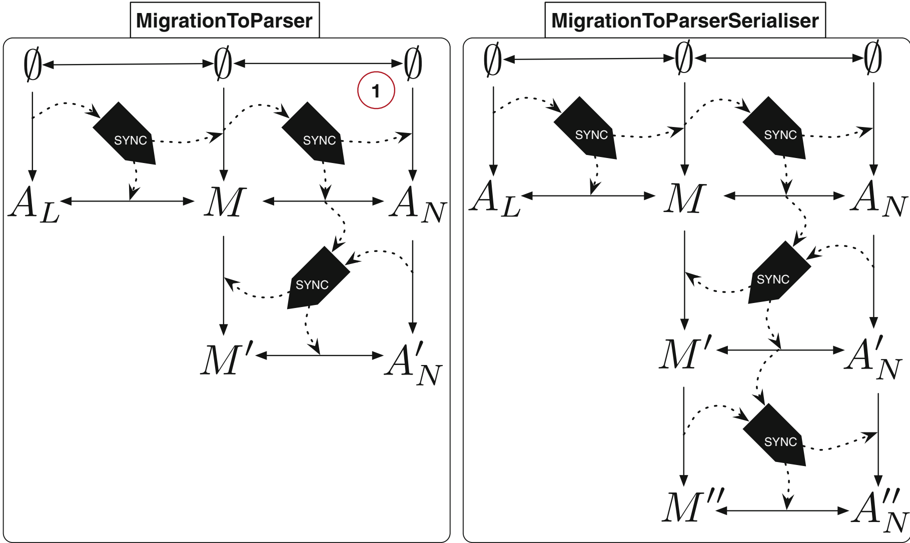


Fig. 8. Method patterns for one-time migration to parser (and serialiser)

To enable concurrent usage of the legacy and new editors, the method patterns depicted in Fig. 9 are required. If locks on the model can be enforced then `ConcurrentUsageWithLocks` suffices: compared to `MigrationToParser-Serialiser` the activity "grid" is now completely filled with additional incremental fragments serialising models back to legacy artefacts and incrementally (re)parsing legacy artefacts back to models. Note that it is now irrelevant if the legacy (new) editor is used as a serialiser or not – repeated (incremental) forward transformations from model to legacy (new) artefact are always necessary.

If locking is infeasible (see Subsect. 3.2), then the rather advanced method pattern `ConcurrentUsage`, depicted to the right of Fig. 9 is required. After an initial migration (analogous to all other method patterns), both legacy and new artefacts can be edited concurrently without locking the model. To restore consistency the pattern proposes lifting the changes to the model layer to produce models M'' and M'. These two models then have to be consolidated using an `INT` fragment to produce M^*, which can then be reflected in both legacy and new artefacts. Interestingly, the `INT` fragment required here is a special case of the general `INT` fragment depicted to the right of Fig. 7, as M_S and M_T in the fragment are the same model M in the pattern, representing a model integration between models in the same domain. Although this special case can be built on top of a standard three-way merge for models (e.g., EMFCompare), it still requires non-trivial conflict detection, conflict resolution, user intervention and (most probably) faces scalability challenges.

Example: In our migration project we attempted to implement `Concurrent-UsageWithLocks` and were able to leverage a bx language (TGGs) for parsing and

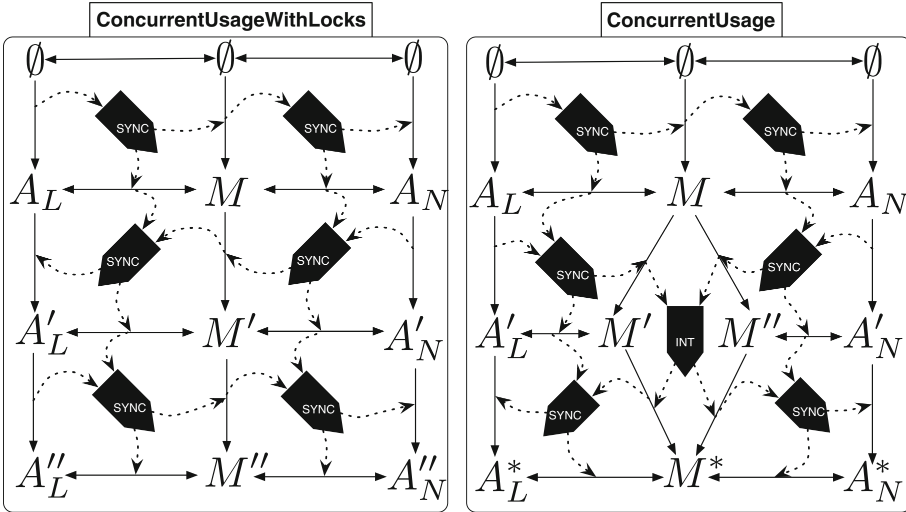


Fig. 9. Method patterns for concurrent usage with and without locks

serialising new artefacts. The export for EA was, however, already implemented as a well-tested, complex unidirectional transformation, so we decided to implement the required import (serialisation) for EA also as a separate unidirectional transformation. This proved to be challenging and was only really successful for class diagrams. One of the main (practical) problems was preserving the layout of the visual concrete syntax. The standard layout in EA was (barely) acceptable for class diagrams and effectively prevented a repeated import of TGG diagrams in practice. Other problems included scalability of the EA API, and the cost of maintaining and adequately testing two separate transformations for parsing and serialising legacy artefacts.

5 Discussion: Leveraging Bx for Editor Migration

In this section, we discuss questions concerning the applicability of our patterns, when and why using a bx language to realise which patterns makes sense, and the current level of automation provided by available state-of-the-art tooling.

5.1 When Do I Need Which Method Pattern?

Figure 10 provides an overview of which method pattern is applicable for which of the usage scenarios identified in Sect. 3. The `MigrationToParser` method pattern can be used to address the *one-time migration* requirement if the new editor is only to be used as a parser. If the new editor is to be used also as a serialiser, then the `MigrationToParserSerialiser` method pattern is required.

Both of these method patterns are generalised by the more advanced pattern `ConcurrentUsageWithLocks` that can be used to support consistent concurrent

Method Pattern	Addressed Usage Scenarios
MigrationToParser	One-time migration when new editor is used only for parsing
MigrationToParserSerialiser	One-time migration when new editor is used for parsing and serialising
ConcurrentUsageWithLocks	One-time migration when new editor is used for parsing and serialising, concurrent usage with locks
ConcurrentUsage	One-time migration when new editor is used for parsing and serialising, concurrent usage with and without locks

Fig. 10. Method patterns and solved scenarios

usage of both editors assuming locks for the model can be acquired and enforced. If locking is infeasible, however, then the method pattern `ConcurrentUsage` is required, which covers all previously mentioned usage scenarios including the concurrent usage of legacy and new editors without requiring locks on the model.

5.2 Why and When Do I Need a Bx Language?

Method patterns are guidelines and do not prescribe the usage of a bx language. All fragments can be implemented as unidirectional model transformations, or even performed manually if required only once for very few artefacts. In all but the simplest method pattern `MigrationToParser`, however, all fragments have to be performed repeatedly implying that a high-level of automation is desirable. We refer to our methods patterns as "bidirectional" as they all contain *pairs* of required fragments for forward and backward transformations. The fragments in each pair are tightly coupled in the sense that they typically cannot be maintained (evolved) separately. While realising such pairs of fragments with a bx language is not mandatory, it can be advantageous as bx languages typically derive multiple fragments (in this case forward and backward) from a *single* specification. Expected advantages include improved maintainability, productivity, and the general quality of the fragments. The *incremental* versions of the fragments are also more challenging to program directly, while bx languages typically support this by automatically exploiting traceability links. Concerning our method patterns, `MigrationToParser` is the only pattern where a bx language is not necessary as the model-to-new-artefact fragment is a one-time activity.

5.3 What is the Current Level of Automation and Availability of bx Tooling for Realising the Method Patterns?

Concerning the method fragments depicted in Fig. 7, all `SYNC` fragments are well-supported by numerous state-of-the-art bx tools. Some bx tools [17,19] require the fragment for incremental forward (backward) synchronisation and derive the fragment for incremental backward (forward) propagation for free. Some bx tools [13,22], typically TGG-based, require a rule-based specification of consistency and automatically derive all `SYNC` fragments from this. The distinction between initial (from scratch) and incremental fragments is useful in practice as

initial fragments can be sometimes realised with a different, optimised strategy. Other bx tools [6] require a constraint-based definition of consistency and are also able to provide all SYNC fragments automatically. Challenges here include learning these languages, achieving adequate scalability in some cases, and exercising full control over the synchronisation process. In contrast, the general INT fragment is still a current research focus of the bx community. This is because it is non-trivial to define precisely what behaviour is desirable and what not. Model integration requires conflict detection and resolution, and probably has to include user interaction in some meaningful way. Nonetheless, some initial attempts already exist: Leblebici et al. [23] are able to handle non-conflicting situations by combining TGGs with ILP solvers, while constraint-based bx tools are typically able to handle model integration, albeit in a non-scalable manner with a fixed global metric such as minimized edit distance. Our method pattern that contains the INT fragment, however, only requires a special case that can be based on a standard three-way merge such as EMFCompare[4].

6 Related Work

There exist numerous (industrial) projects that can be viewed as language editor migration projects and thus be analysed with our usage scenarios and method patterns. In the following we discuss a few that we find particularly relevant.

Blouin et al. [5] report on how a synchronisation layer between textual and visual editors was established using bidirectional transformations in an industrial context. In this case, both textual and visual editors existed and were already in use, but it was impossible to use them concurrently on the same model. The goal of the project was thus to support the usage scenario *concurrent usage with locks* described in Subsect. 3.2. The primary challenge was coping with information loss as neither editor completely covered all parts of the model. Blouin et al. leveraged TGGs as a bx language and were able to exploit its support for deriving incremental fragments required for the method pattern ConcurrentUsageWithLocks.

Hermann et al. [16] present an industrial project concerning the translation of satellite procedures from one textual concrete syntax to another. While the project is not directly motivated by editor usage, Hermann et al. use Xtext for both languages mainly due to its support for both parsing and serialising scenarios (see Fig. 3). To the best of our knowledge, this project actually represents a one-time migration implemented via MigrationToParser and does not require a bx language. Hermann et al. use TGGs, nonetheless, and argue that this enables providing formal correctness guarantees, which was crucial for the project.

Maro et al. [24] report on their experience with integrating graphical and textual editors for domain specific languages based on UML profiles. This project was motivated by the requirement to provide additional textual editors for some developers, corresponding to our usage scenarios *concurrent usage*. Unidirectional transformations were used to implement ConcurrentUsageWithLocks,

[4] https://www.eclipse.org/emf/compare.

and the authors report on challenges and limitations concerning information loss and merging. A bx language could have addressed some of these challenges.

All these projects (and many more) indicate that the domain of language editor migration is indeed relevant in practice. We argue that our usage scenarios and method patterns can help analyse, compare, and reflect on such projects.

Our usage scenarios and method patterns are specified using description languages for consistency management scenarios proposed in our previous work [3] adapting and extending work by Stevens [29] and Engels [11].

The research area of megamodelling provides other viable alternatives as languages that could be used for formalising such patterns. Lämmel's *LAL* language for example, which is based on *MegaL* and has been used to describe basic bx patterns [20], could be used to describe consistency management scenarios.

In the research area of method engineering, there are different *methods* that guide a complex software engineering task by specifying the activities to enact, artefacts to generate, tools to use or roles to involve [11]. A specific manifestation of method engineering is *Situational Method Engineering* (SME) which encompasses all aspects of creating a method for a specific situation [15]. Approaches that follow the SME paradigm consider the situational context in which a method will be applied during the development of the method, so that it can be adapted to the context and is then called situation-specific.

Surveys focussed on bidirectional transformation languages [18,27] discuss relevant properties pertaining to bidirectional model transformations. Results in a similar direction are provided by Eramo et al. [12], and Lano et al. [21] proposing patterns for specifying bidirectional transformations with their tool UML-RSDS. Diskin et al. [10] also suggest a taxonomy for model synchronisation application scenarios and discuss various (types of) examples. To the best of our knowledge, however, no existing work addresses the role, potential and trade-off of bidirectional transformations in the context of language editor migration.

7 Conclusion and Future Work

Language editor migration is a crucial and common task, especially as available language editor frameworks often evolve rapidly. When maintaining DSL-based software systems, the language editor migration task can be well supported with bidirectional transformation (bx) languages. There currently exists, however, no systematic guidelines describing why, when, and how bx languages can be leveraged for language editor migration. We propose a solution to this problem by identifying a set of *bidirectional method patterns* for language editor migration scenarios. We use a recent migration project to show how our patterns can support the structured analysis of the problem domain and help in identifying and describing a set of reusable solution strategies in the solution domain.

Bidirectional transformations are relevant in other domains including language translation, data exchange, and quality assurance. Important future work is to identify bidirectional method patterns for these application domains and investigate the effectiveness and efficiency of the proposed languages for describing consistency management problems. In this context, tool-support should be

established to simplify the correct, systematic specification of consistency related requirements and aspects in the problem domain, as well as method patterns in the solution domain. The envisioned tool support can also provide formal analyses, reuse of method "templates", refactoring, etc. Finally, our description languages can be extended to cover non-functional or quality-of-service (QoS) properties regarding consistency management such as efficiency, scalability, and optimality with respect to an objective function.

References

1. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: a standard-compliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_27
2. Anjorin, A., Lauder, M., Patzina, S., Schürr, A.: eMoflon: leveraging EMF and professional CASE tools. In: Informatik 2011. LNI, vol. 192, p. 281. Gesellschaft für Informatik (GI) (2011)
3. Anjorin, A., Yigitbas, E., Leblebici, E., Schürr, A., Lauder, M., Witte, M.: Description languages for consistency management scenarios based on examples from the industry automation domain. Art Sci. Eng. Program. **2**(3), 1–32 (2018). Article 7
4. Bisbal, J., Lawless, D., Bing, W., Grimson, J.: Legacy information systems: issues and directions. IEEE Softw. **16**(5), 103–111 (1999)
5. Blouin, D., Plantec, A., Dissaux, P., Singhoff, F., Diguet, J.-P.: Synchronization of models of rich languages with triple graph grammars: an experience report. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 106–121. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_8
6. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: a bidirectional and change propagating transformation language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 183–202. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_11
7. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02408-5_19
8. da Silva, A.R.: Model-driven engineering: a survey supported by the unified conceptual model. Comput. Lang. Syst. Struct. **43**, 139–155 (2015)
9. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 21–36. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87875-9_2
10. Diskin, Z., Wider, A., Gholizadeh, H., Czarnecki, K.: Towards a rational taxonomy for increasingly symmetric model synchronization. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 57–73. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_5
11. Engels, G., Sauer, S.: A meta-method for defining software engineering methods. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering. LNCS, vol. 5765, pp. 411–440. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17322-6_18

12. Eramo, R., Marinelli, R., Pierantonio, A.: Towards a taxonomy for bidirectional transformation. In: Di Ruscio, D., Zaytsev, V. (eds.) 2014 SATToSE of CEUR Workshop Proceedings, vol. 1354, pp. 122–131. CEUR-WS.org (2014)
13. Giese, H., Hildebrandt, S., Neumann, S.: Model synchronization at work: keeping SysML and AUTOSAR models consistent. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering. LNCS, vol. 5765, pp. 555–579. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17322-6_24
14. Harmsen, F., Brinkkemper, S., Oei, H.: Situational method engineering for information system project approaches. In: Verrijn-Stuart, A.A., Olle, T.W. (eds.) IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle, pp. 169–194. Elsevier (1994)
15. Henderson-Sellers, B., Ralyté, J., Ågerfalk, P.J., Rossi, M.: Situational Method Engineering. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-41467-1
16. Hermann, F., Gottmann, S., Nachtigall, N., Ehrig, H., Braatz, B., Morelli, G., Pierre, A., Engel, T., Ermel, C.: Triple graph grammars in the large for translating satellite procedures. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 122–137. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_9
17. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K.: GRoundTram: an integrated framework for developing well-behaved bidirectional model transformations. In: Alexander, P., Pasarenau, C.S., Hosking, J.G. (eds.) ASE 2011, pp. 480–483. IEEE Computer Society (2011)
18. Hidaka, S., Tisi, M., Cabot, J., Hu, Z.: Feature-based classification of bidirectional transformation approaches. Softw. Syst. Model. **15**(3), 907–928 (2016)
19. Ko, H.-S., Zan, T., Hu, Z.: BiGUL: a formally verified core language for putback-based bidirectional programming. In: 2016 PEPM, pp. 61–72. ACM (2016)
20. Lämmel, R.: Coupled software transformations revisited. In: van der Storm, T., Balland, E., Varró, D. (eds.) 2016 SLE, pp. 239–252. ACM (2016)
21. Lano, K.C., Alfraihi, H., Yassipour Tehrani, S., Haughton, H.: Patterns for specifying bidirectional transformations in UML-RSDS. In: 2015 ICSEA. IARIA XPS Press (2015)
22. Leblebici, E., Anjorin, A., Schürr, A.: Developing eMoflon with eMoflon. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 138–145. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_10
23. Leblebici, E., Anjorin, A., Schürr, A.: Inter-model consistency checking using triple graph grammars and linear optimization techniques. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 191–207. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_11
24. Maro, S., Steghöfer, J.-P., Anjorin, A., Tichy, M., Gelin, L.: On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience. In: Paige, R.F., Di Ruscio, D., Völter, M. (eds.) 2015 SLE, pp. 1–12. ACM (2015)
25. Mens, T., Van Gorp, P.: A taxonomy of model transformation. ENTCS **152**, 125–142 (2006)
26. 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). https://doi.org/10.1007/3-540-59071-4_45

27. Stevens, P.: A landscape of bidirectional model transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88643-3_10
28. Stevens, P.: Towards an algebraic theory of bidirectional transformations. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 1–17. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_1
29. Stevens, P.: Bidirectional transformations in the large. In: 2017 MoDELS. IEEE Computer Society (2017)

Parallel Model Validation with Epsilon

Sina Madani(✉), Dimitrios S. Kolovos, and Richard F. Paige

Department of Computer Science, University of York, York, UK
{sm1748,dimitris.kolovos,richard.paige}@york.ac.uk

Abstract. Traditional model management programs, such as transformations, often perform poorly when dealing with very large models. Although many such programs are inherently parallelisable, the execution engines of popular model management languages were not designed for concurrency. We propose a scalable data and rule-parallel solution for an established and feature-rich model validation language (EVL). We highlight the challenges encountered with retro-fitting concurrency support and our solutions to these challenges. We evaluate the correctness of our implementation through rigorous automated tests. Our results show up to linear performance improvements with more threads and larger models, with significantly faster execution compared to interpreted OCL.

1 Introduction

Many MDE tools face performance difficulties when dealing with very large models. Scalability has been a long-standing issue with MDE [1]. Moreover, it is a multi-faceted problem, with challenges in model persistence, collaboration, domain-specific languages, queries and transformations [2]. Most popular MDE tools were not designed to handle models with millions or even tens of thousands of elements. Furthermore, their execution engines typically perform unnecessary computations and do not exploit capabilities of modern hardware. Improving the efficiency of existing model management programs would help to address their poor performance over large models.

This paper aims to improve the execution time of model validation constraints by devising a novel parallel execution approach which can scale not only with the constraints, but also with the model elements. Specifically, we apply a novel rule and data-parallel approach to the task of model validation. We implement our solution by modifying the execution engine of the Epsilon Validation Language – a hybrid (i.e. declarative and imperative) model validation language. In doing so, we uncover a multitude of practical challenges with concurrency and provide solutions to such challenges. We hope that by choosing a complex model validation language and supporting all of its features without any additional constructs or change in syntax or semantics, other model validation solutions with equal or more limited expressive power can also be adapted in a similar manner. Our implementation is open-source, available from Epsilon Labs repository [3].

The remainder of this paper is as follows. Section 2 reviews the most relevant work related to our research. Section 3 introduces the Epsilon Validation

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 115–131, 2018.
https://doi.org/10.1007/978-3-319-92997-2_8

Language. Section 4 discusses challenges encountered with parallel model validation. Section 5 presents an overview of our parallel implementation. Section 6 evaluates the correctness and performance of our implementation, including a comparison to interpreted and compiled OCL. Section 7 concludes the paper and outlines further development opportunities.

2 Related Work

Our work is a direct continuation of a previous effort to parallelise the Epsilon Validation Language in a Masters' project [4]. The implementation and choice of metamodel for performance evaluation was inspired by this work. Although some promising speedups were achieved and the main challenges with parallelisation were identified, there remained some outstanding issues with the implementation and evaluation, which we hope to have addressed after substantial refactoring.

2.1 Background

Generally there are three approaches to improving model management program performance: *incrementality*, *laziness* and *parallelism*. The MDE community has put substantial effort into studying incrementality as this can lead to significant performance improvements. This arises by caching results for model elements which have already been evaluated so that after a small change, the program is only executed over the changed elements. Furthermore, a *reactive* execution engine can be obtained by combining incrementality with laziness, so that the program is not only re-executed on a subset of the model, but also only when the results of the program are actually used. Most research in this area focuses on optimising model transformation engines such as ATL [5], as the ATL language and its engine's architecture make such semantics more straightforward to implement. Consequently, there are incremental [6], lazy, [7], parallel [8], distributed [9] and reactive [10] extensions of ATL.

Although incrementality and laziness are valuable optimisations, they do not improve execution times when computations are mandatory, for example when a program is executed on a model for the first time, or a large proportion of the model is changed, and when the program results are always used. By contrast, parallelism exploits modern hardware to perform computations at a higher rate, rather than reducing the overall number of computations.

2.2 Model Validation Optimisations

The most well-known and commonly used language for model querying and validation is the Object Constraint Language (OCL) [11], which is a functional language free from side-effects and imperative features. Most optimisations of model validation algorithms are built on OCL.

Cabot and Teniente [12] designed an incremental model validation algorithm which ensures the smallest/least work expression can be provided to validate a

given constraint in response to a CRUD (Create/Read/Update/Delete) event/change in the model. It automatically generates the most efficient expression for incremental validation for a given event. Tisi et al. [13] proposed an iterator-based lazy production and consumption of collection elements. Iteration operations return a reference to the collection and iterator body, which produces elements when required by the parent expression. Laziness in this context is useful when a small part of a large collection is required, as the iteration overhead can actually be worse than eager evaluation in some cases.

Vajk et al. [14] devised a parallelisation approach for OCL based on Communicating Sequential Processes (CSP). The authors' solution exploits OCL's lack of side effects by executing each expression in parallel and then combining the results in binary operations and aggregate operations on collections. They demonstrate equivalent behaviour between the parallel and sequential OCL CSP representations analytically. Their implementations use CSP as an intermediate representation which is then transformed into C# code. Users must manually specify which expressions should be parallelised. The authors' evaluation was brief, with relatively small models and simple test cases. Despite the absence of any non-parallel code in their benchmark scenarios, their implementation was 1.75 and 2.8 times faster with 2 and 4 cores respectively.

3 Epsilon Validation Language

Epsilon [15] is an Eclipse project that provides languages for model management tasks such as validation, transformation, comparison and pattern matching. All task-specific languages build upon a feature-rich model-oriented language – the Epsilon Object Language (EOL) [16] – a dynamically typed, interpreted language which supports imperative programming. EOL also supports native types, effectively allowing for execution of arbitrary Java code. A key feature of Epsilon is that it works across a range of modelling technologies by abstracting model operations through a connectivity layer, so a script written to work on an EMF model can also be used on an XML document or a spreadsheet without changes.

The Epsilon Validation Language (EVL) [17] extends EOL to enable users to express their validation constraints in a more structured and declarative manner. Since EOL supports all features of OCL with similar syntax and built-in operations, EVL is can be used in the same manner as OCL. Users define constraints within the context of a model element type, where each constraint has a check expression (or statement block for more complex constraints) returning a Boolean. In addition, constraints may also have a *guard*, which is semantically identical to prepending the constraint check expression with a Boolean expression followed by the *implies* operator. Guard blocks can also be declared in a context. Constraints may have dependencies on other constraints using the *satisfies* operation, which returns the result of calling the specified constraint(s) for the current element. Constraints declared as *lazy* will only be executed when invoked by a *satisfies* operation. A context declared as lazy is equivalent to having all of its constraints being lazy. EVL also allows users to defined fixes for constraints, which can modify the model with arbitrary imperative code. Like

all Epsilon rule-based languages, EVL has *pre* and *post* blocks, which allow for arbitrary code to be run before and after the main program, respectively.

3.1 Example Program

Suppose that we have a model of a Java program, and we want to ensure that every class overrides the *equals* method according to the contract[1]. Listing 1 shows how this could be implemented in EVL. Note that by declaring the *hasHashCode* constraint as lazy, we only check it once per *ClassDeclaration* as a pre-condition for the *hasEquals*. If *hasHashCode* fails for the current element under consideration (referred to as *self*), then we avoid executing the *hasEquals* check expression for the current class. This means a class may fail to satisfy either *hasHashCode* or *hasEquals*, but not both. Therefore our results will never contain the same class more than once. Also note that by declaring the *getMethods* operation as cached, we avoid re-evaluating it for a given model element, so that if a class satisfies *hasHashCode*, we do not need to find all of its methods again in line 9.

Listing 1. EVL program over Java metamodel

```
1  @cached
2  operation AbstractTypeDeclaration getMethods() : Collection {
3    return self.bodyDeclarations.select(bd|bd.isKindOf(MethodDeclaration));
4  }
5
6  context ClassDeclaration {
7    constraint hasEquals {
8      guard : self.satisfies("hasHashCode")
9      check : self.getMethods().exists(method |
10       method.name == "equals" and
11       method.parameters.size() == 1 and
12       method.parameters.first().type.type.name == "Object" and
13       method.modifier.isDefined() and
14       method.modifier.visibility == VisibilityKind#public and
15       method.returnType.type.isTypeOf(PrimitiveTypeBoolean))
16   }
17   @lazy
18   constraint hasHashCode {
19     check : self.getMethods().exists(method |
20       method.name == "hashCode" and
21       method.parameters.isEmpty() and
22       method.modifier.isDefined() and
23       method.modifier.visibility == VisibilityKind#public and
24       method.returnType.type.isTypeOf(PrimitiveTypeInt))
25   }
26 }
```

[1] Equal objects must have the same hash code, but unequal objects do not necessarily have different hash codes.

3.2 Execution Semantics

The execution algorithm for sequential EVL is given in Listing 2.

Listing 2. Simplified sequential EVL algorithm

```
1  preBlock.execute();
2  for (Context context : contexts) {
3    for (Object element : context.getAllOfKind()) {
4      if (!context.isLazy() && context.guard(element)) {
5        for (Constraint constraint : context.getConstraints()) {
6          if (!constraint.isLazy() && constraint.guard(element)) {
7            if (!constraint.check(element)) {
8              unsatisfiedConstraints.add(constraint, element);
9            }
10         }
11       }
12     }
13   }
14 }
15 postBlock.execute();
```

For each context (line 2), we loop through all elements of that type and subtypes (line 3). Provided that the guard blocks of each context and constraint are satisfied and they are not marked as lazily evaluated (lines 4 and 6 respectively), we simply execute the check block (line 7) of each constraint within the declared context (line 5) for the current element. We add each failure to the set of unsatisfied constraints (line 8). Not shown in Listing 2 is the constraint trace, which keeps track of results to avoid re-evaluating constraint and element pairs in case of a satisfies operation (i.e. dependencies between constraints). The semantics of how this is used will be discussed in the next section. Also note that the *pre* and *post* blocks (lines 1 and 15, respectively) are not of interest as they may contain arbitrary imperative code. We have also excluded fixes for simplicity.

4 Challenges with Parallelisation

Our observation from Listing 2 is that each iteration of the three loops need not be performed sequentially, since there is no dependency between them (except for occasional constraint dependencies, discussed below). Fixes in EVL are performed optionally after validation and initiated by the user, so the model is only queried, never written to[2]. In theory, this makes the task of executing read-only operations (check blocks) within a loop inherently parallelisable. However in practice, this is complicated by a number of factors, to which we now turn.

[2] Parallel execution of fixes is beyond the scope of this paper.

4.1 Accessing Data Structures

A key challenge with retro-fitting concurrency into an existing program is handling of access to data structures. When multiple threads have shared access to the same mutable data, the non-deterministic nature of parallel execution can lead to inconsistent states (Readers-Writers problem). There are generally three solutions to this problem: Not sharing such data between threads, making the data immutable; or using synchronization whenever accessing the data [18]. Unfortunately in most cases, the easiest option of the three (synchronization) is adopted. This has a major impact on performance not only because a single thread can execute synchronized regions at a time, but also the overhead introduced by synchronization mechanisms. This is especially problematic for data structures which are subject to frequent writes.

Even though model validation is in principle a read-only task, intermediate data structures such as the set of unsatisfied constraints need to be written to concurrently. Furthermore, caches (such as those used to store model elements) can present problems if they are written to during execution. In Epsilon, caching of model elements is performed lazily, i.e. when all elements of a specified type are requested for the first time.

4.2 Control Flow Traceability

It is important to be able to report on errors encountered during execution. EVL scripts are interpreted, so errors such as accessing an invalid model property are reported at runtime. Epsilon therefore records the execution stack trace so that in the event of an error, the location of the fault can be identified and reported to the user. When executing concurrently however, each thread could be executing different parts of the script or the same parts with different data. When an exception occurs, a co-ordination mechanism is needed to stop all threads from executing, and for the cause of the exception to be correctly reported. Furthermore, since exceptions are usually propagated to the program's top level, the reporting needs to be able to capture the stack trace of the thread which encountered the issue and make it available to the main thread, as parallel execution should be terminated at this point.

4.3 Handling Properties and Variables Scope

EVL is a structured extension of EOL, which supports almost every feature of a general-purpose scripting language. Amongst these are user-defined operations which may be defined in the context of types such as model elements or even built-in types. More fundamentally however is the ability to define variables in different scopes. Epsilon therefore has an internal frame stack which is used heavily throughout the code base. With multiple threads executing concurrently, the scoping of variables needs to be respected in an equivalent manner to sequential execution. So, for example, whenever a variable is declared in an executable block, once that block has finished execution, the variable should be discarded

and inaccessible from all threads. Similarly, if a variable is declared globally in the *pre* section, it should be visible at all times to all threads.

Furthermore, EOL also allows individual objects (e.g. model elements) to have extended properties associated with them. These properties should be accessible from multiple threads.

4.4 Lazy Constraints and Dependencies

A classic impediment of parallelism is dependencies. In EVL, this can occur through *satisfies* operation calls. This is typically used in the guard block of a constraint to prevent it from executing if another constraint (or set of constraints) is not satisfied for the same model element. With multiple threads of execution, the target(s) of a *satisfies* operation may be executing concurrently with the caller. This means that there may be a duplication of effort, with the same constraint being executed at least twice, or the caller may need to wait for the result. In the latter case, not only does performance become single-threaded but there is a co-ordination overhead of notifying the caller when the result is made available. Further complicating matters are *lazy constraints*, which are only executed when invoked by a *satisfies* operation.

4.5 Testing for Correctness

Finally, we would like to emphasize the non-deterministic nature of concurrent programs. With single-threaded execution, the behaviour of the program is predictable, so a test suite which passes once will always pass for the same program with identical inputs. However with multiple threads, those same tests may become "flaky"; failing only on some occasions (depending on thread scheduling). In the best case, inconsistent output would result in a failure on at least one occasion, thus exposing a potential issue. Much more dangerous is correct behaviour under test conditions but spurious runtime exceptions resulting from a malformed internal state. Furthermore, debugging concurrent programs is also difficult, since the same tools and techniques used to detect issues with sequential programs may be inadequate or misleading when used for concurrent programs.

5 Parallel Solution

In this section we give a high-level overview of our solution to the problems identified in the previous section. Firstly we begin with an outline of the parallel execution approach.

5.1 Architecture

Our parallel solution abstracts the execution process using an extension of the *ExecutorService* [19] interface. This allows us to, in theory, substitute any parallel execution infrastructure without depending explicitly on threads. Our implementation uses a custom *ThreadPoolExecutor* [20] with a fixed pool of threads.

Parallel execution begins when the EVL script has been parsed and the models under validation are fully loaded into memory, and ends once all constraints have been checked.

5.2 Parallelisation Strategies

To achieve maximum parallelism, it is important to choose the appropriate level of granularity. For instance, parallelising only constraints themselves would have no performance benefits if there is just a single constraint in the script. A similar argument can be made for contexts. Since the issue of scalability is rooted in the size of models, parallelism should ideally be performed at the element level. Parallelisation is performed by wrapping the desired jobs into a function and passing it to the *ExecutorService*. We have experimented with the following parallel implementation strategies:

Element-Based. In this strategy, we parallelise the second for loop (line 4 onwards) in Listing 2. Each context and constraint is executed in a single thread, but a separate job is created for each element. This is ideal if the model is large and the number of constraints and contexts is small.

Stage-Based. This strategy is unlike the previous two in that it splits the execution into three distinct phases, where the input to each phase is the output from the previous phase. In the first phase, we loop through all elements in all contexts (as in lines 2 and 3), and submit a job for each context and element pair. The job simply checks whether the context should be executed (as in line 4) and if so, it adds the context and element pair to a thread-local batch data structure. Once this is complete, the results from all threads are merged and passed on to the next phase. In the second phase, we loop through the results from the first phase and all of the constraints for each context in the results (as in line 6) and check whether the constraint should be executed (as in line 6) and if so, it adds the element and constraint pair to another thread-local batch data structure. As before, once the jobs have completed the thread-local results are merged and passed to the final phase. In the third phase, we submit a job for each constraint and element pair (as in lines 7–9) and await the results.

This strategy clearly separates the three stages of the algorithm, with the main advantage being that we can achieve maximum parallelism at all of the for loops. Furthermore, it may be helpful for garbage collection since the data is more clearly scoped as a result of the staged filtering process. On other hand, the overhead introduced by additional intermediate results structures could be detrimental to performance and/or memory consumption.

Constraint-Based. This strategy differs from the element-based solution in that the parallelisation is performed at the third for loop (line 6 onwards). This means we create a job for each constraint and model element pair (i.e. it is both

data and rule parallel), but the context guards are executed by the main thread. This is ideal if there are many constraints and there are no guarded contexts.

5.3 Thread-Safe Data Structures

It is useful to classify access to internal data structures during execution of the script into one of three categories: read only, write only, and read and write. The first category is inherently thread-safe since it is immutable. Such structures include the script itself, the model, the constraints and constraint contexts. The second category will never be queried during execution. An effective solution for this is to create a per-thread data structure and then merge all of the thread-local data structures once execution has completed. Since no thread will ever attempt to read from the structure, we do not need to merge or synchronize access during execution. The set of unsatisfied constraints (i.e. the results data structure) belongs in this category. The third category is unsurprisingly the most complex to deal with. Structures which fall into this category include the operation contributor registry (a cache for storing operations available on a given object), constraint trace (a cache of executed constraint-element pairs and their results), execution controller (which keeps track of the stack trace and allows for debugging of statements and expressions) and the frame stack.

Our solution is to use a thread-local structure (serial thread confinement) with base delegation. We will use the frame stack as an example to illustrate this. The idea is that each thread has its own frame stack (which is only accessible from that thread) so that whenever the *getFrameStack()* method is called, we return the frame stack associated with the calling thread. Each thread-local frame stack also has a reference to the main thread's frame stack. Whenever a variable lookup is performed, we first check the thread-local stack and if it is not present, we then look in the base. Once parallel execution has finished (i.e. all constraint and element pairs have been checked), we merge the thread-local results back into the base frame stack (i.e. that of the main thread). In the constraint-based strategy, the main thread also needs to write to the frame stack during execution, so we make the base structure thread-safe by using an appropriate collection. In all cases, this is either a *ConcurrentLinkedDeque* [21] (e.g. for frame stack); a lock-free double-ended queue structure where writes are based on atomic compare-and-swap operations or *ConcurrentHashMap* [22] (where there is no synchronization for reads and locking for writes is of high granularity). We found that this approach eliminated many concurrency issues and is sufficient for supporting EVL's imperative features without introducing a major performance bottleneck due to excessive synchronization.

5.4 Exception Handling

By using a thread-local execution controller, each thread is able to keep track of its own execution trace so that when an error occurs, the cause can be identified in a similar manner to sequential execution. However we found that propagation

and signalling that an exception has occurred to be more involved. Our solution is to use an *ExecutionStatus* object which encapsulates the state of success and/or failure within our *ExecutorService*. The idea is that when all jobs have been submitted to the executor, we start a termination thread which blocks until the *ExecutorService* has finished executing all jobs. Meanwhile, the main thread locks onto the *ExecutionStatus*, waiting for a signal. So at this point, both the main and termination threads are idle. If the *ExecutorService* completes all jobs successfully, the termination thread signals a condition which notifies the main thread, at which point the termination thread ends and normal execution is resumed. If an exception occurs, the main thread is also signalled and the exception can be propagated as usual. The exception signalling occurs by calling the *setException* method in our *ExecutionStatus* object. Before this method gets called, we first capture the exception message, since the thread-local stack trace will disappear once parallel execution ends.

5.5 Dependencies

The original EVL algorithm added every constraint and element pair it checked to the constraint trace. This was wasteful since in most cases there are no dependencies between constraints. This is also the only structure for which we use a synchronized collection rather than thread-local base delegation, which introduces considerable overhead after checking each constraint and element pair. We changed this behaviour (in both sequential and parallel EVL) to avoid unnecessary writes to the constraint trace whilst also limiting the number of times each constraint-element combination is checked to at most 2 times. This is achieved by keeping track of the set of constraints depended on. When a *satisfies* operation is invoked, we first check whether the constraint is in this set. If so, we then proceed to check the trace for the specific constraint and element. If the result is not present, we perform the check and add it to the trace. If the constraint was not in the set of constraints depended on, we add it and also add the result to the trace. In practice, we optimise the checking of the constraints depended on every time a constraint is executed using a flag which indicates whether the constraint is a dependency. This flag is set to true on the constraint when it is first invoked by a *satisfies* operation. If this flag is true, we know to check the trace for a result, otherwise we proceed as usual.

As an example, suppose that constraint A depends on constraint B. If A runs before B, then B is checked during A. However B is then added to the trace and constraints depended on, so when B runs, it will not be re-checked. If B runs before A, then unfortunately B is checked again when A runs, but won't be checked afterwards because it will be in the trace. Future invocations (i.e. with different elements) will then know to check the trace first because constraint B will be in the set of constraints depended on.

6 Evaluation

As our solution is built on an already established model validation engine and does not change the syntax, semantics or supported features of the existing platform, our evaluation criteria will focus exclusively on *correctness* and *performance*. We consider correctness to be a hard requirement, since concurrent programs are notoriously difficult to reason about due to the non-deterministic nature of execution.

6.1 Test Models and Scripts

Our main test script runs over models conforming to the Java 5 language metamodel provided by MoDisco [23]; a model-driven reverse engineering project designed to migrate legacy code artefacts into models. We chose this metamodel because it is substantially complex (749 elements) yet relatively easy to comprehend given the familiarity of the domain to Java programmers. Moreover, we are able to obtain models from real code artefacts automatically using MoDisco's Java Discovery feature as opposed to synthetically generating large models. Of course, since there are plenty of open-source Java projects, it is easy to obtain models of various sizes. For convenience and reproducibility, we used the models from [24], which vary from approximately 100,000 to over 4.35 million elements.

For our validation constraints, we took inspiration from the Findbugs[3] project, which lists a large number of "code smells" in Java code. Some of these require sophisticated static analysis, so in order to minimise errors with our validation constraints we implemented a subset of the simpler bug locators. Our EVL program consists of 31 constraints across 16 contexts (model element types), written in a declarative style. To ensure that our parallel implementations scale as intended, we also created three other scripts. One of these contains a single constraint for each of the 16 contexts, another contains 9 constraints within a single context, and one which consists of a single constraint within a single context. The rationale is to test throughput and identify any potential weaknesses in the scalability of our solutions.

6.2 Correctness

It would be challenging to formally prove correctness of our parallel solution using static analysis techniques due to the size and complexity of the codebase. Instead, we opt for a thorough series of automated dynamic JUnit tests. In this section, we give a brief overview of our testing methodology.

Epsilon already has a large suite of unit tests, especially for EOL, which we build upon. Our test suite for EVL ensures that all language features are exercised thoroughly. This is achieved by having a test script, a minimal plain XML model and by assertion of an expected number of unsatisfied constraints for each context and constraint in the script. The features tested include pre and

[3] http://findbugs.sourceforge.net/bugDescriptions.html.

post blocks, lazy constraints, constraint dependencies, contextless constraints, constraint pre-conditions, imperative code, user-defined, cached and imported operations, constraint messages, fixes as well as ensuring correct scoping of variables. We also have another script which accesses a non-existent property of the model to ensure correct propagation and reporting of exceptions in the user's code.

Our second test suite consists of equivalence tests with the sequential implementation. The infrastructure for this is rather complex, since we need to ensure that we are comparing the same model and script combinations whilst varying the modules (execution strategies); which themselves have different configurations as well. We automate this check by calculating an ID for each model, metamodel and script combination. We refer to a combination of model, metamodel, script and module as a *scenario*. Our first test is whether the scenario can actually execute without exceptions. Once this is established, we then check that for the oracle scenario (which uses the original sequential implementation), the results and internal data structures are "equal". These include the unsatisfied constraints, frame stacks, the constraint trace, constraints depended on, operation contributors and stack trace manager. We use the scripts described in the previous subsection as well as other complex scripts and models which were developed independently from the project. For this suite alone, we have 15 models, 5 metamodels and 9 scripts.

Since some changes were made to the Epsilon core code base, we also need to ensure that the original sequential EVL engine produces correct results when executed over a real model and script, as opposed to ones solely designed for testing the engine internals. To do this, we took our *java_findbugs* script and rewrote it in OCL. Since we execute the same model with EVL and used the same constraint names, we can could compare the references (i.e. memory address) of the *EObjects* (model elements) in our test suite to ensure that the set of results from EVL and OCL are identical.

All three of the above test suites are parameterised with an EVL engine implementation, so we could repeat the tests for all our solutions. Since we have three parallelised implementations, each of which accepts a number of threads to use as a parameter, we opted to use 1, 2, the number of logical cores on the system and many (8191) threads for each implementation. To detect concurrency issues, we also parametrised each suite with a number of cycles, which would repeat the tests a specified number of times. Given the very large number of combinations of modules and parameters, scripts and models as well as the tests themselves, each run results in thousands of unique tests.

After running the tests and real experiments on both small and large models tens of thousands of times on a HPC cluster, we can be confident in the correctness of our implementation from a practical sense as we did not encounter any exceptions, crashes or incorrect results.

6.3 Performance

Our platform for performance evaluation has the following characteristics: Windows 10 Enterprise (v1607), Intel Core i7-4790K @ 4.00 GHz (4 cores/8 threads) CPU, 16 GB DDR3-1600 MHz RAM, Samsung 850 EVO 500 GB SSD, Java HotSpot 64-bit Server VM (build 9.0.4+11). We disabled Turbo Boost for a fairer comparison when varying the number of threads. The arguments to the JVM were as follows[4]:

```
-XX:InitialRAMFraction=16 -XX:MaxRAMFraction=1
-XX:+AggressiveOpts -XX:+UseParallelOldGC
```

We measure the parsing and loading time of a model independently from the execution time of a module, using *System.nanoTime()*, and report our results in milliseconds. We calculate memory consumption (in MB) by summing the peak usage of all memory pools after measuring execution time. We run each experiment three times and use a script to automatically calculate the mean, speedup and efficiency of our results. We should also note that each experiment is run in a separate JVM invocation to minimise interference and warm-up effects.

6.4 Analysis of Results

Table 1 shows our main result, which we split into five sections (separated by alternated shading) for convenient analysis. All speedups are relative to the sequential EVL implementation unless otherwise indicated. The number of threads are in parenthesis where applicable.

Our parallel implementations perform similarly for the *findbugs* script, with around 3× speedup using four threads. This decreases slightly with model size, but for models with hundreds of thousands of elements, the performance improvements are still significant. The imperfect efficiency of our parallel solution can be mostly explained by the overhead of creating and submitting jobs to the executor, as well as merging thread-local results. As demonstrated by our second group of results, when running a parallel version with a single thread as the baseline we see an almost perfect speedup using two threads. The original sequential implementation is over 10% faster than the single-threaded stage-based implementation, which can be interpreted as an estimate of the parallelisation overhead. The third group of our results shows broadly similar results across the parallel implementations, with single-threaded efficiency of 89% relative to the sequential implementation for three million elements. The fourth group of results act as a test of throughput, since the script contains only a single constraint. Here we see that despite the large model size, the absolute execution time is too small for parallelism to provide significant benefits, with efficiency dropping to just 33%. The last group of results shows that for multiple constraints within a single context, the element-based implementation is clearly superior. With four threads, we observe 100%, 90% and 84% efficiency for the

[4] We use the ParallelOld garbage collector since we're interested in throughput.

Table 1. Selected benchmark results

MODULE	SCRIPT	ELEMENTS	TIME	SPEEDUP	MEMORY
Sequential	findbugs_all	4M	9 227 448	—	2 758
Stage–Based (4)	findbugs_all	4M	3 009 693	3.066	4 342
Compiled OCL	findbugs_all	4M	22 024	418.972	475
Sequential	findbugs_all	1M	948 871	—	3366
Stage–Based (4)	findbugs_all	1M	330 027	2.875	4 967
Compiled OCL	findbugs_all	1M	7 078	134.059	559
Sequential	findbugs_all	200K	47 181	—	5 016
Stage–Based (4)	findbugs_all	200K	16 636	2.836	5 043
Compiled OCL	findbugs_all	200K	2 593	18.196	12
Sequential	findbugs_all	2M	3 815 590	1.119	3 782
Stage–Based (1)	findbugs_all	2M	4 265 051	—	4 822
Stage–Based (2)	findbugs_all	2M	2 162 179	1.973	5 050
Stage–Based (4)	findbugs_all	2M	1 251 400	3.408	4 981
Stage–Based (8)	findbugs_all	2M	874 808	4.875	5 031
Sequential	findbugs_all	3M	5 892 807	—	4139
Stage–Based (1)	findbugs_all	3M	6 626 802	0.889	4 428
Elements–Based (1)	findbugs_all	3M	6 693 093	0.88	4 196
Constraints–Based (1)	findbugs_all	3M	6 609 898	0.892	4 443
Interpreted OCL	findbugs_all	3M	5 845 796	1.008	3 442
Compiled OCL	findbugs_all	3M	18 757	314.166	478
Sequential	findbugs_1Constraint	4.3577M	11 024	—	1 269
Stage–Based (4)	findbugs_1Constraint	4.3577M	8 248	1.337	2 149
Elements–Based (4)	findbugs_1Constraint	4.3577M	8 180	1.348	2 320
Interpreted OCL	findbugs_1Constraint	4.3577M	12 745	0.865	584
Sequential	findbugs_1Context	2.5M	30 193	—	1 236
Elements–Based (8)	findbugs_1Context	2.5M	6 038	5.000	3 490
Constraints–Based (8)	findbugs_1Context	2.5M	7 139	4.229	3 700
Stage–Based (8)	findbugs_1Context	2.5M	7 126	4.237	3 693
Elements–Based (4)	findbugs_1Context	2.5M	7 527	4.011	3 274
Constraints–Based (4)	findbugs_1Context	2.5M	8 379	3.603	3 368
Stage–Based (4)	findbugs_1Context	2.5M	8 997	3.356	3 582
Interpreted OCL	findbugs_1Context	2.5M	25 701	1.175	697

element, stage and constraint-based implementations respectively. However it is the Hyper-threaded performance which shows marked differences. Although the constraint-based implementation is faster than the stage-based one with four threads, this difference disappears once we add the remaining logical processors. With eight threads, they both achieve 4.23× speedup, whereas the element-based implementation is five times faster.

We observe similar performance between interpreted OCL and sequential EVL for our main *findbugs* script, though there is a 15% difference for both the single constraint and single context variants, with OCL being faster in the former case and slower in the latter. However we should note that the implementations of EVL and Eclipse OCL are fundamentally different. Compiled OCL is in a league

of its own in terms of performance, though its advantage greatly diminishes with smaller models. The benefits of using EVL over compiled OCL include a richer feature set and the ability to work with any modelling technology, amongst others [17]. Furthermore, the performance of parallel EVL will improve over time as the number of cores in developer workstations increases. Currently, compiled Eclipse OCL requires the user to embed their constraints in the metamodel and manually regenerate the code for any changes to the constraints or metamodel.

Overall, these results indicate what one would expect from a parallel algorithm in that with smaller problems, the overhead of co-ordination outweighs the gains but with a bigger problem, the parallel version is able to "catch up" and overtake the sequential algorithm's performance [26]. Typical speedups for parallel model management programs in the literature with four threads range between 2.5×–3× (in the case of LinTra [25], 1.19× with four threads and 3.24× with sixteen). However these are in the context of model-to-model transformation; a task which is arguably more complex.

7 Conclusions and Future Work

In this paper, we have presented a novel parallel model validation solution which is scalable with both the number of constraints and number of model elements. Along the way, we have identified and provided solutions to concurrency issues arising from uncommon features in model validation such as constraint dependencies and imperative programming constructs. We have also tested our solution by not only exercising all features of the language, but also through equivalence testing with the non-concurrent version as well as with OCL; a popular model querying and validation language with a well-defined specification. In terms of performance, we observed roughly linear improvements in execution times as we increased the number of threads, though naturally our parallel solution does impose an overhead which reduces the efficiency in cases where the absolute execution times are relatively small. However we realise that smaller models and scripts benefit significantly less, if at all, from parallelisation.

To further improve performance, we intend to combine our parallel solution with an incremental one [27], which will undoubtedly present new challenges. To improve scalability, we are also considering a distributed solution; though the lack of shared memory and communication costs in distributed systems would require some modifications to our proposed parallel solution. If our solution is modified to accommodate partial and lazy loading of models from non-volatile memory, we could also avoid the upfront cost (both in time and memory) of parsing the model and possibly even make the process multi-threaded. Another possibility is to parallelise first-order logic operations on collections, since these are commonly used and are usually pure functions (i.e. free from side effects).

Going beyond model validation, we plan to refine our solution by applying it to other rule-based model management tasks in Epsilon, such as pattern matching (EPL) [28], model comparison (ECL) [29] and model-to-text transformation (EGL) [30]. In principle, our systematic analysis and devised approach should also be applicable to other model management languages outside of Epsilon.

References

1. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Scalability: the holy grail of model driven engineering. In: Proceedings of the First International Workshop on Challenges in Model Driven Software Engineering, Toulouse, pp. 10–14 (2008)
2. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ràth, I., Varrò, D., Tisi, M., Cabot, J.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest (2013). Article No. 2
3. Parallel EVL implementation. https://github.com/epsilonlabs/parallel-erl
4. Smith, M.: Parallel model validation. Masters' thesis, University of York (2015)
5. Joualt, F., Allilaire, F., Bèzivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
6. Jouault, F., Tisi, M.: Towards incremental execution of ATL transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 123–137. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13688-7_9
7. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 32–46. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_4
8. Tisi, M., Martínez, S., Choura, H.: Parallel execution of ATL transformation rules. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 656–672. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_40
9. Benelallam, A., Gómez, A., Tisi, M., Cabot, J.: Distributed model-to-model transformation with ATL on MapReduce. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 37–48 (2015)
10. Martínez, S., Tisi, M., Douence, R.: Reactive model transformation with ATL. Sci. Comput. Program. **136**(C), 1–16 (2017)
11. Object Constraint Language (OCL) specification. http://www.omg.org/spec/OCL/About-OCL/
12. Cabot, J., Teniente, E.: Incremental evaluation of OCL constraints. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 81–95. Springer, Heidelberg (2006). https://doi.org/10.1007/11767138_7
13. Tisi, M., Douence, R., Wagelaar, D.: Lazy evaluation for OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling Co-located with 18th International Conference on Model Driven Engineering Languages and Systems, Ottawa, pp. 46–61 (2015)
14. Vajk, T., Dávid, Z., Asztalos, M., Mezei, G., Levendovszky, T.: Runtime model validation with parallel object constraint language. In: Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation, Wellington (2011). Article No. 7
15. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, Potsdam, pp. 162–171 (2009)
16. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon object language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_11

17. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of OCL for capturing structural constraints in modelling languages. In: Abrial, J.-R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 204–218. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-11447-2_13
18. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley, Boston (2005)
19. java.util.concurrent.ExecutorService. https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ExecutorService.html
20. java.util.concurrent.ThreadPoolExecutor. https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ThreadPoolExecutor.html
21. java.util.concurrent.ConcurrentLinkedDeque. https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html
22. java.util.concurrent.ConcurrentHashMap. https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ConcurrentHashMap.html
23. Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014)
24. Eclipse platform EMF models. http://atenea.lcc.uma.es/index.php/Main_Page/Resources/LinTra#Input_Models_3
25. Burgeño, L., Troya, J., Wimmer, M., Vallecillo, A.: Parallel in-place model transformations with LinTra. In: Proceedings of the 3rd Workshop on Scalable Model Driven Engineering, L'Aquila, pp. 52–62 (2015)
26. Goetz, B.: From Concurrent to Parallel. QCon 2017, London. https://www.infoq.com/presentations/tecniques-parallelism-java
27. Incremental EVL. https://github.com/epsilonlabs/incremental-evl
28. Kolovos, D.S., Paige, R.F.: The epsilon pattern language. In: Proceedings of the 9th International Workshop on Modelling in Software Engineering, Buenos Aires, pp. 54–60 (2017)
29. Kolovos, D.S.: Establishing correspondences between models with the epsilon comparison language. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 146–157. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02674-4_11
30. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The epsilon generation language. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 1–16. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69100-6_1

SysML Models Verification and Validation in an Industrial Context: Challenges and Experimentation

Ronan Baduel[1,2(✉)], Mohammad Chami[1], Jean-Michel Bruel[1], and Iulian Ober[1]

[1] IRIT, University of Toulouse, Toulouse, France
ronan.baduel@irit.fr
[2] Bombardier Transportation, Crespin, France

Abstract. This paper presents a solution for SysML model verification and validation, with a return of experience from its implementation in an industrial context. We present this solution as a way to overcome issues regarding the use of SysML in an industrial context. We contribute by providing a method and a list of the existing challenges and experimentation results. We advocate the need to have semantics for SysML models without having to define a full domain-specific modeling language. We highlight the work, requirements and benefits that arise from the application of existing technical solutions, and hint at new perspectives and future development in system verification and validation.

Keywords: MBSE · Model verification · Systems modeling · SysML

1 Introduction

This paper focuses on the *verification* and *validation* (V&V) of system models, built as part of the system development process at Bombardier Transportation (BT) for producing a broad portfolio of railway products. The Systems Modeling Language (SysML) [1] is used to develop the system models based on a BT customized System Modeling Method (SysMM) [2]. The main objective is to develop a generic V&V solution based on SysML without any tool dependent criteria so that it is reusable across all BT divisions and projects.

1.1 MBSE and V&V

Systems engineering is an interdisciplinary process for supporting the system life cycle. Model-Based Systems Engineering (MBSE) introduces new capability into systems engineering practice and is defined by INCOSE as *"the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases"* [3].

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 132–146, 2018.
https://doi.org/10.1007/978-3-319-92997-2_9

As there are several definitions of the terms *verification* and *validation* (V&V), we refer in this paper to the definitions of the standard IEEE 1012-2012 [4], which we apply in the context of model development, and not on the scale of the whole development process. From this point of view, *verification* ensures that the models created during the early steps of the development process have been correctly built, meaning they are free of errors and represent a coherent system. As for *validation*, it ensures that the system represented by the models match the requirements traced to the information displayed or induced.

1.2 Motivating Example

From a technical point of view, the main aspect in verifying a model is ensuring that no errors were made in the specification of the system design. Creating models and having correct models are two different things, and can impact the rest of the development process. Similarly to validation, the earlier an error is detected, the less the cost [5].

From an organizational point of view, within large organizations, ensuring that everyone create models under the same guidelines and constraints is a challenging task. It is crucial that the modeling team members work the same way and are able to exchange and communicate around their delivered models with others without any misunderstanding or consistency issues. This gets more complex with teams spread across continents and/or companies. Having one defined modeling method across an organization and applying it the same way are two different things.

During the early phases of MBSE adoption at BT, the focus on models' V&V was triggered mainly by specific projects based on particular customers or countries needs. As MBSE enabled the reuse of models specification across projects, the goals of V&V was extended towards being more generic and project-independent. This however introduced the discovery of hundreds of errors by the BT V&V team even sometimes for a single verification or validation rule. It was not that the models were globally false, but rather that the project specific teams had their own interpretation of the method or specific modeling practice, gained from experience. What it did mean is that the models could neither be easily reused by other teams, nor could they be adapted while reproducing the same modeling approach. This is a main challenge for large organizations that are driven by project specific customers in contrast with those able to generalize their products and offer a predefined product portfolio (e.g., in automotive). Therefore, the need for reusing V&V of the delivered models is crucial to ensure proper systems models reuse. Moreover, it is crucial to implement the suitable adoption approach, similar to the D3 MBSE Adoption Toolbox [6].

1.3 Outline of the Paper

Section 2 presents BT SysMM, how it was implemented, and what were the specifications of the verification solution to be developed. Section 3 discusses the state of the art introducing the solutions on which we based our work on.

Section 4 presents our solution and the work realized, providing a return on experience. Section 5 shows an example of application, and Sect. 6 describes the challenges known beforehand as well as those encountered when developing and implementing our solution. Finally, Sect. 7 concludes on what has been done and gives future directions for this work.

2 Background on BT SysMM

MBSE has been deployed at BT for several years across various applications, for instance in requirements, functional and safety engineering [2]. The BT System Modeling Method (SysMM) [2] consists of three main tasks. Each one aims at analyzing the system of interest (SOI) on a specific abstraction level (see Fig. 1). SysMM describes how BT engineers analyze, define, and represent their SOI using system models. The purpose of SysMM is to manage complexity and increase quality of the design artifacts to reduce development costs.

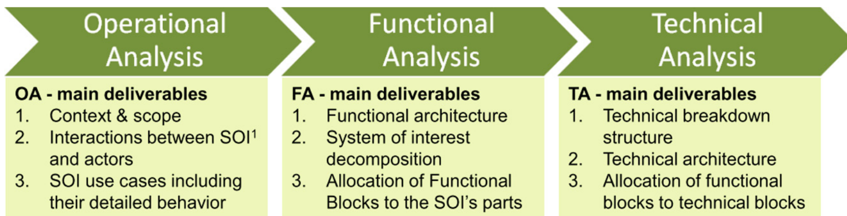


Fig. 1. BT system modeling method tasks [2]

The need for V&V of the SysMM system models was addressed from the beginning. SysMM tasks include V&V activities to ensure the quality of the deliverables. For instance, the *Operational Analysis* deliverables (left side of Fig. 1) are first verified automatically by the system modeling tool to check if the model elements and diagrams are conform with the associated guidelines. Then they are validated by the domain experts to ensure that the system model representation complies with the specification of the real-world system and the system requirements.

However, through the deployment of SysMM on several projects, the implementation of V&V solutions started to get very challenging due to the many changes triggered from the various dimensions such as the applications of modeling (e.g., functional description and variant management) and the hierarchy levels (e.g., train, consist and subsystems). Therefore, the need for a generic and reusable V&V solution was addressed to improve the V&V activities and hence optimize the deployment of MBSE. The targeted approach was built on the following objectives:

- Enable formal, generic and reusable V&V methods to be used across different projects and different departments.

- Ensure an early start of the V&V activities with regards to the system models development and keep it running in parallel to the SysMM tasks.
- Support V&V automation as much as possible to reduce the time consumed on V&V activities and avoid any potential errors due to manual actions.

3 State of the Art

As explained in [7,8], SysML on its own is not the best suited to apply a development method or build meaningful models in systems engineering. We have to ensure that we manipulate system concepts that are represented by corresponding model elements, along with proper semantic and relationships. A good example would be the lack of elements representing a function, which lead to the creation of specific methods on how to define a functional architecture based on SysML [9]. However, it is possible to adapt SysML to our needs through the use of profile, constraints and additional semantic. The Arcadia method [7] is an example of an adaptation of SysML to system development using system concepts. Arcadia is not considered as a DSML by its creators because of the broad scope of its application and its links to modeling standards. However, Arcadia does not follow the SysML standard fully, and has fixed concepts linked to the modeling elements. BT developed a profile that aims to give semantics to SysML elements while following a general modeling method that could be used also for other systems beside trains. From this comes the need for the verification of the models according to the semantics defined in the profile. The difference with Arcadia is that we can adapt our semantics and our profile depending on ours needs and the method used, without relying on a fixed solution and tool.

We consider here an existing solution for SysML model V&V and several examples of its application. The same way system V&V is different from model V&V, there are differences in the ways to perform V&V. Before considering common V&V solutions such as tests or model checking, which would require for the model to be executable, we want to check if its construction is coherent and holds correct meaning compared to a real system. As shown in [10,11], it is possible to have an implementation and verification of a SysML profile through the use of the Object Constraint Language (OCL) [12]. OCL enables to define constraints on a model, which we refer to as *rules* in this paper. We speak of verification rules and validation rules depending on their usage, but they are often called validation rules in practice, as shown in the several tools using this mechanism [13–15].

While OCL is widely used for this purpose, V&V rules can be developed in other languages supported by the modeling or analysis tools. For this reason, we consider the model V&V solution studied in this paper to be the rules mechanism, whether the rules themselves are coded in OCL or some other language.

Regarding the use of OCL to check or analyze a model, we can find several examples of its adaptation to industrial context and needs, offering technical solutions [16]. Some, such as [17] include OCL as a V&V solution in a process for models and instances design. In this paper, we focus on its use in a much

broader context, that is a system development process including many kind of models and taking into account the work of several modeling teams across different projects. We use OCL rules to enforce a semantic and detect error in the model representing the system. Validation using OCL rules is technically possible but it is currently not practical to develop those in a project context, as it will be explained further in this document.

4 BT SysMM V&V

4.1 Method Stakeholders

Figure 2 shows the context of SysMM V&V and the roles of its stakeholders. The V&V activities are part of SysMM and embedded within each task of SysMM (e.g., Operational Analysis). They start in parallel and continue until the deliverables of the SysMM task are verified and validated. Moreover, there is a common V&V part across all the tasks of SysMM, related to the generic and reusable models (such as the model library elements and glossary).

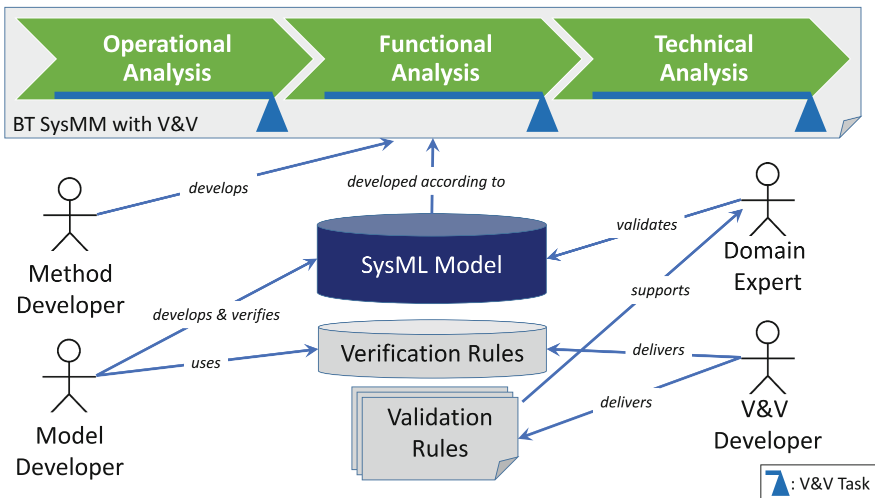


Fig. 2. BT SysMM verification and validation stakeholders context

The context in Fig. 2 indicates that the SysML model is the system of interest under which the V&V takes place. The verification rules are also represented with a model icon because they are implemented using OCL directly in the systems modeling tool. Both the SysML model and the verification rules are included in a project model, whereas the validation rules are documented in a formal textual format and shared through a common guideline. Validation rules are currently broad and/or abstract considerations that cannot be evaluated by a script. While

verification check the model and its semantic, validation targets the information expressed in the model regarding the system requirements and expectations. We could define lists of validation rules that check specific considerations expressed by domain experts, but quite often, the lack of resources (time and skills) to develop and use such rules during the project is a challenge.

The SysML model represents an abstraction of the real world system (e.g., train, subsystems or components). Furthermore, the SysML model is being developed based on the defined method and guidelines bundled here with the BT SysMM. The SysMM V&V identifies four stakeholder roles with their own responsibilities and competencies. Table 1 lists these four roles and describes them in detail.

Table 1. The BT SysMM V&V stakeholders roles description

Stakeholder	Role description
Method developer	Is responsible for defining and developing the system modeling method, its guidelines, training courses and tools' customization specifications. This also includes the V&V method parts and their relationship to other method parts. The method developer possesses a unique governance role in monitoring the deployment of the method on projects to ensure the reusability of delivered system models
Model developer	Is a member of the modeling team that is responsible to develop the system models and verify them according to a defined set of verification rules based on project needs. The verification process is done automatically by the system modeling tool and can be set to be active all the time or triggered by the model developer
V&V developer	Is responsible to develop and maintain the verification and validation rules based on the input from the method developer, domain expert and project needs. Additionally, this includes analyzing the V&V requirements, implementing, testing and delivering them. It is the role of the V&V developer to ensure the reusability of V&V rules across several projects
Domain expert	Is a member of the architects team who possess the authority and knowledge in a particular railway technical domain, e.g., brake, propulsion or train control. The domain expert plays a crucial role in validating the system models' content based on his own experience of the real-world system represented by system models

It is crucial for these roles to be well defined in the company, as not everyone should be able to define, develop, apply or change rules implementing the modeling method or defining the conditions that models have to satisfy to be validated. While we can define any arbitrary number of users, the definition of

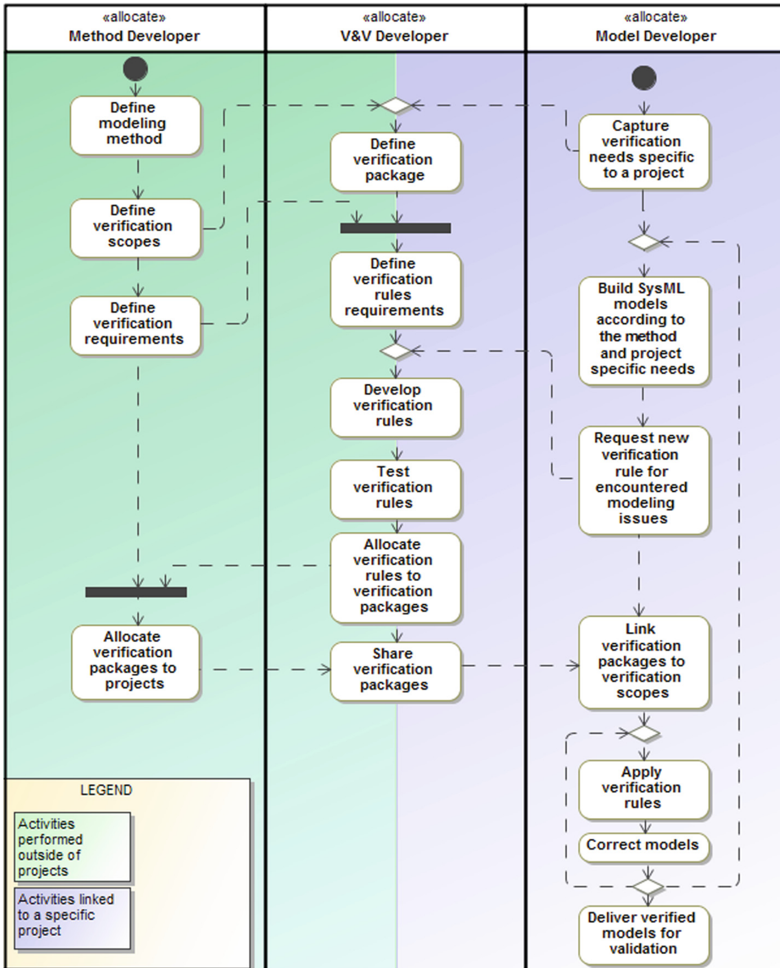


Fig. 3. Verification method

the modeling method and the management of the rules and their packages should be allocated to specific entities. This allows to centralize the skills, development efforts, and rules specifications, while avoiding conflicts and incoherences among the modeling teams. This is part of how we can address the challenges in change management, reusability and conflicting rules discussed later in Sect. 6.

4.2 V&V Method Overview

As we presented the different roles in the previous section, we now show the method and work process they follow in order to specify, develop, share and apply verification rules. This is illustrated in Fig. 3. As validation rules are not yet managed using the rules mechanism, they are not part of the method presented.

Rules are defined and used for a specific purpose and context. A key aspect of our implemented solution is the allocation of verification rules to modules or packages. In this way, rules in a same package can share a same application context and correspond to a same step in the modeling method with the related semantics. As the packages are managed by the method developers, they can be communicated to any team, enabling uniformity and reuse. Rules specific to a project will be contained in their own package. When working on a server, the packages can be automatically updated. Choosing the right verification packages enable us to define, apply and adapt our semantics. Packages can be versioned to be able to work on older projects. We can define packages providing the semantics of other modeling methods when working with or for other providers. Packages are to be built so as to separate conflicting rules.

Verification rules are not just a technical solution, they are specifications on how the modelers should work and what they should deliver. In order to specify, communicate, understand and use the rules, a proper documentation is required. Supposing you work with teams with different tools or an external provider, you can communicate the rules that have to be followed during modeling, even if they are not implemented or compatible with the tools. The documentation should at least specify for each rule: an ID, a target, a method, its current place in the life-cycle and the specification/constraint/error addressed by the rule.

4.3 Benefits

Aside from the semantics, the rules enforce the (modeling) methods and support work processes. Checking the rules on each step results in a report on the quality and level of advancement in the work done, enabling to proceed to the next development step after having checked for errors. Note that by verifying the relationships between concepts/elements, we ensure a certain degree of traceability. Supposing that we have modeled the requirements as artifacts, we can achieve part of the system validation just by ensuring that they are linked to other elements such as functions or scenarios. This is also true across abstraction levels, when switching the SOI from the system to a sub-system.

An advantage of the approach based on verification rules is that it is progressive, empiric, iterative and adaptive. We can specify, update and change the semantics and modeling rules over time. Note that most verification rules should be decided at the start of a project. While we can always develop rules during a project in answer to an immediate need, we should not remove or change any of them once the modeling activities have started. A key point in BT is that new modeling methods are being developed and spread in the different company sites across the world, and with rules they are supported by a common and automatic solution. Modeling teams can check the models and learn at the same time the method implemented by the rules. They also provide a feedback and request new rules. Rules support the training of modeling teams as the rules enforce the way the method has to be applied. In return, the method developers learn from the experience of modeling teams. This create a dynamics that optimizes the work performed and the results obtained across projects, each supplying new rules

and improvements. This would not be possible if we were to impose a new tool with a fixed semantics.

4.4 Issues

Before and during development, we came across several issues that had to be solved, such as how to define our rules or how to reduce the time needed for them to be checked. Some of these were problems we wanted to address with our solution, others resulted from its implementation. This enabled us to express new needs and opportunities that will be presented as challenges in Sect. 6. There is also the matter of maturity between verification and validation rules. Finding a way to develop, apply and check validation rules in an automatic way will be the object of further studies. In the rest of the document, we focus on issues related specifically to verification rules or ones that apply to the rule mechanism solution as a whole.

5 Use Case Example

Traditionally, the work split during the model development between teams is based on the work breakdown structure which defines a list of scopes covering all functionalities of the SOI (e.g., train or subsystem). The functional scope *travel direction*, taken from [2], is used in this section to illustrate the application of SysMM V&V on an example from the railway domain. A scope here is referred to a part of the work breakdown structure of the whole function set. Figure 4 shows some of the SysML diagrams delivered using the SysMM operational and functional analysis tasks.

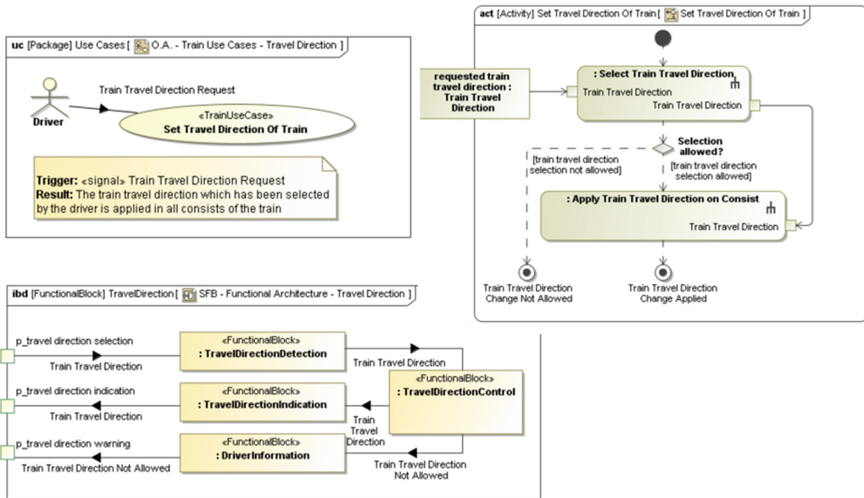


Fig. 4. BT SysMM diagrams example on which V&V is applied [2]

The operational analysis part is demonstrated through the use case and activity diagrams. The use case diagram defines the use case *"Set Travel Direction Of Train"*, its actor (i.e., the driver) and the respective trigger signal *"Train Travel Direction Request"*. The activity diagram describes the internal behavior of this use case in a generic manner independent from any specific functional or technical solution in order to reuse it in several projects.

The functional analysis part is shown with the internal block definition diagram where previously modeled activities are structured in a functional architecture that fits a particular train platform or project. The functional architecture defines all functions needed to cover the *travel direction* scope with functional blocks and their interfaces. These function blocks are linked back to the activities of the operational analysis and allocated later on to the technical blocks solution to ensure traceability.

The scope discussed here is one out of other hundreds of scopes normally modeled to describe the safety related functions of a train. Usually, a set of scopes is assigned to particular domain experts and model development team. The model developers, usually system engineers, takes the responsibility to develop the SysML models based on the input requirements of their own scope.

During the modeling activities, the model developers *verify* their models automatically based on the verification rules implemented in the tool. These rules are aligned with the deployed method and implemented using OCL in the systems modeling tool. Table 2 lists a sample of textual representation of the verification rules for model elements such as use cases or signals. The verification rules check automatically if model elements are modeled according to the defined

Table 2. BT SysMM rules examples

Sample Verification Rules:	
1.	A use case must own at least one activity
2.	A use case name must follow the naming convention guidelines (e.g., starting with a verb and all words are capitalized)
3.	A triggered use case must have at least one actor and one trigger signal
4.	A signal name must follow the naming convention guidelines
5.	Model elements, e.g., use cases, must be unique across the whole model
6.	Each function is linked to at least one activity
Sample Validation Rules:	
1.	Are the use cases' actors complete according to the requirements?
2.	Are all actors and signals considered in the correct way with respect to the requirements linked to the use case?
3.	Does the use case activity describe the exact scenario of real operation as described in the requirements?
4.	Is the functional architecture solution (i.e., functional split and allocation) satisfying the relevant requirements?

method. If not, the model developer is getting a notice about the result of the check i.e., an error, warning or information. One can see from the list that the verification rules check also model consistency and completion.

After the model is verified, it is shared with the responsible domain expert for the sake of *validation*. The second part of Table 2 lists a sample of the validation rules relevant to the presented example. These rules are documented in a formalized textual format and offered to support the domain expert during his model validation activities. The validation rules are always traced back to the system requirements. It is the role of the domain expert to apply his experience in order to check these traceability links and confirm that the model specification is valid with respect to the provided requirements.

6 V&V Adoption Challenges

In the following, we contribute by providing the list of challenges faced during the adoption of the V&V work on SysML models at BT and discuss the need to achieve a common interpretation of them in order to start solving them.

SysML Tools Integration. It is often the situation within large organizations that different departments or different sites' locations use different SysML tools. Challenges do not lie in the tool's diversity instead with the integration between the different tools. Model exchange (i.e., elements and diagrams) is still hardly possible between different SysML tools. In case tool vendors offer it (i.e., normally with the XMI file exchange), an additional effort and customization is always required. The early phase objective of BT was to achieve a model V&V solution across different SysML tools from the beginning of the system model development and not only at the end, i.e., model delivery from each tool. Unfortunately, this objective for having a common SysML V&V solution across different SysML tools is still not possible without having an extra SysML-independent V&V tool which is not preferred. This lead us to have a flexible solution that is the rule mechanism. Rules can be documented and transmitted even if they have different implementations. What is lacking is a way to quickly develop, deploy and/or adapt them across tools and teams. As the rules relate to concepts and data, part of the solution could be to apply them on a metamodel and transport them from one tool to another, based on the work with OSLC [18,19].

Complexity with Large SysML Models. The evolution of systems, components and functions has hugely contributed in growing the number of elements and relations of SysML models. For instance, modeling one functional scope, e.g., brake control or propulsion of a railway vehicle on the vehicle level only, includes in average 20 use cases, 150 signals and 25 function blocks. Covering only the safety related functions on the vehicle level, one should multiply these numbers by around 100 other scopes. The issue here is not only with the high number of model elements but with the dependencies inside a model or across

several models. Very often when dealing with large systems and large teams, complexity issues arise where existing tools and methods can reach their limits in solving them. This kind of high complexity levels has a huge influence on the V&V benefit. Therefore, a suitable solution should deal with the complexity issues and not only on tools-level but also on methods and processes.

Conflicting V&V Rules. As we do not rely on a tool applying a DSML and separating the different scopes of study, we tend to use the same SysML elements for different purposes. For example, while we do not apply a stereotype on a sequence diagram, we do not use the same type of signals in the messages depending on whether we are making an operational analysis, that is an analysis of the system behavior from an external point of view, or whether we are making a functional analysis, meaning we specify the signals exchange between functions. Using the right signals according to one analysis will result in not using those required for the other. Applying a verification rule depends on the verification goal, the scope and the type of the element. If we check all verification rules on the whole project, then independently from performance issues, there will be contradictions and the model will never be considered correct.

V&V Managed Reuse. Many organizations still follow an opportunistic and isolated reuse approach, where a set of data is copied and pasted from one context to another. Unfortunately, this still happens even with work performed on SysML models and results in losing the "source of truth" as soon as the copied source or pasted target is changed. During the early stages of modeling, V&V rules are often created only for a particular deployment (project or product) and thus specific. Reusability between different deployments gets complicated without proper modularity concepts for defining the communality and variability of the V&V rules. According to BT MBSE objectives, reuse is a key factor for improving the system models V&V efficiency. In order to achieve a managed V&V reuse, modularity, governance and variability management must be in place for V&V solutions.

V&V Change Management. The work with V&V rules is subject to many changes, most of which are due to models or rules modifications, emerging from different stakeholders for the aim of optimization – of V&V rules. Working around those changes is eased by following our method, managing rules in PLM and documenting them, but it is not enough. Each change request triggers a sequence of tasks (e.g., review request with impact analysis, change approval, change implementation with review and reporting) in order to reach the final successful implementation and closing the change request. Such tasks are often grouped under the term of change management. Although the usage of methods as agile, scrum and kanban helps a lot in addressing the change and delivering value with a quick impact and continual basis, the responsibility still lies on the personnel side of the team involved. Particularly, dependencies between change

requests are often not visible from the beginning, consume time and impact the V&V solution delivery timeline. Moreover, the integration between agile tools and SysML tools – to achieve a full traceability – is still very challenging when dealing with multi-user environment. Although the technology is heading towards cloud based solutions, their low performance due to large models is still an issue.

V&V Optimization. Although the work done at BT with regard to V&V has evolved enormously for the aim of optimization during the last years, it still requires high effort to analyze all relevant rules and the order of executing them to deliver better V&V results. Particularly, defining the dependencies between the verification rules is still very challenging and needs a lot of tool customizations, specific method solutions and personnel effort. Therefore, there is a need to investigate which other domains could solve the optimization challenges, for instance in [20] the combination of Statistical Machine Learning and OCL demonstrates how Artificial Intelligence can support in solving this challenge and in [21] an implementation of machine learning for a model-based conceptual design evaluation is demonstrated.

7 Conclusion and Future Directions

In this paper we presented the work related to the verification and validation of SysML models from an industrial perspective. Although the usage of OCL is well known for model verification, we first contributed by describing the method and roles used at BT to achieve efficient V&V results and accelerate the system development process with less time consumed on testing and system validation. Our second contribution concerned the description of common challenges faced with model-based V&V in large organizations. After having identified these challenges, a common understanding between the whole modeling team was achieved to justify the reasons behind the previous pitfalls and failures.

Our future work spans in two main directions: on one hand, we aim to analyze and the describe the fulfillment of the V&V method developed in relation to the challenges discussed in this paper. In so doing, we expect to identify the challenges which cannot be solved through method, process or tool solutions. On the other hand, we aim to apply new domains, such as Artificial Intelligence (AI) and machine learning to use the large amount of available data, let the AI system learn from it and support with an optimized V&V results. Finally, we aim to use this work to trigger the MBSE community and particularly the SysML working group in upcoming conference workshops to consider V&V more in detail in future SysML versions (e.g., 2.0) in order to solve industrial adoption challenges from a language prescriptive. The SysML V2 working group [22] states that the next version of SysML should enable a concise representation of the concepts and be able to validate that the model is logically consistent. It should also be highly adaptable and customizable in regard of domain specific concepts. The rules mechanism presented here enables to do both, and it would be interesting

to be able to express the rules based on the SysML language rather than on its implementation in tools, while taking in account the other challenges we expressed.

Acknowledgements. This work is supported by Bombardier Transportation SAS and the ANRT CIFRE grant #2016/0262.

References

1. OMG: OMG Systems Modeling LanguageTM Version 1.5 (2017). http://www.omg.org/spec/SysML/1.5/
2. Chami, M., Oggier, P., Naas, O., Heinz, M.: Real world application of MBSE at Bombardier Transportation. In: The Swiss Systems Engineering Day (SWISSED 2015), Kongresshaus Zurich, 8th September 2015
3. INCOSE: Systems Engineering Vision 2020. Version 2.03 edn. Technical Operations, International Council on Systems Engineering (INCOSE), September 2007. INCOSE-TP-2004-004-02
4. IEEE: 1012–2012 IEEE Standard for System and Software Verification and Validation
5. Stecklein, J.M., Dabney, J., Dick, B., Haskins, B., Lovell, R., Moroney, G.: Error cost escalation through the project life cycle (2004)
6. Chami, M., Morkevicius, A., Aleksandraviciene, A., Bruel, J.M.: Towards solving MBSE adoption challenges: the D3 MBSE adoption toolbox. In: 28th Annual INCOSE International Symposium, Washington DC, USA, 7–12 July (2018)
7. Bonnet, S., Exertier, D., Normand, V.: Not (strictly) relying on SysML for MBSE: language, tooling and development perspectives (2015). OCLC: 255348295
8. Guizzardi, G.: Ontological foundations for structural conceptual models. Ph.D. thesis, Centre for Telematics and Information Technology, Telematica Instituut, Enschede, The Netherlands (2005)
9. Lamm, J.G., Weilkiens, T.: Functional Architectures in SysML. In: Proceedings of the Tag des Systems Engineering (TdSE 2010), Munich, Germany (2010)
10. Berkenkotter, K.: OCL-Based Validation of a Railway Domain Profile (2006). OCLC: 248918751
11. Dragomir, I., Ober, I., Percebois, C.: Contract-based modeling and verification of timed safety requirements within SysML. Softw. Syst. Model. **16**(2), 587–624 (2017)
12. OMG Specification: Object Constraint Language V.2.4. Object Management Group pct/07-08-04, February 2014
13. Eclipse - Papyrus Guide: Validate (OCL) constraints of a profile
14. Sparx Systems: Model Validation (2016)
15. No Magic Documentation: Creating validation rules - MagicDraw 18.2
16. Altenhofen, M., Hettel, T., Kusterer, S.: OCL support in an industrial environment (2006)
17. Delmas, R., Pires, A.F., Polacsek, T.: A verification and validation process for model-driven engineering. EDP Sciences 455–468 (2013)
18. OMG: OSLC4mbse Working Group
19. Shani, U.: A Case for a SysML OWL Ontology (2014)

20. Zhao, L., Li, F.: Statistical machine learning in natural language understanding: object constraint language translator for business process. In: 2008 IEEE International Symposium on Knowledge Acquisition and Modeling Workshop, pp. 1056–1059, December 2008
21. Chami, M., Bruel, J.: Towards an integrated conceptual design evaluation of mechatronic systems: the SysDICE approach. In: Proceedings of the International Conference on Computational Science, Reykjavík, Iceland, pp. 650–659 (2015)
22. OMG: SysML v2 RFP Working Group

Property-Aware Unit Testing of UML-RT Models in the Context of MDE

Reza Ahmadi(✉), Nicolas Hili(✉), and Juergen Dingel(✉)

School of Computing, Queen's University, Kingston, Canada
{ahmadi,hili,dingel}@cs.queensu.ca

Abstract. Modern cyber-physical systems are complex to model due, among other things, to timing constraints and complex communications between components of such systems. Therefore, testing models of these systems is not straightforward. This paper presents an approach for automatically testing components of UML-RT models with respect to a set of formally defined properties. Compared to existing model-based techniques where abstract test cases are complemented with their concrete counterparts, our approach solely leverages on constructs provided by the modeling language to express all artifacts (component to test, test harness, the property of interest) and existing code generator to generate test cases. This helps to reduce the cost of ensuring the consistency between code- and model-level tests. Moreover, to reduce the number of test cases and the associated cost, our approach integrates our test case generators with slicing techniques to reduce the size of the components. A prototype implementation has been sketched and our approach has been evaluated over two case studies.

1 Introduction

Real-time embedded systems (RTE) which are often safety-critical [3] typically interact with physical components (e.g., sensors or actuators) which entails real-time constraints on the behavior of the system, and hence require intensive testing to ensure that they meet stringent requirements. For example, a control system of an elevator's door must guarantee that the opening of the door must not occur when the elevator is moving. As a second example, if an infusion pump system generates a *low-pressure* signal, it must recover automatically and send a follow-up *healthy pressure* signal within 5 min of the first signal [10].

Using code-centric only approaches for developing complex RTE systems is very challenging. Model Driven Engineering (MDE) techniques tackle this challenge by raising the level of abstraction on which the developers construct software. If a model contains faults, these faults will propagate to any refinement of that model or the code that is generated from the model. Therefore, finding and resolving faults at the model level typically is more efficient than finding the same faults in the next stages. Hence, for MDE approaches to succeed, appropriate techniques and approaches for model-level testing and validation are required.

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 147–163, 2018.
https://doi.org/10.1007/978-3-319-92997-2_10

In many existing model testing approaches [7,11,13,15] the system is only tested at the model-level without generating code from the model and running it, so these approaches make assumptions about the environment. Moreover, if only the model is validated and not its code generator, there might be inconsistencies between the expected and the actual behavior of the generated code [25]. It is also possible to generate code from an RTE model and use existing code-centric approaches for testing, but this contradicts the goals of MDE which aims to remove the accidental complexity of source code. Moreover, often understanding the generated code can be challenging for model developers, due to the low readability of the generated code or simply because the model developer is not familiar with the language of the generated code.

In this paper, we present an approach and prototype tool for unit testing models of RTE systems. We use UML-RT, a popular modeling language that is nowadays used for modeling complex industrial systems and is supported by various open-source and commercial tools (Eclipse Papyrus-RT [5], IBM RSA-RTE, IBM RoseRT). Using our approach, we overcome some of the issues mentioned above for testing models of RTE systems. In our approach, we rely on the modeling language to transform a component of the model to a testable component (such that a test harness can drive and test it) and we construct the test suite and the test harness at the model-level, so we can rely on the standard code generator to generate code for the mentioned components and the glue code for integrating them. Using this approach, a modeler can test a model on any platform (by running the code generator for that platform) without being dependent on any particular target language (since the target language can be changed). Moreover, since models and components of RTE systems can be large, we propose concentrating the test generation on user-selected aspects of a model through properties.

In the next section, we briefly introduce UML-RT with an example model. We then explain our approach. Then, an evaluation of the approach is given and then we conclude the paper.

2 UML-RT Modelling Language

UML for Real-Time (UML-RT) [28] is a domain-specific language dedicated to modeling complex real-time systems with soft real-time constraints. UML-RT has its roots in the well-known Real-time Object-Oriented Modeling (ROOM) language [29]. It contains a small set of concepts and provides a light notation that makes it suitable for designing such systems.

The main concepts of UML-RT are *capsules*, *capsule parts*, *ports*, *protocols*, and *connectors*. A capsule is an independent component running in its own flow of execution. Capsules are hierarchically defined using *capsule parts*. A capsule part is an instance of a capsule encapsulated in another capsule. Figure 1 depicts the top level structure of a Cruise Control system modeled in UML-RT. The system consists of five communicating capsule parts. Capsules own ports, allowing them to communicate via message passing. To allow two capsules to communicate, capsule ports are typed with *protocols*, that is, a formal description of

the incoming and outgoing *messages* a capsule can receive from and send to other capsules. Two ports typed with the same protocol can be formally connected through *connectors*. Besides, one of them must be *conjugated*. Conjugated ports invert the direction of incoming and outgoing messages such that a base (non-conjugated) and a conjugated port typed with the same protocol can be connected [28]. Ports can be of different kinds: external ports are boundary ports exposed by the capsule to be connected with other capsules. Internal ports allow a capsule to communicate with the capsule parts it contains. Relay ports are, as their name suggests, ports that are only used to allow messages to cross the boundary of a capsule, enforcing encapsulation.

On the behavioral side, state machines model the behavior of capsules. Example state machines are shown in Sect. 4. A UML-RT state machine is an extension of a Mealy state machine [22] augmented with extra features, including state actions, composite states, and concurrency.

To facilitate modeling, the UML-RT Runtime System (RTS) library includes a set of services that provide utilities for, e.g., importing parts, logging messages, and scheduling. For instance, if a UML-RT capsule contains other capsules, then the container can dynamically import the containing capsules using a *frame* port. For scheduling, the RTS provides *timer* ports.

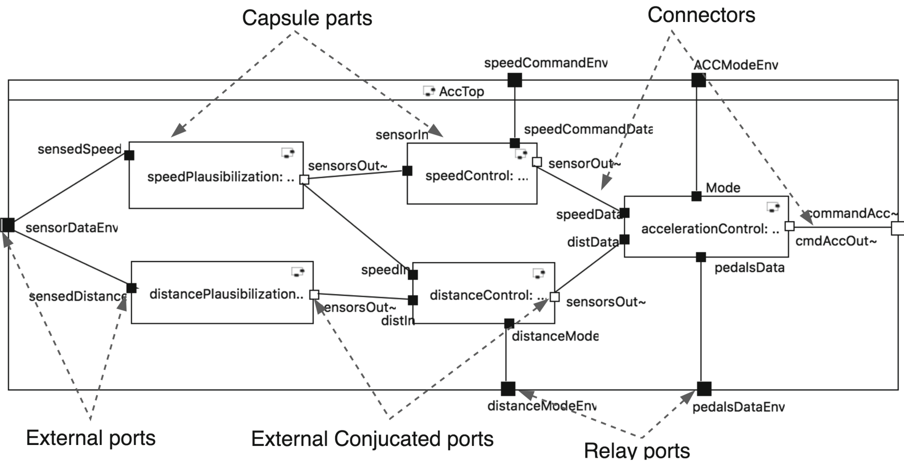


Fig. 1. Adaptive cruise control system model with five capsule parts

3 Approach

In our approach, we rely on the modeling language to express artifacts such as test harness and properties, therefore our approach is not dependent on any other particular programming language, any external test harness, or Unit Testing frameworks (such as Google Test or JUnit). The two terms *Capsule Under Test*

(CUT) and *State machine Under Test (SUT)* refer, respectively, to the UML-RT component we want to test and its state machine denoting its behavior. The *Test Harness (TH)* is the component that tests the CUT.

3.1 Approach Overview

Figure 2 illustrates the workflow associated with our framework. It consists of five consecutive steps (respectively numbered 1a, 1b, 2, 3, and 4). For testing real-time systems modeled in UML-RT, first, a test property is expressed using a UML-RT state machine (Step 1a). The test property is used for examining the behavior of the capsule under test w.r.t. various test inputs. Section 3.2 explains the test properties. We then prepare an individual capsule (Step 1b) from the UML-RT model. The capsule to prepare is denoted C in Fig. 2. Preparing the capsule includes taking the capsule out of its context (its connections with other capsules) and slicing it w.r.t. the specified property. The preparation puts the capsule into a new, testing-specific environment (similar technique has been used before in other contexts [8,31]). Due to slicing the CUT may be significantly smaller than the original capsule, even though the slicing step is optional and our framework works without slicing. The complete process is detailed in Sect. 3.3.

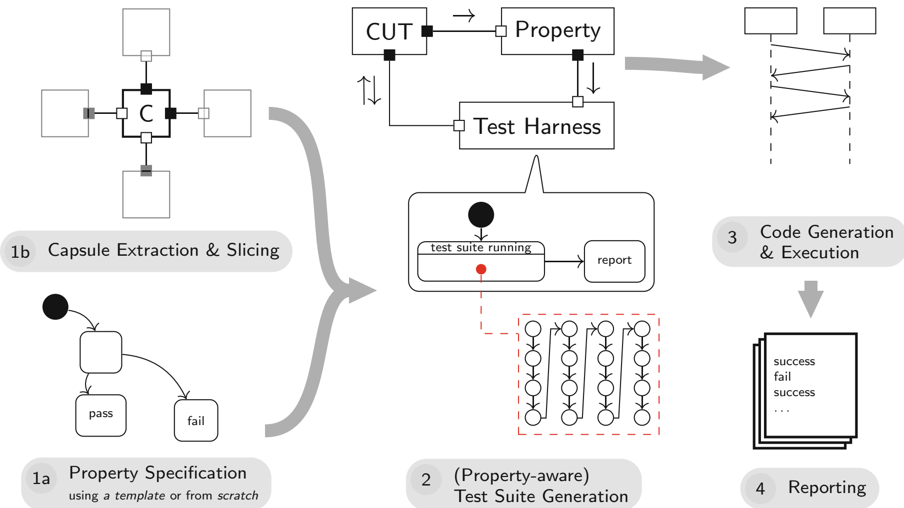


Fig. 2. Framework overview for testing models of real-time systems

The result of the two first steps 1a and 1b is a new model containing the isolated capsule to test CUT, an empty test harness TH, and a Property capsule whose behavior was previously modeled in Step 1a. The middle stage in Fig. 2 shows the three capsules, their connections, and the directions of message flow between them. The goal of the TH is to stimulate the isolated capsule CUT by providing

a series of test inputs. To this end, the test suite is generated and captured as a state machine during Step 2 w.r.t. the property to test and are injected into the TH. In our framework, three strategies are used to generate the test suite. The three strategies are discussed in Sect. 3.4.

During Step 3, C++ code can be generated and the test cases can be executed. As mentioned at the beginning of this section, all artifacts are expressed in the same modeling language. Therefore, we can use any standard UML-RT code generator to generate an executable implementation containing the CUT, TH, and the property. During execution, the TH is executed in order to exercise the CUT to verify whether the property modeled in Step 1a holds, given the test cases generated in Step 2. Once the system is executed, a report is generated during Step 4. The report lists all successful and unsuccessful tests and provides some evidence of the validity of the property w.r.t. the implementation.

We have sketched a prototype implementation to automate testing of UML-RT models using our approach. This prototype has been built as a set of Eclipse plug-ins on top of Eclipse Papyrus for Real-Time (Papyrus-RT). It allows users to select a UML-RT capsule, slice it w.r.t. various criteria, and generate test cases and the test harness from the slice or from the original capsules[1]. In the remainder of this section, we elaborate our approach by detailing each step.

3.2 Property Specification

Test properties are formal specifications of informal requirements, specified by means of simple state machines [10,14]. Even though languages such as Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL) are very expressive languages for formal property specifications, formalizing a property in a state machine can be more intuitive for system engineers who are non-experts in such logic formalisms [10]. Moreover, as opposed to state machine properties, debugging LTL properties is difficult. Properties express predicates on message sequences and thus expect certain sequences of output messages from the CUT.

In our framework, test properties direct the framework to reduce the number of test cases required for testing the property. Figure 7 on page 12 shows a property defined for a Collision Avoidance (CA) system. Based on the state machine of this property, this property fails if the CA system generates a *reverse* output message followed by a *vibrate* output message or vice-versa. Our framework supports defining *time-sensitive* properties, as well. To this end, outgoing messages from the CUT carry a timestamp as an extra parameter, so time-sensitive properties keep track of these timestamps to compute timespans between messages. This is because, in the case of asynchronous communication between capsules in a model, the exact amount of time between sending a message from the CUT and its reception and consumption by the property is undermined [24]. Figure 7 on page 12 shows a time-sensitive property for a traffic light system (the property and the system are both explained in Sect. 4).

[1] The tool along with the sample models presented in this paper can be found at: https://bitbucket.org/rezaahmadi/umlrtunit.

3.3 Capsule Preparation

The process of capsule preparation consists of two steps: extracting a capsule from its context and slicing it w.r.t. the specified property of interest to create a possibly smaller capsule (our CUT).

A UML-RT capsule can be large in terms of its behavioral (state machine) and/or structural aspects (ports, parts, connectors). Since we are only interested in aspects of a capsule (state machine or structure) that may affect a property of interest, we conduct slicing on a capsule by taking the specified property as the criterion to preserve only related parts of a capsule. The slicing operation, as we will show in Sect. 4, may produce a significantly smaller capsule, which results in a smaller test suite, and thus a more efficient testing. In the following, we will explain how a *slicing criterion* is extracted from a property φ.

So, assume φ is a property and *transitions(* φ *)* are the transitions of the state machine of φ, then criterion Cr_{φ} is:

$$Cr_{\varphi} = \{\, trigger_t \,|\, t \in transitions(\varphi) \,\} \tag{1}$$

where $trigger_t$ is a message that triggers the transition t. Once the criterion is constructed, the slicing is conducted on the capsule.

To compute the slice of a capsule, we construct a dependency graph that represents structural as well as behavioral dependencies between UML-RT model objects. The behavioral dependencies capture the dependencies between objects inside a capsule boundary, e.g. the dependencies between states, transitions, and action codes, as well as the dependencies of these objects with ports on the same capsule. The structural dependencies, on the other hand, express potential connections between ports and parts in a composite capsule. Our slicing algorithm is based on traversing the edges in the constructed dependency graph for the given slicing criterion. Since our criterion is a set of messages M that are sent to a set of ports P on a capsule C, we first find all the transitions T in the state machine of C responsible for sending these messages to P and the nodes N in the dependency graph that represent T. Similar to other tools [4,20], to compute the slice, our tool marks other nodes that are reachable from N to identify the relevant model elements from the UML-RT capsule.

So, let $Slice_{\varphi}(C)$ be the results of slicing C with respect to φ, $Outgoings(C, p_i)$ be all outgoing messages of capsule C that are sent on port p_i of C, and $Outgoings(Slice_{\varphi}(C), p_i)$ be all outgoing messages of the slice that are sent on port p_i of the slice. Slicing will preserve the messages that the property state machine listens to. More formally:

$$\begin{aligned} \forall trigger_t \in Cr_{\varphi} \,|\, trigger_t \in Outgoings(C, p_i) \implies \\ trigger_t \in Outgoings(Slice_{\varphi}(C), p_i); \end{aligned} \tag{2}$$

In other words, $Slice_{\varphi}(C)$ preserves the behavior and structure of those parts of C that affect the behavior of φ. Therefore, a test case of the test harness TH_{φ} will cause the original capsule C to exhibit an execution that violates property φ iff that same test case also causes $Slice_{\varphi}(C)$ to violate the property.

Note that a slice is a well-formed subset of the original capsule, which means it includes all original parts of the capsule (including states, transitions, and ports) to maintain the executability feature of the capsule. This is obtained by various dependencies that we consider during slicing as we explained.

3.4 Test Suite Generation

After a capsule is extracted and its slice is computed, test suites are generated on the UML-RT capsule representing our slice. We propose three techniques for generating test cases for UML-RT capsules: *Random*, *Simple Exploration*, and *Symbolic Execution*. For both *Simple Exploration* and *Random* techniques, we consider a *test budget* based on two user input parameters: *test suite size* and *test length* to limit the size of the test suite. Moreover, in each test, for message parameters with primitive types, we generate a random value (based on a range specified by the user), and for complex data types, a random value for each member in that type. In *Symbolic Execution* our test budget is determined by the size of the (SET), which is explained in the following part. Each of these three techniques has different strengths and weaknesses that are detailed in Sect. 4.

Test Generation Using Symbolic Execution. We used the Symbolic Execution (SE) technique proposed in [34] for generating test inputs (similar to Rapos and Dingel [27]). SE traverses a state machine using a breadth first algorithm and creates a SET that represents all the possible executions of that state machine. Figure 3 shows how we integrated SE in our approach for property-aware test generation. First, slicing reduces the size of the capsule by only considering the parts of the capsule relevant for checking a property. Then, SE is used to generate a reduced SET from the 'sliced' capsule, hence reducing the number of test cases that need to be generated. Once the SET is generated from the slice, test cases are generated by solving symbolic paths (using Choco [18]).

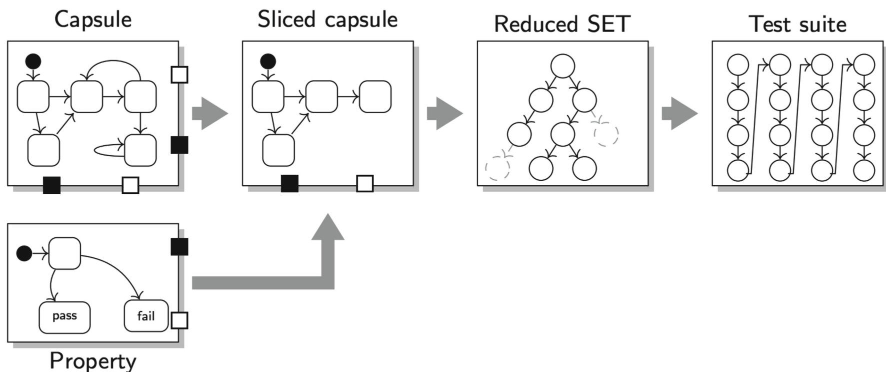


Fig. 3. Property-aware test suite generation using symbolic execution

Other Test Generation Techniques. Due to limitations of SE (not supporting complex data types as message parameters and supporting only a subset of UML-RT action codes), we implemented two other alternative approaches for generating tests for UML-RT capsules: *Random* and *Simple Exploration*. *Random* is a black box test generation technique where test cases are formulated directly from the capsule ports (*internal, timer, frame*, and *user-defined* ports) connected to the capsule. We first find the protocol (type) of each port and create a collection containing the union of messages with *Output* direction from conjugated ports and messages with *Input* direction from non-conjugated ports. A test case is generated based on different combinations of messages. On the other hand, *Simple Exploration* is a white box test generation technique with similarities to previous approaches of test generation for state machines [19,23,30]. We traverse the state machine state by state and for each outgoing transition of the current state, we generate a pair, where in each pair, the first element represents a trigger that fires the transition, and the second element represents the set of outgoing messages that are generated by taking that transition.

3.5 Generating the Test Harness State Machine

We transform the generated test inputs from the previous step into a state machine. We then inject this state machine as a composite state inside the *test suite running* state in the TH (cf. Fig. 2). As a result, the TH and the CUT communicate in a *ping-pong* fashion, CUT requests for the next test input and TH sends the next input until the end of the current test case and finally the test suit. Figure 4 shows parts of the test suite in the TH. A new test case starts once TH receives *nextTest*. Each state has a state entry action code, which is responsible for sending signals to the CUT (as well as collecting the sequences of sent signals for reporting purposes).

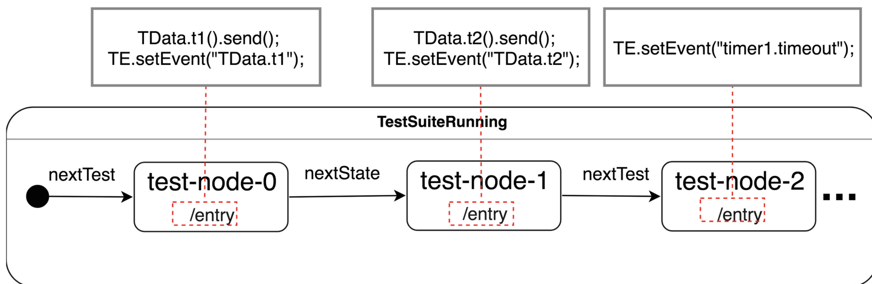


Fig. 4. Parts of the TH representing two test cases.

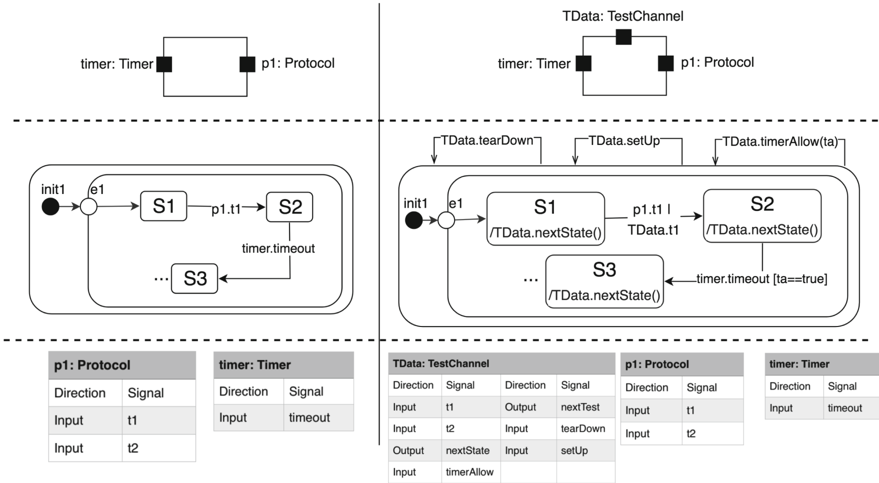
3.6 Refining the Capsule to Make It Testable

Figure 5 illustrates the capsule refinement process. The left side of Fig. 5 shows the capsule to refine. The capsule owns two ports, a timer port, and a user-defined port $p1$ typed with a user-defined protocol. The behavior of the capsule to refine is defined using the state machine shown below the capsule. The state machine of the capsule to refine consists of an initial pseudostate $p1$, and three simple states $S1$, $S2$, and $S3$ embedded in a composite state.

The right side of Fig. 5 shows the transformation of both the capsule and its state machine to prepare it for being driven by the TH. Both structure and behavior are transformed via Model-to-Model (M2M) transformation rules. On the structural side, rules are applied to establish the communication between the TH and the CUT. To support this scenario, a new port called *TData* is added and typed with the *TestChannel* protocol. The *TestChannel* protocol is used to connect the TH to the CUT and to the capsule implementing the property. It includes all the required messages that can drive the CUT in its state space (cf. Fig. 5). In addition, it defines two output messages respectively named *nextState* and $nextTest$, and two incoming messages *tearDown* and *setUp*. The two output messages, *nextState*, and $nextTest$ are messages the CUT can issue to the TH to request the next input signals. This communication between the two continues until the entire sequence of test cases inside the TH runs out of input signals. Since more than one test case might be executed on the CUT, the two input signals *tearDown* and *setUp* are issued by the TH to move the CUT into its initial state and initialize its variables.

Figure 5 includes the two messages $t1$ and $t2$ from the port $p1$ but excludes the *timeout* message from the timer port. Timeout signals are sent to each capsule only by the RTS timing services and after the timeout happens the CUT informs the TH, so the TH can send the rest of the test inputs. Also, note that the above definition does not take into consideration the fact that ports can be conjugated in UML-RT. For conjugated ports, the list of incoming and outgoing messages are inverted compared to their definition in the protocol.

On the behavioral side, the state machine of the CUT needs additional transformations to be driven by the TH. To do so, our rules explore the entire state machine to determine the set of all transitions that are triggered by messages received through the user-defined ports. For each triggerable transition, an extra trigger is created allowing the capsule to be triggered by the TH. Applying this transformation to the SUT on the right side of Fig. 5, the transition between $S1$ and $S2$ is now refined so it can be triggered upon reception of either $t1$ from $p1$ or its doppelganger from $TData$. Note that, for a transition to be fired, only one trigger needs to be activated. Finally, one last transformation is required to allow the TH to drive it. This last transformation allows the SUT and TH to execute in an alternate fashion, i.e., after sending an input to the SUT, the TH waits for an *acknowledgment* message from the SUT before it sends the next message. Acknowledgment messages are modeled using a *nextState* event added in each stable state of the SUT and triggered when entering the state.



Left: original capsule with its ports and state machine. Right: a transformed to testable capsule with same objects. From top to bottom: capsules, state machines, and protocols

Fig. 5. Illustration of the capsule refinement process

4 Case Studies

We evaluated our approach to detect property violations in two case studies. The first one, the *Collision Avoidance* is an industrial-sized system from the automotive domain originally designed in Stateflow at the University of Waterloo [17] and some complementary aspects from [2]. We manually converted these models to a behaviorally equivalent UML-RT model. The second one is an academic model of a *traffic light*. Both models are described below.

Collision Avoidance (CA) System. This system prevents or mitigates collisions by continuously monitoring the road ahead and parts of the side-fronts of the vehicle. Whenever an obstacle is detected, it notifies the driver by audible or visual alerts. In addition, the system automatically brakes, vibrates or steers the wheels in the opposite direction if it detects an imminent collision. Due to space limitation and simplification, only a small part of the system is shown in Fig. 6. In few words, the system behaves as follows: initially, it is *disengaged* (not shown here) and becomes *engaged* when the speed of the vehicle reaches 25 km/h. In the *Engaged* state, the system is constantly collecting various signals, such as signals that enable vibration, indicate the detected threat level, or the collision direction. Based on the threat level values that the system receives from other components of the vehicle:

– if the threat level is 1, a *warning(1)* message is generated and the system moves to the *Warn* state;
– if the threat level is 2, a *warning(2)* message is generated, and the vehicle is slowed down by applying a mild brake (30%). If *vibrate* is enabled, it also generates a *vibrate* message, and then moves to the *Avoid* state;
– if the threat level is 3, then a *warning(3)* message is generated, and a hard brake (80%) is applied. If the system receives a merging collision threat signal *(cd == 0)*, then a *reverse* signal is generated and if the signal shows a forward collision *(cd == 1)*, then a *vibrate* signal is generated (provided that the *vibrate* is enabled). The system then moves to the *Mitigate* state.

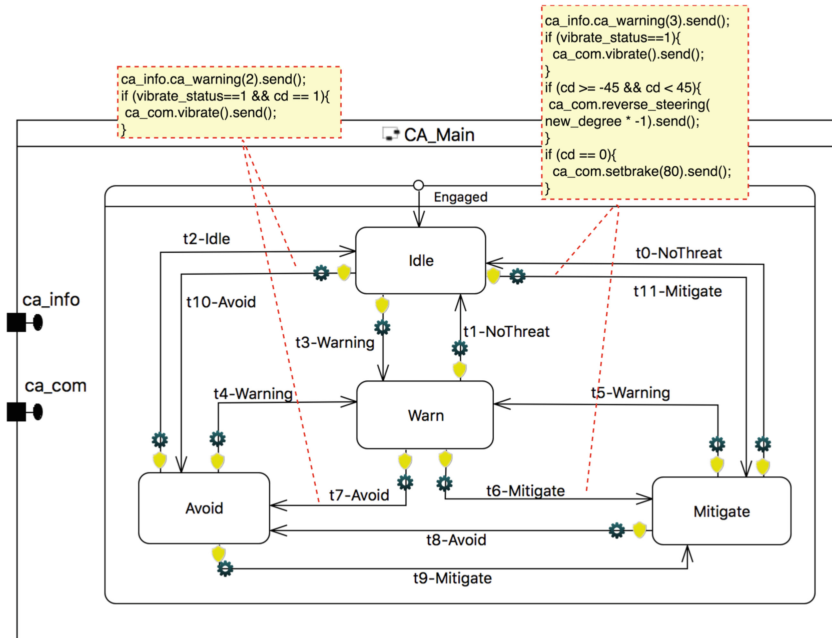


Fig. 6. Collision avoidance system (partial model)

Figure 6 shows one capsule of the system with two ports, *ca_com* and *ca_info*. *ca_com* receives commands from the brake system and threat measurements from other components of the system. This port is also used to send commands for braking and steering wheel systems. *ca_info* port sends information signals to the user panel such as errors or warnings. Based on the specification provided in [2, 17], the system vibrates the steering wheel if it detects a forward collision, and should steer the car in opposite direction if it detects a merging collision. So, we can imagine the following safety property in the system:

Property P1: The CA system should not vibrate the steering wheel (send vibrate signal to its ca_com port) and reverse the steering wheel (send a reverse signal to its ca_com port) at the same time.

In other words, every pair of *vibrate* and *reverse* signals must be separated by at least one *brake* signal. Violation of this property may be disruptive for the user who may lose control of the vehicle. In order to integrate this property into our framework, a state machine of this requirement is specified, which is shown in Fig. 7. Based on the figure, the property fails if the CA system generates a *reverse* output message followed by a *vibrate* output message or vice-versa.

Traffic Light Control (TLC) System. Due to space limitation, we do not show the model of this system[2]. TLC is a system that controls the lights based on several timers, but also on the number of cars detected by a camera. The system can monitor two lanes (where the left lane is used to turn left). It keeps track of the number of cars waiting for the green light on the two lanes of the road. Moving between the three states *Green*, *Yellow*, and *Red* is done with predefined timers. When the traffic light is red, specific signals can trigger an early transition to green when at least five cars are waiting for the green light on the main lane or the activation of a *left turn signal* if two cars arrive on the left lane. This is used to relieve traffic congestion. One property to check can be:

Property P2: If the system detects more than two cars on the left lane, then it must turn the left arrow on after at most 10 s.

The state machine of this property is shown in Fig. 7. When the property receives a *leftLane* signal and the input parameter of the message is less than two (less than two cars were detected on the left lane), then the property is restarted. If the input parameter is greater or equal to two, then the property expects a *leftArrowOn* signal within ten seconds to satisfy the property.

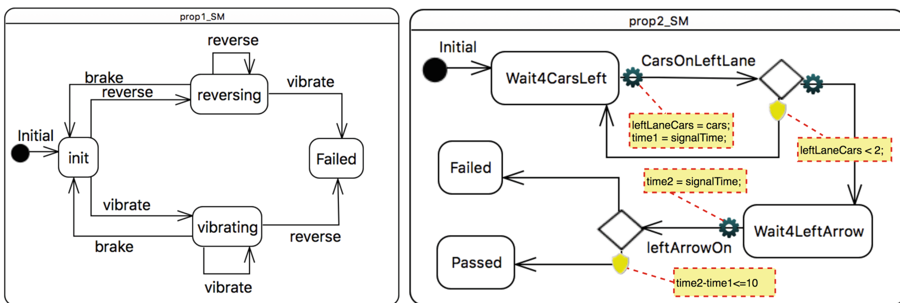


Fig. 7. Two requirements specified for CA system (left) and TLC system (right)

[2] The complete model can be found at: https://bitbucket.org/rezaahmadi/umlrtunit.

Evaluation Results. For testing the systems we started with Symbolic Execution (SE) as our first technique. By considering the property of interest in the CA system, we are interested in transitions in the system that send a reverse or a vibrate message to the *ca_com* port, therefore we slice w.r.t. the mentioned messages and port. In both cases (with/without slicing), we were able to catch a bug in the system: the action code of both *t6-mitigate* and *t11-mitigate* allow for generating both *reverse() and vibrate()*, since the first 'if' statement misses an extra check (*&& cd == 1*) to ensure a *vibrate* signal is generated if and only if the input signals show a possible forward collision. Table 1 shows the outcome of our experiment on the CA system using the SE technique. As shown in the table, in the CA system, slicing reduces the number of test cases and the time required to generate tests by a factor of 2. In this system, the computed slice only preserves *ca_com* port, since the slice is not dependent on *ca_info* port nor on the action code that communicate with this port.

Table 1. Effect of integrating slicing with SE for testing the systems

System	Collision Avoidance (CA)		Traffic Light Control (TLC)	
Slicing used?	Y	N	Y	N
Size (S/T/LOAC/P)	9/22/33/1	12/36/81/2	15/14/25/7	15/14/30/7
Test Cases (#)	3,387	6,271	517	549
Tests G.T. (s)	325.3	662.3	2,157	2,684
S.T. (s)	0.36	n/a	0.08	n/a
T.T. (s)	325.7	662.3	2,165	2,684

LOAC: Lines of action code; S: State; T: Transition; G.T.: Generation time; P: Ports; S.T.: Slicing time; T.T.: Total time

TLC is less complex than the AC system in terms of size of the state machine and lines of action code, but as shown in Table 1, slicing still helped to reduce the number of test cases. Note that, TLC has less number of action codes than CA, but at the same time TLC has more conditional statements in its action code, which leads to more symbolic execution paths, and hence, as shown in the table, it results in more test cases, and longer test case generation time for this system. In TLC, by running all the test cases, the property never enters *Passed* state. In fact, SE generates required test cases and their inputs parameters to drive the system such that the property reaches *Wait4LeftArrow* state, but a timer on the system required to activate a particular transition is not fired. The mentioned transition was responsible for activating the *leftArrowOn* signal, required by the property to be satisfied. By looking into the dependency graph generated by the slicing, we find out that this timer is not dependent on any other action code, which suggests that it is never set.

We also executed the two other test generation techniques (Random and Simple Exploration) on both systems and we applied the slicing operator on both these techniques, as well. We gave these two techniques the more amount

of resources (in terms of the test generation time and the number of generated test cases), and yet they failed to find the bugs (in contrast to SE technique). The reason might be that Random testing blindly generates test sequences and the Simple Exploration technique does not evaluate guards or action code. So in both techniques, some test cases might not be feasible. Moreover, these techniques generate random values for parameters, which does not work well for numerical parameters that range over a wide range of values. Having said that, these techniques are much simpler to implement and work much faster compared to SE techniques, which are appropriate for systems with fewer decision predicates over numerical input parameters. Therefore, as our future work, we may work on a hybrid technique to benefit from the strengths of the three techniques.

5 Related Work and Discussion

There are various approaches and tools for testing UML profiles, (e.g., SysML [1]) and for generating tests from UML state machines [19,23,30] based on different coverage criteria. Compared to ours, none of these approaches direct the test case generation and are not suitable for RTE systems. TTCN [33] and approaches based on it [12] are used to simulate and test the communication between systems, as opposed to our approach that makes an infrastructure around components of an RTE system to make the component testable and to test the system at run-time. In Model-Based Testing (MBT) techniques [26,32] abstract and concrete tests are typically generated from the model and the implementation respectively, which requires ensuring the consistency between the two. Our approach, however, does not require to complement abstract test cases with concrete counterparts and create and maintain the mapping between them. Other approaches more closely related to ours include TGV [15], which is a black box conformance testing technique and uses a test purpose for test selection. STG [7], introduced by the same authors, is more efficient, which generates test data symbolically. However, in STG a synchronized product of test purpose and the specification are sent to the symbolic execution tool (the product is larger than the original specification), which can end up into path explosion [6]. In our approach, we reduce the size of the symbolic execution input to make the symbolic execution a relatively lighter task. Moreover, TGV does not support timers for scheduling purposes. The same limitation exists in tools and techniques such as [14,21] where the user specifies the SUT as a basic statechart, and a simple LTL property is used to direct the test case generation. Besides, in this work, the property and the SUT are formalized in two languages, which forces the modeler to know both LTL and the modeling language notations. In our approach, properties and SUT are both specified using statecharts that are more intuitive and straightforward for modeler compared to LTL properties. Simulink [16] uses its Design Verifier component to specify temporal properties in state machines, but Simulink [16] is more suitable for automatic control and digital signal processing than for modeling real-time applications. Drusinsky [9,10] specifies temporal properties using statecharts (calls them assertions) and uses a commercial code

generator to generate JUnit test cases with random parameters to test the assertions. In our approach, we test the whole system at run-time by generating code from the system as well as the properties, and we support SE as an effective technique for generating input parameters.

6 Conclusion

In this paper, we proposed a framework for automatically testing components of UML-RT models w.r.t. formally specified properties. Compared to existing model-based techniques where abstract test cases are complemented with their concrete counterparts, our approach solely leverages on constructs provided by the modeling language to express all artifacts (component to test, test harness, the property of interest) and the existing code generator to generate code for the artifacts. The generated code is executed to test the system at run-time. Based on our evaluations, our approach was able to find some bugs on two UML-RT models. Our approach has some limitations. The Symbolic Execution engine that we are using does not support complex data types and only supports a very limited subset of the UML-RT action languages. The mentioned limitations are not applicable to Random and Simple Exploration techniques, however, these techniques need more work to generate more accurate parameter values. We intend to address these shortcomings in future work. We may work on a hybrid technique to benefit from the strengths of the three techniques.

References

1. Testing solutions with UML/SysML (2010). http://www.artist-embedded.org/docs/Events/2010/UML_AADL/slides/Session1_Matthew_Hause.pdf. Accessed 06 Jan 2018
2. Model-Based Systems Engineering Design of an Automobile Collision Avoidance System (2011). https://www.isr.umd.edu/~austin/enes489p/projects2011a/CollisionAvoidance-FinalReport.pdf. Accessed 06 Jan 2018
3. A collection of well-known software failures (2016). http://www.cse.psu.edu/~gxt29/bug/softwarebug.html. Accessed 25 Aug 2016
4. Androutsopoulos, K., Clark, D., Harman, M., Hierons, R.M., Li, Z., Tratt, L.: Amorphous slicing of extended finite state machines. IEEE Trans. Softw. Eng. **39**(7), 892–909 (2013)
5. Bordeleau, F., Fiallos, E.: Model-based engineering: a new era based on Papyrus and open source tooling. In: OSS4MDE@ MoDELS, pp. 2–8 (2014)
6. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
7. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: STG: a symbolic test generation tool. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 470–475. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_34
8. Colby, C., Godefroid, P., Jagadeesan, L.J.: Automatically closing open reactive programs. In: ACM SIGPLAN Notices, vol. 33, pp. 345–357. ACM (1998)

9. Drusinsky, D.: Modeling and Verification Using UML Statecharts: a Working Guide to Reactive System Design, Runtime Monitoring and Execution-Based Model Checking (2011)
10. Drusinsky, D.: Practical UML-Based Specification, Validation, and Verification of Mission-Critical Software: Space Exploration and Defense Software Examples in Practice. Dog Ear Publishing, Indianapolis (2011)
11. Feilkas, M., Fleischmann, A., Pfaller, C., Spichkova, M., Trachtenherz, D., et al.: A top-down methodology for the development of automotive software (2009)
12. Grossmann, J., Serbanescu, D., Schieferdecker, I.: Testing embedded real time systems with TTCN-3. In: International Conference on Software Testing Verification and Validation, ICST 2009, pp. 81–90. IEEE (2009)
13. Hessel, A., Larsen, K., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Formal Methods and Testing, pp. 77–117 (2008)
14. Jagadeesan, L.J., Porter, A., Puchol, C., Ramming, J.C., Votta, L.G.: Specification-based testing of reactive software: tools and experiments: experience report. In: Proceedings of the 19th International Conference on Software Engineering, pp. 525–535. ACM (1997)
15. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. Int. J. Softw. Tools Technol. Transf. (STTT) **7**(4), 297–315 (2005)
16. Joshi, A., Heimdahl, M.P.E.: Model-based safety analysis of simulink models using SCADE design verifier. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, pp. 122–135. Springer, Heidelberg (2005). https://doi.org/10.1007/11563228_10
17. Dominguez, A.L.J.: Detection of feature interactions in automotive active safety features. Ph.D. thesis, University of Waterloo (2012)
18. Jussien, N., Rochart, G., Lorca, X.: Choco: an open source java constraint programming library. In: CPAIOR 2008 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP 2008), pp. 1–10 (2008)
19. Kim, Y.G., Hong, H.S., Bae, D.-H., Cha, S.D.: Test cases generation from UML state diagrams. IEE Proc.-Softw. **146**(4), 187–192 (1999)
20. Korel, B., Singh, I., Tahat, L., Vaysburg, B.: Slicing of state-based models. In: Proceedings of the International Conference on Software Maintenance, ICSM 2003, pp. 34–43. IEEE (2003)
21. Li, S., Wang, J., Qi, Z.-C.: Property-oriented test generation from UML statecharts. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 122–131. IEEE Computer Society (2004)
22. Mealy, G.H.: A method for synthesizing sequential circuits. Bell Labs Tech. J. **34**(5), 1045–1079 (1955)
23. Offutt, J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. Softw. Test. Verification Reliab. **13**(1), 25–53 (2003)
24. Posse, E., Dingel, J.: An executable formal semantics for UML-RT. Softw. Syst. Model. **15**(1), 179–217 (2016)
25. Pretschner, A., Philipps, J.: 10 methodological issues in model-based testing. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 281–291. Springer, Heidelberg (2005). https://doi.org/10.1007/11498490_13
26. Pretschner, A., Philipps, J.: 10 methodological issues in model-based testing. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 11–18. Springer, Heidelberg (2005). https://doi.org/10.1007/11498490_13

27. Rapos, E.J., Dingel, J.: Incremental test case generation for UML-RT models using symbolic execution. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 962–963. IEEE (2012)
28. Selic, B.: Using UML for modeling complex real-time systems. In: Mueller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 250–260. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0057795
29. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling, vol. 2. Wiley, New York (1994)
30. Generating tests from UML specifications: generating tests from UML specifications. The Unified Modeling Language, p. 76 (1999)
31. Tkachuk, O.: Domain-Specific Environment Generation for Modular Software Model Checking. Kansas State University, Manhattan (2008)
32. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verification Reliab. **22**(5), 297–312 (2012)
33. Willcock, C., Dei, T., Tobies, S., Keil, S., Engler, F., Schulz, S.: An Introduction to TTCN-3, vol. 2. Wiley, Hoboken (2005)
34. Zurowska, K., Dingel, J.: Symbolic execution of UML-RT state machines. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, pp. 1292–1299. ACM (2012)

MAPLE: An Integrated Environment for Process Modelling and Enactment for NFV Systems

Sadaf Mustafiz[1(✉)], Guillaume Dupont[1], Ferhat Khendek[1], and Maria Toeroe[2]

[1] ECE, Concordia University, Montreal, QC, Canada
{sadaf.mustafiz,ferhat.khendek}@concordia.ca, gdupont@encs.concordia.ca
[2] Ericsson Inc., Montreal, QC, Canada
maria.toeroe@ericsson.com

Abstract. The Network Functions Virtualization (NFV) paradigm is making way for the rapid provisioning of network services (NS). Defining a process for the design, deployment, and management of network services and automating it is therefore highly desirable and beneficial for NFV systems. The use of model-driven orchestration means has been recently advocated in this context. As part of this effort, we propose a process enactment approach with NFV systems as the target domain. We provide support for automated process execution with a megamodel-based enactment approach. An integrated process modelling and enactment environment, MAPLE, has been built into Papyrus for this purpose. Process modelling is carried out with UML activity diagrams. The enactment environment transforms the process model to a model transformation chain, and then orchestrates it with the use of megamodels. We demonstrate our environment by enacting a NS design process.

1 Introduction

Automating the end-to-end management of network services (NS), in other words, enacting the workflow or process for network service management without manual intervention is highly desirable in the NFV domain and remains a major challenge for network operators and service providers [1,2]. The European Telecom Standards Institute (ETSI) has very recently launched a zero-touch network and service management group. As stated in [3], the challenges of 5G will trigger the need for a radical change in the way networks and services are managed and orchestrated.

We believe the application of model-driven engineering (MDE) methods and tools is essential to further such developments in the NFV domain [4]. MDE advocates the use of models as first class citizens in the engineering process. The models are manipulated with model transformations which form the backbone for automation in MDE. ETSI has recently released an information model for NFV [5]. Leveraging these models can substantially benefit the NFV systems

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 164–178, 2018.
https://doi.org/10.1007/978-3-319-92997-2_11

by reducing their development and management efforts. Moreover, explicit modelling of the process not only allows for the automation of the NS management process but also paves the way for streamlining or optimizing the process to ultimately speed up deployment time. Such a process model (PM) can potentially be mapped to model transformation chains hence enabling NS management and orchestration via model-driven process enactment [6–8].

Previously, we have proposed a model-based process for NS design and deployment [9]. The proposed workflow is compliant with the NFV reference framework, and is a first step towards the necessary automation of the NS design and deployment process for NFV systems. We followed up the work in [10] by elaborating on a method for NS design and proposing an initial approach for enacting the NS design, deployment and management process. In this paper, we focus on the enactment support and present an integrated process enactment environment for NFV systems. MAPLE (**MA**GIC **P**rocess Mode**l**ling and **E**nactment Environment) provides support for model management with the use of megamodels. We demonstrate the use of MAPLE on the NS design process, which represents a portion of the NS management process.

We adapt the Papyrus [11] environment to provide tool support for process enactment. Papyrus is the tool of choice of ETSI NFV. It is an open-source Eclipse-based UML 2 modelling environment which mainly provides a graphical editor for creating UML 2 compliant diagrams, but also includes extensive support for SysML, UML-RT as well as other UML-based domain-specific languages. One of the core ideas of Papyrus is that it is completely customizable: appearance, diagrams, palettes, etc. Everything (in theory) can be tweaked, allowing one to use this tool as a base for building custom environments fairly easily. Nowadays, Papyrus is widely used in the industry, and it is thus far more interesting and useful to develop plug-ins for this platform than to create standalone, isolated programs.

This paper is structured as follows: Sect. 2 gives a brief background on megamodelling and process modelling. Section 3 discusses the main functionalities of our enactment environment, and presents its architecture. Section 4 presents our NFV case study, and demonstrates the use of the environment on the NS design process. Section 5 discusses some related work. Finally, Sect. 6 concludes with some future work.

2 Background

This section provides a brief background on some of the underlying concepts, namely megamodels and process models.

Megamodels: Model management approaches typically use megamodels which provide structures to avoid the so-called ‘meta-muddle’ [12]. A megamodel contains artifacts (which are models), relations between them (which may be transformations), and other relevant metadata. A megamodel can be seen as a map to find and link together all involved models. It can be used to enforce conformance

and compatibility checks between the various models and transformations. It is also useful for reusing and composing transformations in transformation chains.

Process Models: In our work, we use UML 2.0 Activity Diagrams [13] to represent and visualize a process model. Activity Diagrams are typically used to model software and business processes. These allow the modelling of concurrent processes and their synchronization with the use of fork and join nodes. Both control-flow and object-flow can be depicted in the model. An activity node can either be a simple *action* (representing a single step within an activity) or an *activity* (representing a decomposable activity which embeds actions or other activities). An activity specifies a behaviour that may be reused, i.e., an activity can be included in other activity diagrams to invoke behaviour. Along with the activities, the input and output models associated with each activity are also clearly identified via input and output parameter nodes (denoted by the rectangles on the activity border). Since UML 2.0 Activity Diagrams are given semantics in terms of Petri Nets [13], the precise formal semantics allow the activity diagrams to be simulated and analyzed.

3 Process Enactment Environment

In our approach, process enactment is carried out with the use of transformation chain orchestration in combination with model management means. Transformation chaining is the preferred technique for modelling the orchestration of different model transformations [14]. Orchestration languages are used for the composition of the transformations in order to model the chain as sequential steps of transformations. Complex chains can incorporate conditional branches and loops, and also can model composite chains (a chain including other transformation chains). Figure 1 gives an overview of our enactment approach.

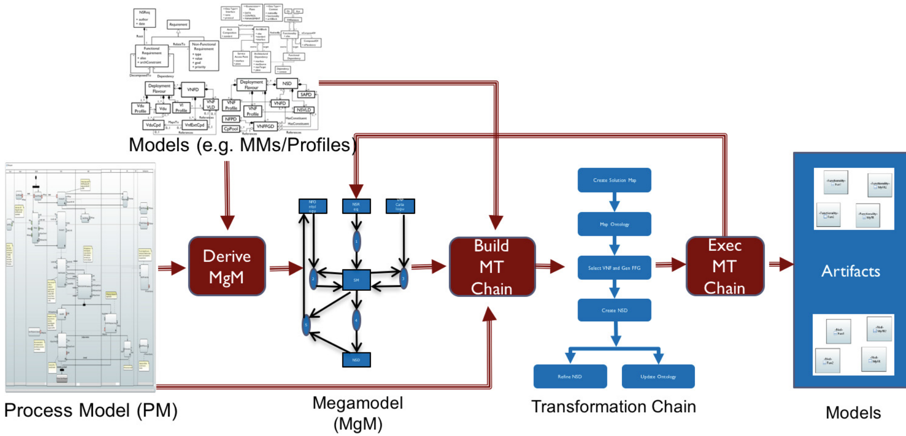


Fig. 1. Process enactment approach

3.1 Functionalities

We elaborate here on the main functionalities provided by MAPLE.

Creating the Process Model (PM). As mentioned earlier, in our work we use UML Activity Diagrams to represent and visualize Process Models. It is also possible for such processes to be modelled with some other workflow modelling language, for instance BPMN [15].

The Eclipse Papyrus Activity Diagram environment is used to create a PM. The PM instance conforms to the Activity Diagram language. Each PM includes a set of activities or actions. In our work, the behaviour of each of these activities is implemented with a model transformation. For our purpose, we need to associate these actions with the model transformations which implements them. We use the attributes of the activity nodes in the diagram to add information about the associated transformations.

Each action in the PM is also associated with a set of input and output models. Papyrus requires all metamodels to be mapped to profiles to allow model instances to be created and to be used as source or target models of the transformations. As per the ETSI NFV modelling guidelines, our models also comply with the NFV Papyrus *OpenModelProfile* [16].

Deriving the Megamodel (MgM). One of the main issues we had to address in this work is related to resource management: how can we centralize information about resources we need so that we can easily access them? The problem induced by this question is actually not trivial: a transformation involves several metamodels that can be expressed using heterogeneous technologies. The transformation itself can be considered as a model conforming to a specific metamodel, for instance ATL [17], QVT [18], Epsilon [19], etc. Besides, transformations define a very precise configuration as to what must be fed in and pulled out, something we need to be aware of whenever we run the transformations.

We use megamodels for model management as part of MAPLE. The MgM is derived in two steps: (1) by registering the resources, and then (2) by registering the PM. *Registering the resources:* To begin with, the resources which are part of the project (metamodels/profiles) are registered in the MgM. This is carried out automatically by going through the project workspace (referred to as *workspace discovery*), and an initial MgM is derived at this stage.

Registering the PM: Following the workspace discovery, the MgM is incrementally built by carrying out a *PM discovery.* This step involves registering the PM and the associated model transformations in the MgM. The PM needs to be linked with the elements of the MgM so that we can reach, when needed, every information relevant to enact it. However, we do not want any constraint on the shape of the process model, effectively decoupling its metamodel as much as possible from the megamodel one. Therefore, the MgM should be ideally independent from the PM, since it is meant for keeping track of resources. We do not want to (and actually cannot) refer to the MgM in the PM, but we still need

a link between these two entities. This link is created by weaving the PM and the MgM, and storing the details in a *weave model*. A weave model is a special kind of model that defines relationships between the objects and relations of other distinct models (at least two) [20]. The weave model binds every relevant element of the PM to their corresponding resources in the MgM, without touching the structure of either of them. The weave model is dependent on the PM. In other words, the weave metamodel is always specific to the PM language it weaves. Thus, if we had to adapt the environment for another PM language (e.g., expressed with BPMN), it would be necessary to create a new weaver to bind the new type of PM with the MgM.

Building the Transformation Chain. The PM is given translational semantics by mapping it to a transformation chain. The chain is in essence a schedule with the required details (sequence of actions, transformations used, inputs and outputs of the transformations). This allows us to build a generic enacter, instead of having an enacter for each kind of PM. Having a generic enacter also leaves scope for integrating other formalisms for modelling the PM.

The translation from a PM to a transformation chain is implemented as an ATL model transformation, which takes as input various data (the PM, the weave model, the MgM and if applicable, additional environment information) and yields the corresponding transformation chain.

Executing the Transformation Chain. Once the transformation chain is created, we need to be able to execute the chain in order to enact the PM. For this purpose, we developed an *enacter*, which is simply a program that can execute the correct actions in the right order, based on a schedule, namely the transformation chain model.

Similar to UML 2 Activity Diagrams, the generated chain is also given token-based semantics. Therefore, the enacter developed is based on controlling the tokens and activating the actions when needed.

3.2 User Interface

One of the major constraints in the development of MAPLE was that it must be based on Eclipse so that it can be integrated in Papyrus. As the MgM is a complex structure, we decided to only allow it to be manipulated through an interface (a user interface for the user but also an API for the developer) to avoid corrupting it with incorrect additions. It is possible to create and maintain more than one MgM for a given project.

Our main goal was to have an extensible tool, which would allow any technology, transformation type/engine, and PM language to be adapted and accommodated at a later time. The fact that we must base ourself on Eclipse puts a lot of constraints on the user interface the tool can provide. Eclipse is primarily

an editor, and megamodels should not be intruding its overall workflow (especially given the fact that they will not be used all the time). For this reason, the interface of the tool was provided via global and contextual menus.

Registering resources is carried out via a dialog box which allows the user to select the MgM in which to register the resource(s) and to select the discoverer (workspace or PM) (see Fig. 2). The user request to begin enactment of a process model is carried out via Eclipse's launch configuration system (see Fig. 3). A PM launch configuration is characterized by a MgM, a PM and a specific translator (provided through an extension point, see Sect. 3.3).

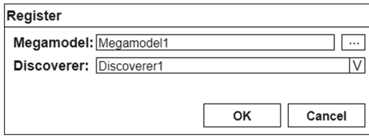


Fig. 2. The *Register* dialog

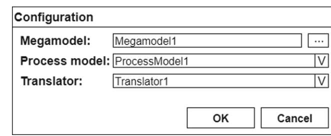


Fig. 3. Launch configuration UI

3.3 Architecture

The backend architecture of the enactment extension developed for Papyrus (Neon Release) is shown in Fig. 4. In the figure, the core functionalities are represented using rountangles (rounded-corner rectangles). The rectangles are extensions, that is modules without which the system can work but can be recognized by the system for adding new functionalities.

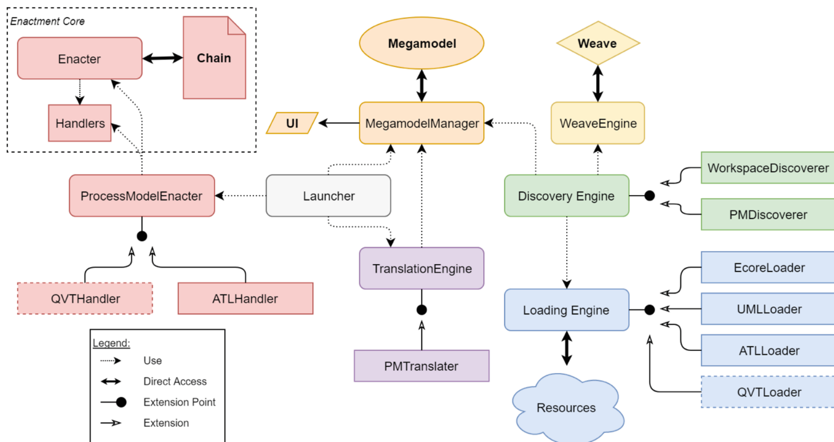


Fig. 4. Backend architecture (Color figure online)

Model Loaders. This part corresponds to the blue boxes in Fig. 4 and is centered around the *loading engine*. The goal of this subsystem is to provide a high level interface with the actual resources of the system (i.e., the file system). It is able to recognize the correct way of loading a file and to extract data from it that could be interesting to have in the megamodel, for example. Typically, when trying to register a file in the MgM, the discovery engine asks the loading engine to load it so that it can extract data. Defining *loaders* (such as, `EcoreLoader`, `UMLLoader`) allows different technologies to be incorporated in the global system, making it able to understand new formats.

Registering Resources. This part corresponds to the green boxes in Fig. 4 and is centered around the *discovery engine*. The *discovery* is the process by which a resource is added to the MgM. It includes the recognition of the resource (determining that it is a metamodel, a transformation, a profile, etc.) as well as the extraction of its data (URI, name, inputs and outputs, etc.). This is also the process that can *weave* the PM to the MgM, with the help of the *weave engine*.

As in the case of the loaders, it is possible to develop a custom *discoverer* (for instance, as shown in Fig. 4, `WorkspaceDiscoverer` or `PMDiscoverer`). This allows the tool to be extended with discoverers for other kinds of workflow modelling languages. As the weave is entirely dependent on the PM language, it should be noted that the weaving process is completely devolved to the discoverer, which must provide both a specific weaver (an implementation) as well as fill it. In our case, the weave model conforms to a variant of the UML Activity Diagram formalism.

Megamodel Management. This part corresponds to the orange part in Fig. 4 and is centered around the *megamodel manager*. This is the part that allows the user to access the MgM: request to register a resource, create or delete an MgM, etc. It also includes an extensive API for manipulating the MgM, ensuring its validity throughout the process.

Translation. This part corresponds to the purple part in Fig. 4 and revolves around the *translation engine*. The goal of this part is to carry out the translation of a PM to a transformation chain that can be enacted. It exposes an extension point that allows anyone to plug his own translation means, which is required if ever we want to use another type of PM.

Scheduling and Enactment. This part corresponds to the red part in Fig. 4. It is composed of two subparts: a generic enacter (`Enacter`), independent from the project (inside the dashed box labelled "Enactment Core"), and an interface between this enacter and the remaining part of the project, through the `ProcessModelEnacter` part.

It should be noted that the `ProcessModelEnacter` is not an independent enacter, but a layer on top of the generic enacter, providing it with specific behaviour as to what to do with each action; that is, actually executing the transformations they correspond to, using interfaces with different transformation engines defined as extensions (e.g.: `ATLHandler`, `QVTHandler`, `JavaHandler`).

Launcher. This part corresponds to the gray rectangle in Fig. 4. This is what orchestrates everything needed to execute the PM, which includes translating to a model transformation chain and enacting the chain. Enactment configurations (data associated with the PM to translate and enact it) are stored as a standard Eclipse launch configuration, which are also managed by this part.

4 NFV Case Study

We have used MAPLE to model and enact part of the NS management PM proposed in [9]. In this section, we demonstrate the enactment of the NS design process which is one of the activities of the NS management process.

4.1 Network Service Design

An *NS*, such as VoIP, is a composition of Network Function(s) (NF) and/or other NSs, interconnected with one or more *Forwarding Graphs (FG)*. These graphs interconnect the NFs and describe the traffic flow between them. *Virtualized Network Functions (VNF)* are the building blocks of an NS in NFV. VNFs are software pieces that have the same functionality as their corresponding *Physical Network Functions (PNF)*, e.g., a virtual firewall (vFW) vs. a traditional firewall device. The design of an NS consists of defining an *NS Descriptor (NSD)*, a deployment template which captures all this information. This template is provided to the NFV Orchestrator for the NS lifecycle management.

Coming up with the deployment template for an NS is not an easy task for an inexperienced tenant who has limited knowledge regarding the details of the target NS. Instead of these details, the tenant may request at some level of abstraction the functional and non-functional characteristics of the targeted NS. The gap between these NS requirements (*NSReq*) and the NS deployment template is filled with an automated NS design method. With the help of a network function ontology (*NFOntology*), it is indeed possible to fill this gap and design automatically *NSDs* from *NSReqs*. The NF ontology captures the known NF/service decompositions and their (standard) architectures. In the NS design method, the *NSReq* decomposition is guided by the *NFOntology* to a level where proper network functions can be selected from an existing VNF catalogue. After the selection of the VNFs, the method continues with the design of the forwarding graphs given the characteristics of the selected VNFs and their dependencies. This generates a set of forwarding graphs, which are refined further based on the non-functional requirements in the *NSReq*, resulting in the target *NSD*. As a final step, the NF ontology is enriched with the new decompositions. It is also possible to enrich the NF ontology with new standards and new services. Please note that the goal of this paper is not to describe the details of the NS design method but to show how the process is enacted using our tool. The NS design process and the associated modelling languages which are part of the NS Design PM are described in details in [10]. A revised version of the PM is shown in Fig. 5.

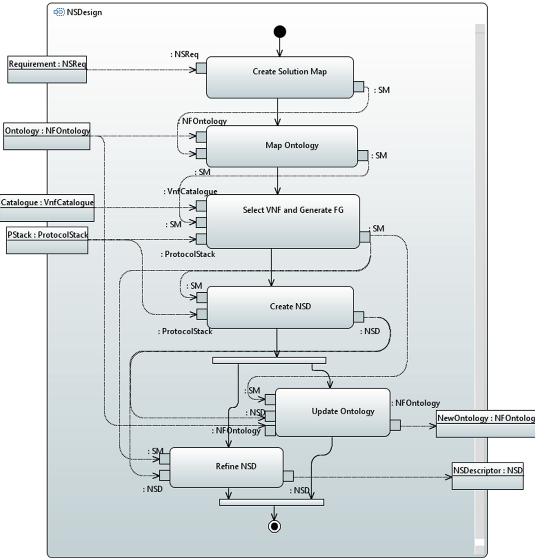


Fig. 5. NS design PM [10]

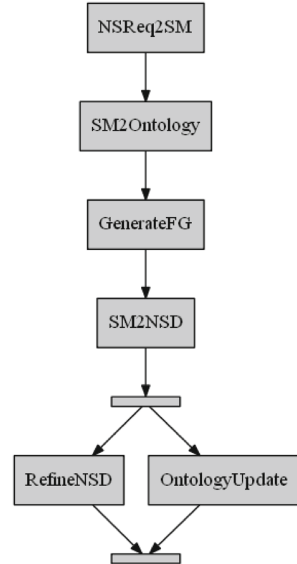


Fig. 6. NS design chain

4.2 Using MAPLE for NS Design

We begin by registering the NS design resources (profiles for `NSReq`, `NFOntology`, etc.) that are used in the process by creating an initial MgM (see graphical view of the MgM in Fig. 7). This MgM includes the meta-metamodels (UML and Ecore pre-loaded in the base MgM) and conformance links. In the MgM, the metamodels are represented in orange, UML profiles in green, transformation models in brown, and other models (PM, weave model) in gray. The dashed links represent conformance relationship, and the solid black links represent object flow.

The next step is to register the PM which automatically refines the MgM based on information available in the PM. Each activity in the NS Design PM is implemented as an ATL transformation. Each associated transformation is stored as an attribute of the corresponding activity node. The PM itself is also added as a resource (see Fig. 8). While discovering and registering the PM, whenever an object flow links two pins and that flow and pins do not have any assigned name, MAPLE can detect it and create an intermediate model. This step resulted in creating an initial repository of models, NFV-specific languages, and tools along with the relationships between the artifacts.

When the process is requested to be enacted, the NS design PM is mapped to a transformation chain (see Fig. 6). Orchestration of the chain is carried out with the use of the embedded orchestration engine. The process execution begins by taking `NSReq` models as inputs and creating an intermediate model, `SolutionMap`, which is incrementally refined. Once the initial `NSD` is created, MAPLE enables NSD refinement and ontology enrichment to be carried out

concurrently since they are independent of each other, hence optimizing deployment time. The enactment ends with the generation of the target models, `NSD` and `NFOntology` (not shown here for space reasons). With this environment, NFV users with limited modelling expertise and minimal knowledge about the underlying ATL transformations can generate a target NSD with basically a few clicks. The configurations for ATL, making sure that the correct models are passed into each transformation, do not need to be handled by the user. The same PM can be enacted again with different inputs if desired, and it can also be reused as part of another PM if required. A demo video of the enactment environment is available at https://users.encs.concordia.ca/~magic/maple-demo.php.

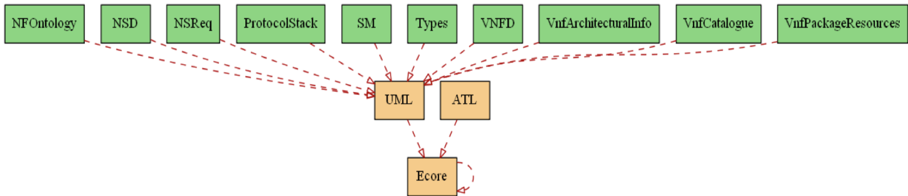


Fig. 7. NS design megamodel - initial version (Color figure online)

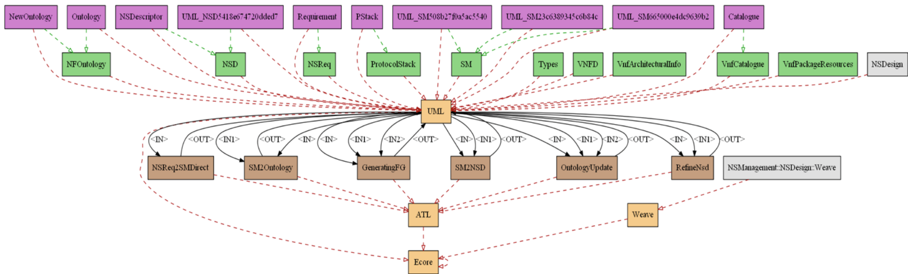


Fig. 8. NS design megamodel - complete version

This work sets the basis for the enactment of the entire NS design, deployment and management process. Each activity in the NS lifecycle involves a complex chain of tasks. We are working on modelling the behaviour of the other activities, e.g. *NS Instantiation* and *VNF Instantiation*. The entire PM can then be mapped on to a composite chain of transformations along with an extended MgM to allow for automated deployment and management of network services. When the NS management PM is enacted using MAPLE, the resulting MgM will prove to be a very useful repository for NFV projects in the industry and academia.

4.3 Discussion

In the NFV community, application of MDE is still at the initial stages where information modelling with class diagrams and state machines are more typical. Advanced applications of MDE in this domain is minimal. This work is meant to set the pillars for a project which can advance greatly with the appropriate use of MDE, as well as provide much-needed inspiration to the open-source NFV industry projects to embrace MDE methods and technology.

The proposed enactment support, MAPLE, has been built into a mainstream, open-source, industry standard tool which is also NFV's tool of choice - Eclipse Papyrus. MAPLE requires the PM to be created using UML Activity Diagrams in Papyrus. Once the PM is created, enacting it is fairly intuitive using our environment and as elaborated earlier in this section, only requires a few steps on the user end. For the process to be enacted, the actions which are part of an activity need to be implemented. This can be done using model transformations (implemented with ATL or any general-purpose language) or by defining executables (Java, C, etc.) for the actions. The latter allows legacy code to be integrated and used within the PM.

MAPLE is easily extensible, and we plan on providing support for other transformation languages including Epsilon and QvT. The tool can also be extended to support other workflow modelling languages. One of the main contributions is the underlying support for model management. The megamodel forms a repository of models, metamodels, and tools for the NFV domain, which can be of great use in this domain. The generated models are also persisted and added back to the repository to be inspected, compared, or used in transformations at a later time. The MgM makes it possible to explicitly deal with dependencies between models. It is used to carry out type checks between the metamodels of the models to be transformed and to check the compatibility between the metamodels and the transformations by querying the MgM. In case of activities which include a heterogeneous set of transformations, the MgM determines which transformation engine should execute the transformation.

In addition to being useful for carrying out conformance and compatability checks, the created MgM can be used for advanced traceability support. The metamodel we have defined for megamodels already includes traceability-related elements. Work is in progress to extend the tool for this purpose. Such a feature would indeed be beneficial for NFV systems. For instance, it would then be possible to retain trace links all the way from the generated NSD back to the original NS requirements and the ontology. With the MgM, trace info can also be maintained for multiple executions of the PM. Such information can be useful for data analytics which is an important component of any NFV system.

5 Related Work

We have covered the state of the art with regards to process modelling in the NFV domain in [9]. Process enactment is a widely adopted method in the business process modelling domain. The model-based methods and tools in this

domain mostly support SPEM/BPMN processes and use BPEL for orchestration [21,22]. UML Activity Diagrams together with MDE enablers, such as model transformations, are not widely used in this area for modelling and enacting processes. However, model-driven orchestration has been recently promoted for NFV, and viewed as a more robust method than business process workflows for NFV orchestration [4]. BPMN-like workflows are in general implementations of specific task-oriented cases which are appropriate for immutable business processes as stated in [4]. In software defined environments which evolve rapidly, such workflows bring about difficulties and risks.

A few model-based continuous integration and deployment methods and tools have been proposed with cloud as the target domain [23,24], which use domain-specific languages to model the deployment of cloud applications and provide means to adapt the models for use in the runtime environment. These approaches do not support process modelling. While model-based approaches exist in the NFV domain [2,25], the application of advanced MDE techniques, such as model management and transformation chain orchestration, is minimal for NFV systems. With this work, we aimed to show how MDE means can be used in the NFV domain to further the vision of automating end-to-end network service management.

There has been a lot of work on megamodelling [12,26–29], transformation chaining [6,7,30–32], and a combination of both [33–35] in the MDE community. The FTG+PM framework [35] which supports process execution based on an FTG (Formalism Transformation Graph - a subset of megamodels) is similar to our approach. However, the FTG and PM is defined as a single combined formalism, and so the FTG needs to be created together with the PM. There is no support for automated derivation of the MgM (which is a core part of our approach), and the research tool developed only supports execution of T-Core transformations. Tooling for model management (AM3) including automated megamodel discovery was developed as part of AMMA [26]. Tools supporting workflow or transformation chain orchestration [6,30,33,36] also existed at some point.

Our initial intention was to reuse and integrate existing components to build our environment. However to the best of our knowledge, no working tools to serve our purpose exist at the moment. We looked into Eclipse-based tools including MoDISCO/AM3, UNiTi, TraCo for megamodelling support and Wires*, MWE2 (Modelling Workflow Engine), ATLFlow for orchestration support. None of the tools with megamodelling support were usable (incompatible with Eclipse Papyrus, unavailable, or failed to work) at the time of this work. Orchestration engines available were not adequate for our needs, since we wanted to support concurrent executions of model transformations. With regards to the translation engine, Wires* could have been adapted for our purpose since it supported orchestration of ATL transformation chains derived from UML Activity Diagrams. However, the tool is no longer maintained and not available for use. For this reason, we developed our own translation engine and orchestration engine as well as the underlying model management support which allowed us to offer a flexible and extensible integrated environment for process modelling and enactment in Papyrus.

6 Conclusion

Automating NS management is one of the challenges in the NFV domain, and this is what we have aimed to address in this paper. This work resulted in a comprehensive and extensible environment, MAPLE, for model-driven process enactment. We have used the existing UML Activity Diagram environment in Papyrus and integrated process enactment means with it.

In our approach, we followed the model-driven paradigm all through. The core of the approach combines orchestration of model transformation chains with model management means. We begin with a process which is modelled as a UML Activity Diagram (referred to as the PM). The activities in the PM are associated with model transformations. Input and output objects are model instances of some existing domain-specific language. For model management, we build a megamodel (MgM) of the target system. This MgM contains information of all MDE resources that are being used by the process, as well as the link(s) between these resources. The PM itself is also a resource which is registered in the MgM. To enact the PM, the MgM is used along with the PM to build a model transformation (MT) chain. Token-based enactment means have been implemented to orchestrate the MT chain.

The enactment support has been created with NFV systems as the target domain, and was applied to the NS design process proposed in [10]. The approach along with the tool support is not restricted to NFV, and can be used in various domains for process enactment. Full traceability support is not yet available. However, the environment designed and implemented is not closed; it can be improved and, most of all, extended as required to meet NFV needs.

Acknowledgment. This work is partly funded by NSERC and Ericsson, and carried out within MAGIC, the NSERC/Ericsson Industrial Research Chair in Model Based Software Management.

References

1. Mijumbi, R., Serrat, J., Gorricho, J.L., Latre, S., Charalambides, M., Lopez, D.: Management and orchestration challenges in network functions virtualization. IEEE Commun. Mag. **54**(1), 98–105 (2016)
2. Chen, Y., Qin, Y., Lambe, M., Chu, W.: Realizing network function virtualization management and orchestration with model-based open architecture. In: 11th International Conference on Network and Service Management (CNSM 2015), pp. 410–418. IEEE (2015)
3. ETSI: Zero-touch Network and Service Management, December 2017
4. Berezin, A.: Utilizing Declarative Model-Driven TOSCA Orchestration for NFV. DZone, March 2017
5. ETSI: Network Functions Virtualisation (NFV) Release 2; Management and Orchestration; Report on NFV Information Model: ETSI GR NFV-IFA 015 V2.1.1, January 2017
6. Rivera, J.E., Ruiz-Gonzalez, D., Lopez-Romero, F., Bautista, J., Vallecillo, A.: Orchestrating ATL model transformations. In: MtATL 2009, pp. 34–46 (2009)

7. Etien, A., Aranega, V., Blanc, X., Paige, R.F.: Chaining model transformations. In: Analysis of Model Transformations Workshop, AMT 2012, pp. 9–14. ACM (2012)
8. Basciani, F., Di Ruscio, D., Iovino, L., Pierantonio, A.: Automated chaining of model transformations with incompatible metamodels. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) MODELS 2014. LNCS, vol. 8767, pp. 602–618. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11653-2_37
9. Mustafiz, S., Palma, F., Toeroe, M., Khendek, F.: A network service design and deployment process for NFV systems. In: 15th IEEE Network Computing and Applications, NCA 2016, pp. 131–139. IEEE Computer Society (2016)
10. Mustafiz, S., Nazarzadeoghaz, N., Dupont, G., Khendek, F., Toeroe, M.: A model-driven process enactment approach for network service design. In: Csöndes, T., Kovács, G., Réthy, G. (eds.) SDL 2017. LNCS, vol. 10567, pp. 99–118. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68015-6_7
11. Eclipse: Papyrus. https://eclipse.org/papyrus/
12. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003–2004. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005). https://doi.org/10.1007/11538097_3
13. Object Management Group: Unified Modeling Language (UML 2.5), March 2015
14. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, 1st edn. Morgan & Claypool Publishers, San Rafael (2012)
15. Object Management Group: Business Process Model and Notation (BPMN 2.0) (2011)
16. ETSI: Network Functions Virtualisation (NFV) Release 2; Information Modeling; Papyrus Guidelines: ETSI GR NFV-IFA 016 V2.1.1, March 2017
17. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MODELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006). https://doi.org/10.1007/11663430_14
18. Object Management Group: Meta Object Facility 2.0 Query/View/Transformation Specification (2011)
19. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69927-9_4
20. Del Fabro, M.D., Valduriez, P.: Semi-automatic model integration using matching transformations and weaving models. In: ACM Symposium on Applied Computing, SAC 2007, pp. 963–970. ACM, New York (2007)
21. Maciel, R.S.P., da Silva, B.C., Magalhes, A.P.F., Rosa, N.S.: An integrated approach for model driven process modeling and enactment. In: XXIII Brazilian Symposium on Software Engineering, pp. 104–114, October 2009
22. Aldazabal, A., Baily, T., Nanclares, F., Sadovykh, A., Hein, C., Ritter, T.: Automated model driven development processes. In: ECMDA workshop on Model Driven Tool and Process Integration, pp. 50–52 (2008)
23. Artač, M., Borovšak, T., Di Nitto, E., Guerriero, M., Tamburri, D.A.: Model-driven continuous deployment for quality DevOps. In: 2nd International Workshop on Quality-Aware DevOps, QUDOS 2016, pp. 40–41. ACM (2016)
24. Ferry, N., Song, H., Rossini, A., Chauvel, F., Solberg, A.: CloudMF: applying MDE to tame the complexity of managing multi-cloud applications. In: IEEE/ACM 7th International Conference on Utility and Cloud Computing, pp. 269–277, December 2014
25. OASIS: TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0. OASIS Committee Specification Draft 04, May 2017

26. Allilaire, F., Bézivin, J., Brunelière, H., Jouault, F.: Global model management in Eclipse GMT/AM3. In: Eclipse Technology eXchange Workshop (eTX) - A ECOOP 2006 Satellite Event, July 2006
27. Fritzsche, M., Brunelière, H., Vanhooff, B., Berbers, Y., Jouault, F., Gilani, W.: Applying megamodelling to model driven performance engineering. In: 16th IEEE Engineering of Computer Based Systems, ECBS 2009, pp. 244–253, April 2009
28. Sandro, A.D., Salay, R., Famelis, M., Kokaly, S., Chechik, M.: MMINT: a graphical tool for interactive model management. In: MoDELS 2015 Demo and Poster Session Co-located with ACM/IEEE MoDELS 2015. CEUR Workshop Proceedings, vol. 1554, pp. 16–19. CEUR-WS.org (2015)
29. Simmonds, J., Perovich, D., Bastarrica, M.C., Silvestre, L.: A megamodel for software process line modeling and evolution. In: Model Driven Engineering Languages and Systems (MODELS), pp. 406–415. IEEE (2015)
30. Oldevik, J.: Transformation composition modelling framework. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 108–114. Springer, Heidelberg (2005). https://doi.org/10.1007/11498094_10
31. Wagelaar, D.: Blackbox composition of model transformations using domain-specific modelling languages. In: 1st European Workshop on Composition of Model Transformations (CMT), pp. 15–19 (2006)
32. Heidenreich, F., Kopcsek, J., Assmann, U.: Safe composition of transformations. J. Object Technol. **10**(7), 1–20 (2011)
33. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: UniTI: a unified transformation infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_3
34. Fritzsche, M., Gilani, W.: Model transformation chains and model management for end-to-end performance decision support. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 345–363. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18023-1_9
35. Lúcio, L., Mustafiz, S., Denil, J., Vangheluwe, H., Jukss, M.: FTG+PM: an integrated framework for investigating model transformation chains. In: Khendek, F., Toeroe, M., Gherbi, A., Reed, R. (eds.) SDL 2013. LNCS, vol. 7916, pp. 182–202. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38911-5_11
36. Eclipse: Modeling Workflow Engine 2 (MWE 2). https://www.eclipse.org/Xtext/documentation/306_mwe2.html

Detecting Conflicts Between Data-Minimization and Security Requirements in Business Process Models

Qusai Ramadan[1(✉)], Daniel Strüber[1], Mattia Salnitri[2], Volker Riediger[1], and Jan Jürjens[1,3]

[1] University of Koblenz-Landau, Koblenz, Germany
{qramadan,strueber,riediger,juerjens}@uni-koblenz.de
[2] Politecnico di Milano, Milano, Italy
mattia.salnitri@polimi.it
[3] Fraunhofer-Institute for Software and Systems Engineering ISST, Dortmund, Germany

Abstract. Detecting conflicts between security and data-minimization requirements is a challenging task. Since such conflicts arise in the specific context of how the technical and organizational components of the target system interact with each other, their detection requires a thorough understanding of the underlying business processes. For example, a process may require anonymous execution for a task that writes data to a secure data storage, where the identity of the writer is needed for the purpose of accountability. To address this challenge, we propose an extension of the BPMN 2.0 business process modeling language to enable: (i) the specification of process-oriented data-minimization and security requirements, (ii) the detection of conflicts between these requirements based on a catalog of domain-independent anti-patterns. The considered security requirements were reused from SecBPMN2, a security-oriented extension of BPMN 2.0, while the data-minimization part is new. SecBPMN2 also provides a graphical query language called SecBPMN2-Q, which we extended to formulate our anti-patterns. We report on feasibility and usability of our approach based on a case study featuring a healthcare management system, and an experimental user study.

Keywords: Conflicts · Security · Data-minimization · BPMN

1 Introduction

Protecting the privacy of users has become a key activity in companies and governmental organizations. A important requirement for privacy is *data-minimization* [14,34], that is, to minimize "*the possibility to collect personal data about others*" and "*within the remaining possibilities, [to minimize] collecting personal data*" (Pfitzmann and Hansen in [28], p. 6). In addition to security concepts such as confidentiality, five data-minimization concepts, namely,

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 179–198, 2018.
https://doi.org/10.1007/978-3-319-92997-2_12

Anonymity, *Pseudonymity*, *Unlinkability*, *Undetectability* and *Unobservability*, are considered as fundamental for protecting the users' privacy. These concepts were first defined by Pfitzmann and Hansen [28] and later included in the ISO 15408 standard of common criteria for information technology security evaluation [16].

Although privacy-enhancing technologies [35] address specific data-minimization needs, privacy breaches often do not come from loopholes in the applied protection technologies, but from conflicting requirements on the business level of the target system [6,13–15,25]. For example, in healthcare, users have strong privacy concerns about how and for what purpose their health information is handled, which may interfere with an organization's documentation responsibilities for ensuring complete accountability. Various approaches have been proposed that deal with security and data-minimization requirements in a unified framework from the early stages of development [8,11,17,26]. However, these approaches focus on the identification of security and data-minimization requirements in the elicitation phase without analyzing trade-offs or detecting conflicts between them. The final output is usually a set of textual requirements. Relying on textually-specified security and data-minimization requirements to manually uncover conflicts between them is a difficult and error-prone task. The main reasons for that are:

First, conflicts between the data-minimization and security requirements depend on the context of how the technical and organizational components of the target system interact with each other. Specifically, conflicts not only result from trade-offs between requirements related to the same asset in the system (e.g., anonymous vs. accountable execution of a task), but also from those related to different assets. For example, a task may be required to be executed anonymously, while writing data to a secure data storage where the identity of the writer must be known for accountability reasons. The detection of such conflicts requires an understanding of the underlying business processes and their included interactions between security and data-minimization requirements. However, no existing approach supports the modeling of data-minimization requirements in business process models in the first place.

Second, a single data-minimization concept may have varied meanings based on *what* (which of the system assets) and from *who* (i.e., adversary type) to protect. These variations make it hard to decide whether two specific requirements are conflicting. For example, providing fully anonymous execution of a specific task hinders the ability of the system to keep the task's executor accountable, leading to a conflict. In contrast, providing *partial* anonymity by means of using pseudonyms is not conflicting with accountability. So far, there exists no approach to detect such conflicts between security and data-minimization requirements in the design of a concrete system.

To address these challenges, we propose an extension of the Business Process Modeling Language (BPMN, [1]), supporting: (i) the specification of process-oriented data-minimization and security requirements in BPMN models, (ii) the detection of conflicts between security and data-minimization requirements

based on a catalog of domain-independent anti-patterns. While the security annotations were reused from the security-oriented BPMN extension SecBPMN2 [33], our approach is the first to directly support modeling data-minimization requirements in BPMN models. It is also the first to support automatic conflict detection between specified security and data-minimization requirements in BPMN models. To express the anti-patterns, we extended SecBPMN2-Q, a graphical query language for BPMN models. We validate our approach using a case study based on a healthcare management system, and an experimental user evaluation.

The paper is organized as follows. Section 2 provides the necessary background. Section 3 introduces our BPMN extension. Section 4 presents the considered types of conflicts and our approach to detect them. Section 5 presents the tool support for our approach. Sections 6 and 7 are devoted to the validation based on a case study and user evaluation. Sections 8 and 9 discuss related work and conclude, respectively.

2 Background

We introduce the fundamental data-minimization concepts used in our work, and a BPMN-oriented security engineering approach whose security concepts we reused.

Data-Minimization Concepts. Pfitzmann and Hansen [28] define five data-minimization concepts that can be refined into privacy requirements for the target system [8,11,17,26]. (i) *Anonymity* is the inability of an adversary to sufficiently identify a subject within a set of subjects, called the anonymity set. (ii) *Pseudonymity* is a special case of anonymity where a pseudonym is used as an identifier for a data subject other than one of the data subject's personal identifiable information. (iii) *Unlinkability* is the inability of an adversary to sufficiently distinguish whether two items of interests (IOIs, e.g., subjects, messages, actions,...) within a system are related or not. (iv) *Undetectability* is the inability of an adversary to sufficiently distinguish whether an IOI is exist or not. By the definition [28], undetectability of an IOI can only hold against outsider adversary (i.e., neither being the system nor one of the participants in processing the IOI). (v) *Unobservability* is the undetectability of an IOI against all subjects uninvolved in it (i.e., outsider adversary) and the anonymity of the subject(s) involved in the IOI against other subject(s) involved in that IOI (i.e., insider adversaries).

Business Process Model-Based Security Engineering. [20] is a promising research direction in the field of security engineering. The key idea is to extend graphical business process modeling languages such as BPMN [1] to supports the modeling and analysis of procedural security requirements as early as during the design phase. Among various approaches [20], only the work proposed in [31] considers a data-minimization requirement, namely anonymity. However, further fundamental data-minimization requirements such as unlinkability and undetectability were not addressed yet.

To capture conflicts between security and data-minimization requirements, a unified framework for modeling both types of requirements is needed. Compared to other approaches, we found that the security concepts from SecBPMN2 [33] offer the following advantages: (i) In contrast to the work in [7,10,18,22,31,32,37] which support only a restricted set of security aspects, SecBPMN2 offers 10 security annotations, namely *accountability*, *auditability*, *authenticity*, *availability*, *confidentiality*, *non-repudiation*, *integrity*, *separation of duties*, *binding of duties*, and *non-delegation*. *Accountability* specifies that the system should hold the executors of the activities responsible for their actions. *Authenticity* imposes that the identity of a given activity's executor must be verified, or that it should be possible to prove a given data object as genuine, respectively. *Audibility* indicates that it should be possible to keep track of all actions performed by an executor or accessor of an activity, data object, or message flow. *Non-delegation* specifies that an activity shall be executed only by assigned users. *Binding of duties* and *Separation of duties* requires that the same person or different persons should be responsible for the completion of two related tasks, respectively. *Confidentiality* and *Integrity* indicate that only authorized users are allowed to read or modify data from a given activity, message flow, or data object, respectively. *Availability* indicates that it should be possible to ensure that an activity, data object, or message flow is available and operational when are required by authorized users.

Reusing these concepts allows us to study interactions between a comprehensive set of security and data-minimization requirements, enabling a powerful approach to conflict detection. (ii) While other works [27,36] use textual stereotypes to enrich business process models with security requirements (e.g., ≪confidentiality≫), SecBPMN2 represents security annotations using graphical icons [33]. The example in Fig. 2, explained in detail later, illustrates the specification of confidentiality, accountability, non-repudiation and binding-of-duty requirements in a BPMN model (icons in orange). Graphical annotations have the potential to increase the complexity of the resulting business process models less than textual ones would do [24], and as consequence, may contribute to the usability of our approach.

In addition, SecBPMN2 provides a query language for specifying queries that can be matched against a given SecBPMN model, called SecBPMN2-Q [33]. We reuse and extend this query language in our approach for specifying conflicts as anti-patterns.

3 Modeling Data-Minimization and Security Requirements

We propose a BPMN extension for specifying data-minimization and security requirements. Our support for data-minimization requirements is new, while the security-specific elements are reused from the security-oriented BPMN extension SecBPMN2. We first present a running example and then a complete description of our extension.

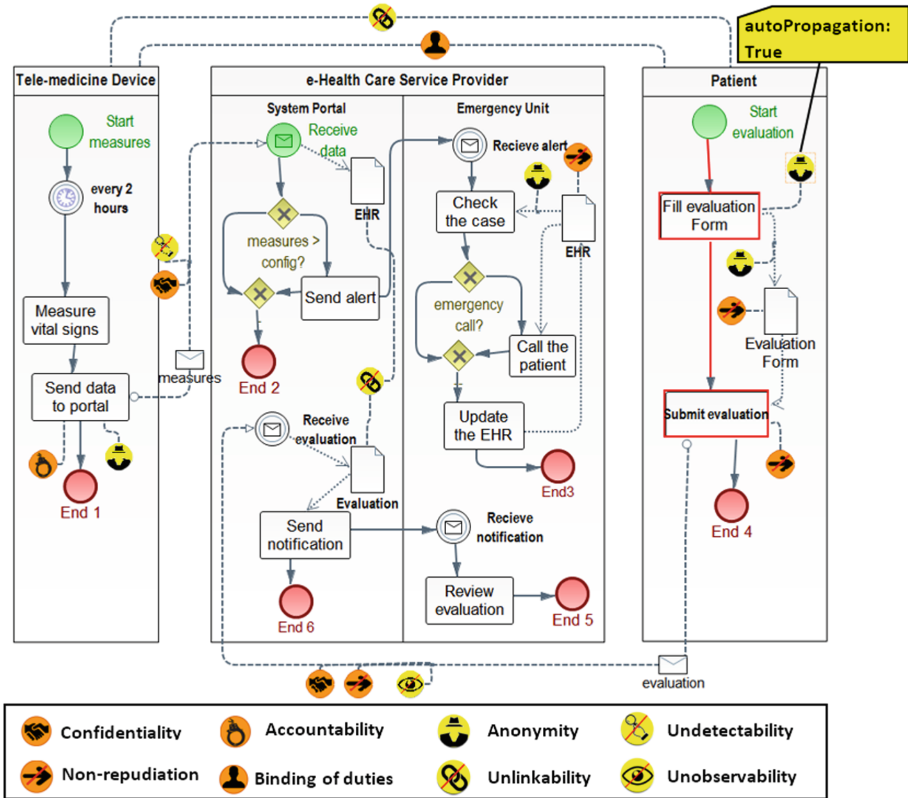


Fig. 1. Running example: Specifying data-minimization and security requirements in a healthcare business process. (Color figure online)

Running Example. Figure 1 represents a business process in the context of healthcare management. A patient makes use of a telemedicine device to receive an over-distance healthcare service. A patient can also evaluate the service through an online evaluation portal. Executors of a business process are represented by *pools* and *swimlanes* such as Tele-medicine Device and System Portal respectively. Communication between pools is represented by *message flows*; the content of such communications are *messages*: for example, Tele-medicine Device sends the message measures to System Portal. Atomic activities are represented with *tasks*, for example Send alert. *Data Objects* provide information about what activities require to be performed and/or what they produce, for example electronic healthcare record (EHR). A *data association* is a directional association used to model how data is written to or read from a data object. For instance, the Check the case task needs the EHR data object to be read. Events are represented with circles. *Start* events and *End* events mark the initial and terminal points. *Catch events* represent points in a business process where an event needs to happen, for example at this time. Gateways specify deviations of the execution

sequence: the gateway measures>=config? allows either the right or left branch to be executed.

Security concepts are represented with orange icons. *Confidentiality* is associated to message flows, meaning that the content of the message is to be preserved and not to be accessed by unauthorized users, respectively. *Accountability* is associated to Submit evaluation meaning that the task's executor must be monitored. Our new data-minimization concepts, discussed below, are represented with yellow icons.

Data-Minimization Annotations. To allow users to enrich business process models with data-minimization requirements, we extended BPMN's *artifact* class with four concrete data-minimization concepts namely, *anonymity*, *undetectability*, *unlinkability* and *unobservability*. The meta-model of these concepts is shown in Fig. 2: gray parts represent SecBPMN2 elements; white parts are new elements. Since an additional concept described by Pfitzman et al., *pseudonymity*, is a special case of anonymity, we use one annotation for both concepts. An attribute called *level* captures the required anonymity level (i.e., full anonymous vs pseudonymous). Using one annotation to represent related concepts is recommended to reduce graphical complexity [20]. A special type of association called *SecurityAssociation* is used to link security annotations with elements in the business process model. Additional details are captured using attributes and references, describing in particular the items of interests (IOIs) and adversary perspectives, as introduced in Sect. 2. In this section, we focus on the meta-model elements being relevant for conflict detection, leaving the discussion of others (in particular, the specification of *Mechanisms* and *DataSubjectRoles*) outside the scope of this paper.

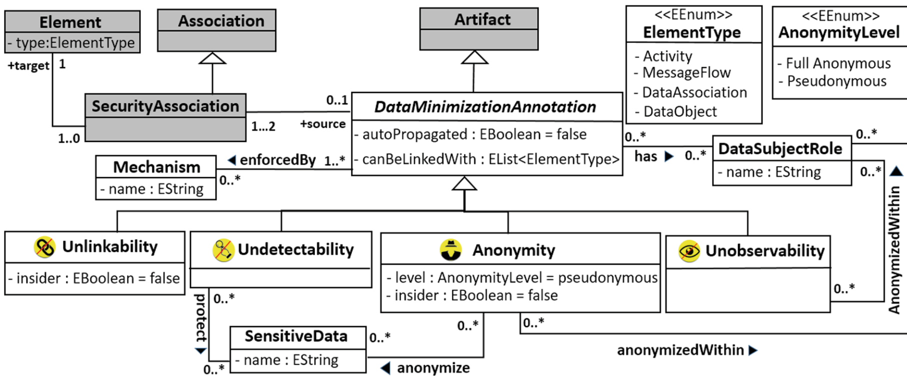


Fig. 2. Meta-model of our BPMN extension. Attributes show their default values. (Color figure online)

To reduce specification overhead, data-minimization annotations have an attribute *autoPropagated* which supports the propagation of the requirement to selected other elements in the model. Four cases are possible, depending on

the type of the element the annotation is linked with: (1) For an activity, the requirement is propagated to all following tasks in the same lane. (2) For a message flow, the requirement is propagated to all message flows that goes from the source-Pool of the considered messageFlow to its target Pool. (3) For a data input association, the requirement is propagated to all data input associations that read data from that data object in the same lane. (4) For a data output association, the requirement is propagated to all data output associations that write data to that data object in the same lane.

We designed the graphical syntax of the data-minimization annotations following Moody's guidelines for increasing the usability of modeling languages [24]. The data-minimization annotations share two common visual aspects with security annotations in SecBPMN2: they all have a solid texture, and a circular shape; they differ in their fill color, using yellow instead of orange. We believe that having different colors for security and data-minimization annotations contributes to usability.

In the rest of this section, all data-minimization annotations are defined. Each of them is defined in terms of one or more variants, one for every type of BPMN element it can be linked with. We mapped each annotation to a restricted list of element types to avoid overlapping meaning of different data-minimization annotations. For example, two messages cannot be linked to each other as related (i.e., unlinkability) if they are sent anonymously (i.e., anonymity). Therefore, having both unlinkability and anonymity annotations for message flows would be redundant.

Anonymity comes in four variants for the different BPMN elements it can be linked to: (i) *Anonymity-Activity* specifies that the executor of the task should be anonymous with respect to a given adversary perspective. (ii) *Anonymity-MessageFlow* specifies that the sender of the message should be anonymous with respect to a given adversary perspective. (iii) *Anonymity-DataOutputAssociation* specifies that the task should not write personal identifiable information to the data object. (iv) *Anonymity-DataInputAssociation* specifies that the task should only retrieve an anonymized variant of the data object.

The exact meaning of this annotation can be shaped by two attributes: The attribute *level* specifies the required anonymity level (i.e., fully anonymous or pseudonymous). In some scenarios, the system requires that the executor of an activity should be accountable, and thus, pseudonyms should be used to de-identify the executor of the activity. The attribute *insider* specifies against who to protect. The considered adversary type is either just outsider (*false*) or both outsider and insider (*true*). We define the outsider adversary as any entity being part of the surrounding of the system considered. The insider is any entity being part of the system considered, including the system itself.

The example model Fig. 1 shows three anonymity annotations associated to different BPMN elements. Consider, for example, the one associated to the Fill evaluation form activity. This annotation specifies that a patient shall be able to execute the Fill evaluation form task anonymously within the set of all patients without being identifiable by either outsider or insider adversaries. Since the

requirement is propagated, the same requirement applies to the Submit evaluation task.

Unlinkability comes in two variants, depending on which BPMN elements the annotation linked with. (i) *Unlinkability-Process* can be linked with two pools/lanes to specify that an adversary of the given type shall not be able to link two executed processes as related. In other words, if linked to two pools, this annotation imposes that a subject may make use of multiple services without allowing others to link these uses together as related [28]. (ii) *Unlinkability-DataObject* can be linked with two data objects to specify that, from the given adversary perspective, it should not be possible to link the two data objects as related. Since unlinkability can only be applied to two specific processes or data objects, it cannot be propagated to other elements. The attacker type is specified using the *insider* attribute, in the same way as in the *anonymity* case.

The example model includes two unlinkability annotations. Consider, for example, the unlinkability annotation associated with the two data objects namely, EHR and Evaluation. This annotation specifies that both outsider and insider adversaries must not be able to link an EHR and an Evaluation data objects as related.

Undetectability has three variants, depending on the BPMN elements it is linked with. (i) *Undetectability-Activity* specifies that an adversary should not be able to detect whether an activity is executed or not. (ii) *Undetectability-MessageFlow* specifies that an adversary cannot sufficiently distinguish a true messages from a false ones (e.g., random noise). (iii) *Undetectability-DataInputAssociation* specifies that a task should not be able to distinguish whether a piece of data is exists in a data object or not.

The example model shows an undetectability annotation linked with the message flow between the Send data to portal task and the Receive data start event. The annotation specifies that outsider adversaries must not be able to distinguish true messages sent over the message flow between the Send data to portal task and Receive data event from a false ones. In other words, at a specific time, an outsider adversary cannot detect whether the Tele-midicine device is sending data or not.

Unobservability can only be applied to message flows, leading to precisely one variant called *Unobservability-MessageFlow*: the sender of the message should be anonymous with respect to insider adversaries and the message itself should not be detectable by outsider adversaries.

The example model includes an unobservability annotation linked with the message flow between the Submit evaluation task and the Receive evaluation catch event. This annotation specifies that an outsider adversaries should not be able detect true messages being sent over the message flow from false ones, and the patient who sent messages over the message flow must be anonymous to the insider adversary.

4 Conflict Detection

Uncovering conflicts during the design of business processes is of vital importance to avoid privacy violations and expensive fixes in the later development phases. Detecting conflicts manually in annotated business process models is a challenging task, especially in real cases where business process frequently are composed of many tasks. We present an automated conflict detection techniques which takes as input a BPMN model with data-minimization and security annotations, and reports a list of *conflicts* and *potential conflicts*. The former represent definitive mismatches between two requirements; the latter may result in conflicts under certain circumstances. Consequently, our tool shows conflicts as *errors*, and potential conflicts as *warnings* to the user.

Conflicts between security and data-minimization requirements occur in two flavors: First, requirements related to the same asset in the system may be conflicting. For example, consider the *accountability* and *anonymity* annotations linked with the Send data to portal task in Fig. 1. For accountability, the system needs to track the executor of this task's responsibility, while the anonymity annotation specifies that the executor should be fully anonymous against insider adversaries. Second, requirements related to different, dependent assets may be conflicting. For example, in Fig. 1, consider the *anonymity* and *non-repudiation* annotations linked with the Fill evaluation form task and the Evaluation form data object, respectively. The former imposes that an executor to the Fill evaluation form task should be fully anonymous against insider adversaries; the latter indicates that an accessor to the Evaluation form data object should not be able to deny that she accessed the Evaluation form. Since the Fill evaluation form task writes data to the Evaluation form, a conflict is reported.

Potential conflicts as considered in our work result from control flows between activities with specified requirements. For example, Fig. 1 includes a path between the *anonymity*-annotated fill evaluation form task and the *non-repudiation*-annotated Submit evaluation task. Such situations not necessarily give rise to an actual conflict. For instance, imagine a flow between two tasks where the first task allows a customer to anonymously use a service and the second task allows the service provider to prevent a customer from being able to deny his payment for receiving a service. In this situation, it may be sufficient for a service provider to prove that a customer performed the payment task without uncovering which service a customer is paying for, and as a consequence, preserve the customer anonymity. Such potential conflicts should be reported and discussed during the design of the business process models.

Automated Conflict Detection Using Anti-patterns. We propose an automated conflict detection technique that relies on encoded knowledge about conflicts and potential conflicts between pairs of requirements. Specifically, we propose a catalog of *conflict anti-patterns* which are matched against the given business process model in order to detect conflicts and potential conflicts. Our patterns are formulated in a specialized query language, which extends an existing query language called SecBPMN2-Q [33]. SecBPMN2-Q supports custom graphical queries enriched with security requirements that can be

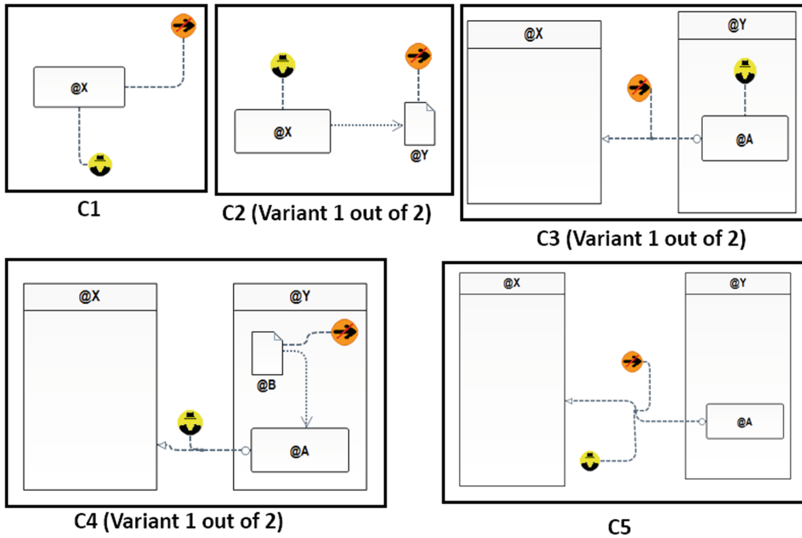


Fig. 3. Conflicts C1–C5 between *non-repudiation* and *anonymity* as anti-patterns.

matched to SecBPMN2 models, usually for verification purposes. We extended SecBPMN2-Q so that it supports our new data-minimization annotations as well, allowing us to specify conflicts as anti-patterns that can automatically detected.

Figure 3 shows a selection of anti-patterns defined using our SecBPMN2-Q extension. Together, the depicted anti-patterns represent all conflicts that can happen between *non-repudiation* and *anonymity*. The patterns include labels of the form "@X", which act as placeholders for element names, allowing us to formulate the anti-patterns in a domain-independent way. All anonymity annotations in Fig. 3 are specified with the following attributes: $\{anonymity\ level = full\ anonymous,\ insider = true\}$.

Consider, for example, conflicts C1 and C5 in Fig. 3. These conflicts arise when *non-repudiation* and *anonymity* annotations are linked to the same task or message flow, respectively. C1 can be matched to one place in the example model, which is highlighted in Fig. 1: The *anonymity* annotation of the Fill evaluation form is propagated to the Submit evaluation task, which is annotated with a *non-repudiation* annotation. In contrast, C5 does not occur in the example model, since the model does not have an *anonymity*- and *non-repudiation*-annotated message flow. C2, C3 and C5 each come in two variants, resulting from duality: the direction of the data object call (read or write) can be inverted, and the assignment of requirements to elements can be swapped.

Figure 4 shows three anti-patterns specifying potential conflicts between *anonymity* and *non-repudiation*. In these patterns, we use a *walk* relation (illustrated using an edge with double arrowhead), which is defined for pairs of activities, events or gateways. It allows to match all pairs of elements in the input model for which there is a path between the source and the target element.

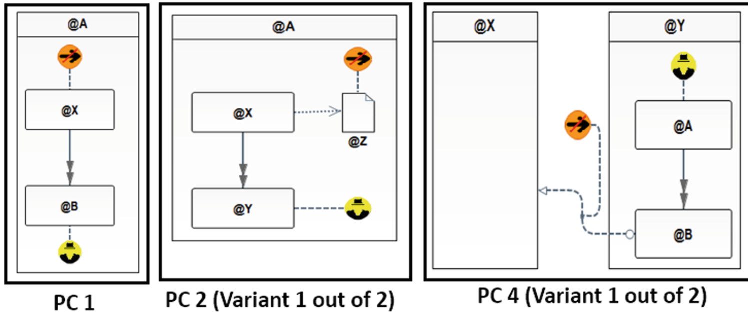


Fig. 4. Potential conflicts between *non-repudiation* and *anonymity* as anti-patterns.

Note that 3 out of 8 overall potential conflicts for the considered pair of requirements are shown. Two additional cases called PC3 and PC5 are formed in analogy to C3 and C5 in Fig. 3; three additional variants arise from duality like in the discussion of the conflicts. Again, all anonymity annotations are specified with the following attributes: {*anonymity level* = *full anonymous*, *insider* = *true*}.

The potential conflict PC1 in Fig. 4 includes a *non-repudiation*-annotated and an *anonymity*-annotated task between which a path exists. PC4 specifies a path between an *anonymity*-annotated task and another task that sends messages over a *non-repudiation*-annotated message flow. Both situations may lead to a conflict, depending on the actual circumstances in the system.

From matching potential conflicts PC1 and PC4 to the example model in Fig. 1, two warnings will be reported: (i) There is a path between the *anonymity*-annotated Fill evaluation form task and the *non-repudiation*-annotated Submit evaluation task, thus violating PC1. (ii) There is a path between the *anonymity*-annotated Fill evaluation form task and the Submit evaluation task which sends messages over a *non-repudiation*-annotated message flow, leading to a violation of PC4.

Catalog of a Domain-Independent Conflicts. As mentioned in Sect. 2, SecBPMN2 in fact supports 10 different security requirements. Similar to our data-minimization annotations, the same security annotation can be linked to different BPMN elements. We analyzed all possible situations where conflicts or potential conflicts between a security and a data-minimization annotation may happen. For each identified situation, we specified an anti-pattern using our extension of SecBPMN2-Q.

We now give an overview of the resulting catalog of anti-patterns. A more detailed account is found in [30]. Table 1 shows all pairs of requirements for which we identified a (potential) conflict. Each cell shows the number of conflicts, plus the number of potential conflicts between the considered pair of requirements. For example, there are 8 conflicts and 8 potential conflicts for non-repudiation and anonymity. The origin of these numbers is explained in the previous descriptions of Figs. 3 and 4. The other numbers arise from the various possibilities of

linking a data-minimization or a security annotation with other BPMN elements. In total our catalog contains 140 anti-patterns.

Table 1. Overview of conflict + potential conflict anti-patterns per pair of requirements.

Requirement pair	Accountability	Authenticity	Auditability	Non-repudiation	Non-delegation	Binding of Duties	Separation of Duties	Anonymity	Confidentiality	Integrity	Availability
Anonymity	2+2	6+6	8+8	8+8	2+2	1+0	1+0	4+5	0+8	0+11	0+11
Unlinkability	0+0	0+0	0+0	0+0	0+0	0+1	0+0	0+0	0+0	0+0	0+0
Unobservability	1+1	3+3	4+4	4+4	1+1	0+0	0+0	2+2	0+4	0+6	0+6

Considering conflicts and potential conflicts, *Accountability*, *Authenticity*, *Audibility*, and *Non-delegation* represent different requirements to keep insider users accountable for their actions. To preserve them, the identity of an action's executor must be verified. Therefore, similarly to *Non-repudiation*, all of these security concepts may have conflicts or potential conflicts with *Anonymity* (where required against insiders) and *Unobservability*, since part of its definition implies full anonymity against insiders.

Binding and *Separation of duties* can conflict with *Anonymity* if any of the activities to which they are applied also require to be executed anonymously. For instance, it will be hard, in case of *Binding of Duties*, to prove that two fully-anonymously executed activities are executed by the same person or not. A potential conflict between the *Binding of duties* and *Unlinkability* is also possible: *Unlinkability* is linked to two pools and indicates that the two process executions should not be linked to each other as related. Therefore, it may conflict with *Binding of Duty.*

Confidentiality, *Integrity*, and *Availability* represent different requirements to allow authorized users to read, modify, or access a system asset, respectively. The satisfaction of these requirements relies on authorization, which, however, does not necessarily imply identification: The literature provides many techniques for performing authorization without uncovering the real identity of an action executor, for example, *zero-knowledge protocols* [23]. However, the system developers may choose to implement these requirements by a mechanism that relies on identification, such as access control, which may lead to conflicts with data-minimization requirements. Hence, a decision about whether a conflict arises cannot be made on the abstraction level of process models. Therefore, as shown in Table 1, we classified the interactions between these security requirements and the data-minimization requirements as potential conflicts.

In some cases, *Confidentiality* and *Integrity* are considered as supplementing requirements to *Anonymity* [15]. For instance, anonymity against outsider adversaries implies that the outsider adversaries should not be able to trace a message back to its sender. However, if the sent message contains personal identifiable information and it is sent in clear (i.e., without encryption), an outsider attacker can easily link the messages to its sender. Such kind of interactions can not be considered as conflicts or potential conflicts, and thus, they are omitted from Table 1.

Conversely, conflicts may not only occur between security and data-minimization requirements. Table 1 indicates that a particular *Anonymity* annotation might conflict with other *Anonymity* or *Unobservability* annotations. For instance, requiring full anonymous execution for an activity is in conflict with requiring the users to execute the same activity anonymously using their *pseudonyms*. *Undetectability*, by definition [28], only shields against outsider attackers. Therefore, it is omitted from the Table 1 since it does not give rise to conflicts with security or data-minimization requirements.

5 Tool Support

We developed a prototypical implementation of our work on top of STS [2], the supporting tool for the BPMN extension SecBPMN2 [33]. Our implementation supports the two main contributions of this work: First, the modeling of data-minimization and security requirements in BPMN models, using a suitable model editor. Second, automated conflict analysis in data-minimization- and security-annotated BPMN models, based on our catalog of anti-patterns. The examples shown Figs. 1, 3 and 4 come from screenshots of our implementation. Our conflict detection approach takes as an input a security- and data-minimization-annotated BPMN model. The output is a set of textual messages that describe the detected conflicts. On demand, the conflict can be highlighted in the model. For example, the highlighted path in Fig. 1 is the result of selecting the conflict message that describes the PC1 anti-pattern in Fig. 4. Our implementation is available online at http://www.sts-tool.eu/downloads/secbpmn-dm/.

6 Case Study

To study the feasibility our approach, we applied it in a healthcare scenario. We extended a teleconsultations healthcare management case study from the Ospedale Pediatrico Bambino Gesù, a pediatric Italian hospital. The case study was part of the *VisiOn* research project [3]. The main objective of VisiOn consists in increasing the citizens awareness on privacy. The final outcome of the project was a platform that can be used by public administrations and companies to design their systems, using privacy as a first-class requirement.

The teleconsultations case study described a situation where a patient EHR can be transfered from the OPBG system to specialists in another hospital for a

teleconsultations purposes. In this scenario many security requirements are considered (e.g., confidentiality, accountability) but the privacy preferences were more related to data anonymization. In this paper, we extended this scenario to cover situations where data-minimization plays an important role not only protecting the users data but also their activities and communications. To this end, we modeled a process featuring an over distance healthcare service, an excerpt being shown in Fig. 1. Using our approach, as explained in Sect. 3, we were able to enrich the model with data-minimization requirements that represent privacy preferences for patients.

For conflict detection, we annotated the model with security requirements that represent security needs from the system point of view. Assessing the accuracy of conflict detection based on this model required a ground truth. To this end, we manually analyzed the model and identified 8 conflicts and 20 potential conflicts, a subset being discussed in Sect. 4. Applied to the model, our conflicts detection technique precisely detected these expected 8 conflicts and 20 potential conflicts. The used version of the model with all data-minimization and security requirements can be found in https://github.com/QRamadan/conflictsDetection/.

7 Usability Validation

Our main contribution is twofold: we provide support for the enrichment of the BPMN models with data-minimization requirements, and the detection of conflicts between data-minimization and security requirements. As a preliminary evaluation for both contributions, we performed a user experiment that focused on two research questions: **(RQ1)** *How usable are our data-minimization annotations, compared to textual requirement specifications?* **(RQ2)** *How useful is the conflict detection output?* The focus of RQ1 is on our new data-minimization annotations; the security annotations from SecBPMN2 were already evaluated in an earlier work [33]. As participants, we recruited 6 doctoral students and 1 post-doc from three institutions. We asked the participants to rate their experience in process modeling (in particular, BPMN), privacy, and security using 5-point Likert scales. Six participants rated their BPMN experience as 3 or 4. Six participants rated their security expertise as 3 or 4. The self-assessed privacy experience was 4 for one participant, 3 for four participants and 2 for two participants. In total, this distribution approximates the knowledge of the intended user group.

The set-up of our experiment involved a questionnaire with embedded model excerpts based on the model from our case study. Models were included in two versions: with visual data-minimization and security annotations (*proposed approach*), and with textual data-minimization requirement description and visual security annotations (*baseline*). The participants received a description sheet of all used annotations. For RQ1, participants were asked to complete comprehension tasks and, afterward, to state their subjective notation preference for solving our tasks, for communication with non-technical stakeholders, and for developing their own projects. For RQ2, participants were asked to identify conflicts

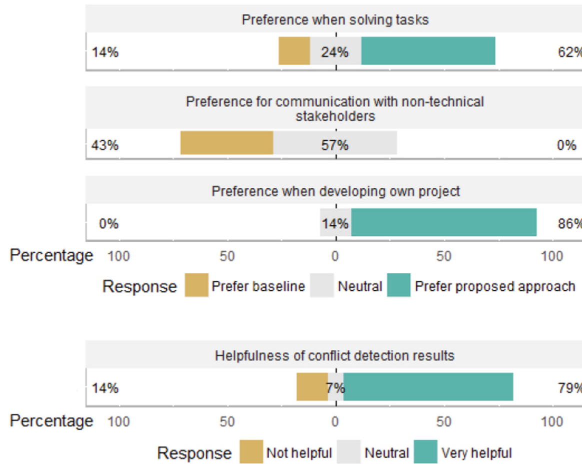


Fig. 5. Results: preference scores for notation (RQ1) and helpfulness of output (RQ2).

manually. Afterwards, we showed them the output of our tool and asked them to rate its helpfulness to be used it for this task. We also asked the participants for informal feedback using a free-form input field.

The summarized results are shown in Fig. 5; the shown scores accumulate the answers to multiple related questions. Regarding RQ1, we found that the participants mostly preferred our proposed approach for solving the tasks (62%), and for developing their own project (86%). One participant stated that "*with the extension, it's a lot easier to detect the model elements that are affected by a requirement than with the text version (have to find the relevant elements and correlate text and model)*". However, for communication with non-technical stakeholders, all participants gave a neutral (57%) or negative (43%) answer. An explanation offered by a participant was that to "*fully understand the effect of the annotations is very hard. This is the reason why I rated both variants equally usable for non-technical audience*". Regarding RQ2, the majority of participants rated the conflict detection results as very helpful for the identification of conflicts (79%). Two of the participants stated that the detection results pointed them to conflicts that they had not noticed when inspecting the models manually. In summary, our results give a promising outlook for the usability of our approach.

Threats to Validity. Owing to the limited sample size, we relied on descriptive statistics, leaving a comprehensive user study with a more rigorous statistical analysis to future work. The actual usefulness in practice may significantly depend on the considered model, of which we only considered one in our experiment. Usability was assessed through a subjective questionnaire rather than objective performance measures. This threat can be addressed using a different kind of experiment, as we plan to conduct in the future. Finally, we only considered the comprehension, rather than the editing of models. On the other hand, understanding is a necessary part of any editing process.

8 Related Work

Conflicts Between Security and Data Minimization. To the best of our knowledge, no existing approach supports conflict detection between security and data-minimization requirements. Hansen et al. in [15] defined six privacy and security goals for supporting the privacy needs of users. The authors considered a subset of the data-minimization concepts in [28], namely *anonymity* and *unlinkability*, and discuss their relationships. However, conflicts are discussed on the conceptual level, while in our work, we argue that the specific conflicts arising in a system can be identified by analyzing the data-system's minimization and security requirements. The perspective papers of Ganji et al. [13] and Alkubaisy [6] highlight the importance of detecting conflicts between security and privacy requirements, for data-minimization requirements in particular. Both papers discuss the components required for a potential approach, however, without providing a complete solution. Ganji et al. [13] envision a realization based on the SecureTropos framework as future work.

Data-Minimization-Aware Approaches. Various works in security requirements engineering aim to specify privacy requirements using the data-minimization concepts proposed in [28]. In Deng et al.'s LINDDUN framework [11], both misuse cases and data-minimization requirements can be identified by mapping predefined threat-tree patterns to the elements of a data-flow diagram. Kalloniatis et al. [17] propose the Pris methodology, which maps data-minimization and other security concepts to a system's organizational goals to identify privacy requirements. Pris introduces privacy-process patterns that describe the effect of privacy requirements to organizational processes. Mouratidis et al. [26] present a conceptual framework that combines security and data minimization concepts, and show its use to specify details about privacy goals such as the involved actors and threats. Beckers et al. [8] propose a privacy-threats analysis approach called ProPAn that uses functional requirements modeled in the problem-frame approach to check if insiders can gain information about specific stakeholders. Ahmadian et al. [4] support a privacy analysis for system design models, based on the four privacy key elements of purpose, retention, visibility and granularity. Since none of these approaches considers conflicts between data-minimization and security requirements, our approach can be seen as complementary: Their output can be used as input for our approach to allow the enrichment of the business process models with data-minimization and security requirements and then to perform conflict detection.

Diamantopoulou et al. [12] provide a set of privacy process patterns for data-minimization and security concepts, aiming to provide predefined solutions for different types of privacy concerns in the implementation phase. In addition to textual description of the patterns, BPMN design patterns were provided to guide operationalization at the business process level. This work is complementary to ours, as it focuses on the implementation of data-minimization requirements, rather than on the detection of conflicts.

9 Conclusions and Future Work

We proposed an extension of the BPMN modeling language to enable the specification of data-minimization and security requirements in a unified framework. Based on this extension, we introduce a technique for conflict detection between the specified requirements. Our technique analyses data-minimization- and security-enriched models based on a catalog of a domain-independent anti-patterns, which we formulated in an extension of a graphical query language called SecBPMN2-Q. We validated our approach in a case study based on a healthcare management system, and an experimental user study.

In the future, we aim to formally validate the completeness of our technique. Encoding the semantics of data-minimization and security requirements using graph transformations would allow us to apply formal conflict detection [9,19] for that purpose. We also aim to extend our approach to support the resolution of conflicts. As a first step, we aim to extend existing work in [33] to ensure that the enriched model is aligned with the collected security and data-minimization requirements. This will allow us to identify and fix unintentional conflicts (e.g., errors during the enrichment of the model with security and data-minimization requirements). Although a fully automated process would be appreciated, we believe that the resolution of actual conflicts (e.g., between two requirements related to different views of system stakeholders) is a sensitive issue that requires human intervention, a further challenging task that involves reasoning on the privacy impact of different solution strategies [5,21]. Once that all conflicts are resolved, the system design typically needs to be aligned with the specified privacy and security requirements, a challenge that can benefit from the use of model transformation technology [29]. Finally, we intend to perform a comprehensive user experiment to study the usability and validity of our approach in a broader setting.

Acknowledgment. We wish to thank Paolo Giorgini and the STS tool development team in the University of Trento for providing us with access to the source code of the STS tool. We also wish to thank the participants of our study. This research was partially supported by: (I) The Deutsche Akademische Austauschdienst (DAAD), (II) the project "Engineering Responsible Information Systems" financed by the University of Koblenz-Landau.

References

1. BPMN 2.0. http://www.omg.org/spec/BPMN/2.0/
2. STS. http://www.sts-tool.eu/downloads/secbpmn-dm/
3. VisiOn. http://www.visioneuproject.eu/
4. Ahmadian, A.S., Strüber, D., Riediger, V., Jürjens, J.: Model-based privacy analysis in industrial ecosystems. In: Anjorin, A., Espinoza, H. (eds.) ECMFA 2017. LNCS, vol. 10376, pp. 215–231. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_13

5. Ahmadian, A.S., Strüber, D., Riediger, V., Jürjens, J.: Supporting privacy impact assessment by model-based privacy analysis. In: ACM Symposium on Applied Computing. ACM (2018, to appear)
6. Alkubaisy, D.: A framework managing conflicts between security and privacy requirements. In: International Conference on Research Challenges in Information Science, pp. 427–432. IEEE (2017)
7. Arsac, W., Compagna, L., Pellegrino, G., Ponta, S.E.: Security validation of business processes via model-checking. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) ESSoS 2011. LNCS, vol. 6542, pp. 29–42. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19125-1_3
8. Beckers, K., Faßbender, S., Heisel, M., Meis, R.: A problem-based approach for computer-aided privacy threat identification. In: Preneel, B., Ikonomou, D. (eds.) APF 2012. LNCS, vol. 8319, pp. 1–16. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54069-1_1
9. Born, K., Lambers, L., Strüber, D., Taentzer, G.: Granularity of conflicts and dependencies in graph transformation systems. In: de Lara, J., Plump, D. (eds.) ICGT 2017. LNCS, vol. 10373, pp. 125–141. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_8
10. Brucker, A.D., Hang, I., Lückemeyer, G., Ruparel, R.: SecureBPMN: modeling and enforcing access control requirements in business processes. In: ACM Symposium on Access Control Models and Technologies, pp. 123–126. ACM (2012)
11. Deng, M., Wuyts, K., Scandariato, R., Preneel, B., Joosen, W.: A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. Requir. Eng. **16**(1), 3–32 (2011)
12. Diamantopoulou, V., Argyropoulos, N., Kalloniatis, C., Gritzalis, S.: Supporting the design of privacy-aware business processes via privacy process patterns. In: International Conference on Research Challenges in Information Science, pp. 187–198. IEEE (2017)
13. Ganji, D., Mouratidis, H., Gheytassi, S.M., Petridis, M.: Conflicts between security and privacy measures in software requirements engineering. In: Jahankhani, H., Carlile, A., Akhgar, B., Taal, A., Hessami, A.G., Hosseinian-Far, A. (eds.) ICGS3 2015. CCIS, vol. 534, pp. 323–334. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23276-8_29
14. Gürses, S., Troncoso, C., Diaz, C.: Engineering privacy by design. Comput. Priv. Data Protect. **14**(3) (2011)
15. Hansen, M., Jensen, M., Rost, M.: Protection goals for privacy engineering. In: 2015 IEEE Security and Privacy Workshops, SPW, pp. 159–166. IEEE (2015)
16. ISO and IEC: Common Criteria for Information Technology Security Evaluation - Part 2 Security functional components. In: ISO/IEC 15408, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) (2012)
17. Kalloniatis, C., Kavakli, E., Gritzalis, S.: Addressing privacy requirements in system design: the PriS method. Requir. Eng. **13**(3), 241–255 (2008)
18. Labda, W., Mehandjiev, N., Sampaio, P.: Modeling of privacy-aware business processes in BPMN to protect personal data. In: ACM Symposium on Applied Computing, pp. 1399–1405. ACM (2014)
19. Lambers, L., Strüber, D., Taentzer, G., Born, K., Huebert, J.: Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In: International Conference on Software Engineering. IEEE/ACM (2018, to appear)

20. Maines, C.L., Llewellyn-Jones, D., Tang, S., Zhou, B.: A cyber security ontology for BPMN-security extensions. In: International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing, pp. 1756–1763. IEEE (2015)
21. Meis, R., Heisel, M.: Systematic identification of information flows from requirements to support privacy impact assessments. In: International Joint Conference on Software Technologies, vol. 2, pp. 1–10. IEEE (2015)
22. Menzel, M., Thomas, I., Meinel, C.: Security requirements specification in service-oriented business process management. In: International Conference on Availability, Reliability and Security, pp. 41–48. IEEE (2009)
23. Mohr, A.: A survey of zero-knowledge proofs with applications to cryptography, pp. 1–12. Southern Illinois University, Carbondale (2007)
24. Moody, D.: The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Softw. Eng. **35**(6), 756–779 (2009)
25. Morton, A., Sasse, M.A.: Privacy is a process, not a PET: a theory for effective privacy practice. In: Proceedings of the 2012 Workshop on New Security Paradigms, pp. 87–104. ACM (2012)
26. Mouratidis, H., Kalloniatis, C., Islam, S., Huget, M.-P., Gritzalis, S.: Aligning security and privacy to support the development of secure information systems. J. UCS **18**(12), 1608–1627 (2012)
27. Mülle, J., von Stackelberg, S., Böhm, K.: A security language for BPMN process models. KIT, Fakultät für Informatik (2011)
28. Pfitzmann, A., Hansen, M.: A terminology for talking about privacy by data minimization: anonymity, unlinkability, unobservability, pseudonymity, and identity management. Technical report, TU Dresden and ULD Kiel (2011)
29. Ramadan, Q., Salnitri, M., Strüber, D., Jürjens, J., Giorgini, P.: From secure business process modeling to design-level security verification. In: International Conference on Model Driven Engineering Languages and Systems, pp. 123–133. IEEE (2017)
30. Ramadan, Q., Strüber, D., Salnitri, M., Riediger, V., Jürjens, J.: Detecting Conflicts between Data-Minimization and Security Requirements in Business Process Models, Long Version (2018). https://figshare.com/s/664b1c79c55130a44e79
31. Rodríguez, A., Fernández-Medina, E., Trujillo, J., Piattini, M.: Secure business process model specification through a UML 2.0 activity diagram profile. Decis. Support Syst. **51**(3), 446–465 (2011)
32. Saleem, M., Jaafar, J., Hassan, M.: A domain-specific language for modelling security objectives in a business process models of SOA applications. AISS **4**(1), 353–362 (2012)
33. Salnitri, M., Dalpiaz, F., Giorgini, P.: Modeling and verifying security policies in business processes. In: Bider, I., Gaaloul, K., Krogstie, J., Nurcan, S., Proper, H.A., Schmidt, R., Soffer, P. (eds.) BPMDS/EMMSAD -2014. LNBIP, vol. 175, pp. 200–214. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43745-2_14
34. Spiekermann, S., Cranor, L.F.: Engineering privacy. IEEE Trans. Software Eng. **35**(1), 67–82 (2009)
35. Van Blarkom, G.W., Borking, J.J., Olk, J.G.E.: Handbook of Privacy and Privacy-Enhancing Technologies. Privacy Incorporated Software Agent (PISA) Consortium, The Hague (2003)

36. Vivas, J.L., Montenegro, J.A., López, J.: Towards a business process-driven framework for security engineering with the UML. In: Boyd, C., Mao, W. (eds.) ISC 2003. LNCS, vol. 2851, pp. 381–395. Springer, Heidelberg (2003). https://doi.org/10.1007/10958513_29
37. Wolter, C., Schaad, A.: Modeling of task-based authorization constraints in BPMN. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 64–79. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75183-0_5

Life Sciences-Inspired Test Case Similarity Measures for Search-Based, FSM-Based Software Testing

Nesa Asoudeh and Yvan Labiche

Carleton University, Ottawa K1S5B6, ON, Canada
yvan.labiche@carleton.ca

Abstract. Researchers and practitioners alike have the intuition that test cases diversity is positively correlated to fault detection. Empirical results already show that some measurement of diversity within a pre-existing state-based test suite (i.e., a test suite not necessarily created to have diverse tests in the first place) indeed relates to fault detection. In this paper we show how our procedure, based on a genetic algorithm, to construct an entire (all-transition) adequate test suite with as diverse tests as possible fares in terms of fault detection. We experimentally compare on a case study nine different ways of computing test suite diversity, including measures already used by others in software testing as well as measures inspired by the notion of diversity in the life sciences. Although our results confirm a positive correlation between diversity and fault detection, we believe our results raise more questions than they answer about the notion and measurement of test suite diversity, which leads us to argue that more work needs to be dedicated to this topic.

Keywords: State-based testing · FSM · Fault detection · Test suite diversity

1 Introduction

Model-based testing [47] has received a lot of attention [33, 38, 43] for many reasons [34]: (i) Software complexity increases and user's tolerance of defects decreases, which calls for more functional testing; (ii) Software testing remains very expensive, which makes automated model-based testing very attractive; (iii) tools are mature enough to be used in a wide range of domains [44]. And there is empirical evidence that model-based testing can result in good return on investment [34].

Transition based models like state machines are widely used for model-based testing. Although a finite-state machine (FSM) can be used to describe the state-based behaviour of a small system, it becomes rapidly too large as the size of the system increases [20]. Extended finite state machines (EFSMs) are then preferred. Testing an implementation against its (E)FSM specification, and protocol testing are examples of applications of (E)FSMs in model-based testing. It can also be of great help in the challenging process of software integration [23], such as services in the cloud [19].

We argued in previous publications that state-based testing from an (E)FSM is a multi-objective optimization problem [4, 5]: (i) one wants feasible paths despite

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 199–215, 2018.
https://doi.org/10.1007/978-3-319-92997-2_13

conflicting actions and guard conditions (one wants test cases that can actually execute), which is akin to the un-decidable path sensitization problem [7]; (ii) one wishes to obtain a test suite that is adequate for some selection criterion [9] (transition coverage has proved to be a good minimum [47]) in an attempt to ensure some level of quality of the test suite; (iii) one is interested in increasing diversity among test cases since, intuitively and experimentally, diversity relates to fault detection [20]; (iv) one wishes to reduce costs, measured for instance as the cumulative length of test cases in a test suite. We presented [4, 5] a promising test suite construction procedure, using these four objectives, based on a multi-objective genetic algorithm.

One of the main factors affecting the success of any search-based algorithm is (are) the objective function(s). In this paper we specifically focus on similarity measurement. Although the impact of different test case similarity measures on effectiveness at detecting faults has been already studied [20], it is not clear whether those results would apply to a multi-objective test suite construction like ours [4, 5]. Indeed, the measures of similarity were used to evaluate a pre-existing test suite, in an attempt to relate similarity to fault detection (i.e., measure a test suite "quality" in terms of similarity and fault detection), not to guide the construction of an entire test suite such as in our multi-objective genetic algorithm.

As a genetic algorithm has its roots in biology and genetics we continue in the same line of thoughts and use methods which biologists use to measure similarity from individuals to populations. In this paper we therefore turn our attention to test suite diversity (the opposite of similarity), get inspiration from our colleagues in the life sciences, and look at how they determine diversity of species. We study different mechanisms to compute the diversity of an entire test suite, relying on different pair-wise diversity comparisons of test cases within a test suite on a case study. In this paper we specifically focus on state based testing from an FSMs, rather than from an EFSM, since we observed [6] that counters in EFSM guard conditions have an adverse effect on the convergence of our multi-objective search [4, 5]; we want to study measures of test suite diversity without the confounded, uncontrolled effect of such counters.

Section 2 discusses related work on diversity measurement in software testing and in the life sciences. Section 3 discusses the design of our case study and we discuss results in Sect. 4. Section 5 discusses threats to the validity of our study. We conclude in Sect. 6.

2 Related Work

We first summarize our entire test suite construction technique (Sect. 2.1). We then discuss related work in similarity measurement for software testing (Sect. 2.2) and similarity measurement, actually diversity, in the life sciences (Sect. 2.3).

2.1 Multi-objective Entire Test Suite Construction

We proposed to use a multi-objective genetic algorithm to create test suites from an (E)FSM [3–6]. We assume the reader is somewhat familiar with genetic algorithms and

only detail below aspects which make our solution different (we create an entire test suite) from others. Since the purpose of this paper is not to introduce our multi-objective test suite construction, we refer the reader interested in more details about that aspect of our work, including precise definitions of mutation and cross-over operators as well as other design decisions, to our previous publications [3–6].

The goal of our genetic algorithm was to find test suites that achieve high adequacy, ideally 100% adequacy, using the all-transitions selection criterion, have a high chance of being feasible in the case the EFSM has guard conditions (we use a surrogate measure of feasibility), minimize cost (i.e., cumulative number of triggered transitions in all test cases), and minimize the similarity between the test paths that constitute the test suites [20].

In a multi-objective genetic algorithm, a chromosome is a solution to the optimization problem, and genes compose a chromosome. In our context, a chromosome is a test suite and a gene is a test case, i.e., a sequence or path of EFSM transitions. One of the objectives of test suite construction is to achieve a certain level of adequacy according to a selection criterion. Since different adequate (i.e., satisfying a criterion) test suites usually exist and have varying numbers of test cases, our chromosomes have variable length (i.e., variable number of test cases/genes).

Since a chromosome has many characteristics (e.g., number of genes/test cases, length of each gene/test path, specifics of each gene/test path), a chromosome can be mutated in many different ways. Specifically, adding/removing a gene to a chromosome (test suite) means adding/removing a test case; Altering a gene means changing the test suite, for instance by mixing transitions between different test cases of the test suite. Similarly, the cross-over operator, which considers two chromosomes (in the general case), considers two test suites in our context. Cross-over can consist in exchanging test cases between two test suites but also exchanging transitions between test cases of different test suites.

2.2 Similarity Measurement in Software Testing

Different pair-wise similarity measures can be used on test cases [20] to identify to what extent two test cases (pair-wise comparison) differ (or are similar to each other). One measure, which is not limited to identical length sequences (recall we have variable length chromosomes), is the Levenshtein distance [18]. Although not the recommendation of Hemmati et al., this is the best measure of similarity they studied [20] that supports variable length sequences. To change this distance measure into a similarity measure, we reward matches by one point and penalize mismatches/gaps by assigning no point [20].

The Hamming distance is another widely used [20] similarity measure. It is based on the edit distance between two strings and is the minimum number of edit operations (i.e., insertions, deletions and substitutions) required to transform one string into the other. The Hamming distance can be used as a sequence-based similarity measure between strings of equal length. For variable length strings it is used as a set-based measure [20].

The last similarity measure we used is the set-based Gower-Legendre (Dice) measure since this is the one suggested by Hemmati et al. [20]. As the Dice measure is

a set-based measure, the order of transitions and their repetitions will be lost in computation. The Dice measure of two test paths *f* and *g* is based on commonalities and differences between the sets of symbols (in our case, transitions of the EFSM) of the test paths: $Dice(f,g) = |f \cap g| / (|f \cap g| + \frac{1}{2}(|f \cup g| - |f \cap g|))$.

Similarly to Hemmati et al. [20], we defined the similarity of a test suite (chromosome) as the sum of the similarity measures computed for each unordered pair of test paths (genes) in that chromosome. The following objective function needs to be minimized: $similarity_{Sum}(S) = \sum_{(f,g) unordered\, pairs\, in\, S} similarity(f,g)$.

The notion of test cases and test suite diversity (or similarity) is an active research topic. Although this paper is not a systematic mapping study, we report on a few significant pieces of work below. These principles (pair-wise measures and aggregation procedure) have been used to measure the diversity of test inputs of test suites [35]. The Euclidian (pair-wise) distance between test inputs is extensively used in (adaptive) random testing [39]. The cosine similarity, Jaccard distance, Euclidian distance measures are also used for pairwise comparisons of tests cases to rank them [48]. Shi et al. present a measure of distance entropy of a test suite, using Shannon entropy, based on pair-wise comparisons (Euclidian distance) [45]. We note (see below) that Shannon (Information) entropy is not the preferred measure of diversity in the life sciences. The test set diameter, based on Kolmogorov complexity, also measures test suite diversity [16]. It would be interesting to study how these measures relate to those used in the life sciences.

Our work is similar to existing published work that study similarity of state-based tests, with the difference that we use measures of similarity to construct test suites instead of measuring diversity of test suites created with other means. Another important difference is our use of similarity measures from the life sciences (see below).

2.3 Similarity Measurement from the Life Sciences

The three major parameters utilized by biologists to express variation of a population are diversity within a population, diversity among populations and distance between two populations. The first and second parameters are referred to as α- and β-diversity respectively and γ-diversity represents total diversity (α + β) [31].

Kosman [31] divides methods of diversity analysis into five categories: Genotypic methods, Gene methods, Genetic methods, Functional methods and True diversity methods. *Genotypic methods* were initially developed for measuring species diversity. They use information about relative frequencies of genotypes in a population. In our context, a genotype can be mapped to a transition, pair of transitions, paths, sub-paths or states. *Gene methods* analyze gene variation in populations and are based on calculations of frequencies of alleles at individual loci[1]. In our context, transitions are example of alleles, and each position of transitions in a test path is a locus. After measuring dissimilarity between pairs of individuals using any measure, a *genetic method* calculates the diversity of a population using a matrix of dissimilarities in which each row and each column represents an individual and each element is the

[1] An allele is a form of a gene (one member of a pair) that is located at a specific position on a specific chromosome. Locus (plural: loci) is the location of a gene on a chromosome.

pair-wise dissimilarity between two individuals. *Functional methods* are mostly used in the study of ecosystems. Each organism (species) is described by a set of functional traits which are believed to be important for the function of an ecosystem. Differences between organisms are assessed based on dissimilarity between their functional profiles. The term *true diversity* was first coined by Jost [25, 26] and later the idea was complemented by Tuomisto [46].

Genotypic, gene, genetic, functional and true diversity methods are further divided based on the type of diversity parameters used to measure dissimilarity: i.e., diversity within population (α), distance between populations, diversity among populations (β) and total diversity (γ). A complete list of parameters and categories can be found in Kosman's paper [31].

In our multi-objective genetic algorithm, we need to measure diversity within a population, i.e., α-diversity: we have a population of test cases (individuals) and we are interested in how diverse they are. Therefore, we now briefly describe methods belonging to each of the five categories that are used to measure α-diversity.

The three most commonly used *genotypic* methods to measure diversity within a population P [31] are Shannon Information, Simpson diversity, and the Stoddard index. These measures rely on the frequency of a genotype in the entire set of genotypes. Nei [37] defined the most common *gene* method [32] for the assessment of diversity within a population. The measure relies on the frequency of alleles at different loci in genes. The average dissimilarity within population and Kosman's diversity within population are two common *genetic* methods [32]. Both rely on a pair-wise dissimilarity measure: $\delta(x_i, x_j)$ is the dissimilarity between individuals x_i and x_j. With Kosman's diversity within a population of size *n*, individuals are matched to form n pairs in a way that the sum of dissimilarities between the pairs has its maximum value. Finding those pairs is similar to solving the assignment problem [8]. Rao has proposed the quadratic entropy as an index of *functional* diversity within a population [40], relying on frequencies of genotypes and pair-wise dissimilarity values. Ricotta and Szeidl [41, 42] proposed two different approaches to generalize Shanon's entropy, originally a *genotypic* measure, as a *functional* method to measure diversity within populations. Jost proposed a conversion of common diversity measures to *true diversity* [25].

Kosman and Leonard performed a conceptual analysis of the different types of methods mentioned above [32]. They compared those measures on organisms with asexual or mixed mode of reproduction. Based on their study, Kosman's diversity within population (KW) is more appropriate and informative than other measures as it includes both genotypic structure of a population and a measure of dissimilarity between various genotypes. Another interesting measure to consider based on this study is the average dissimilarity within population (ADW) as a representative of all the measures that are based on allele frequencies (e.g., Nei's measure).

Therefore, we selected Kosman diversity within population (KW), and average dissimilarity within population (ADW), to apply to our multi-objective genetic algorithm. In a population P with n individuals $x_1, \ldots, x_n$, $\delta(x_i, x_j)$ represents the dissimilarity between x_i and x_j according to measure δ. With $ASS^{\delta}_{max}(P, P)$ the matching of n pairs of individuals such that the sum of dissimilarities between the pairs has its maximum value, we have: $KW(P) = \frac{1}{n} . ASS^{\delta}_{max}(P, P)$ and $ADW(P) = \frac{1}{n^2} . \sum_{i,j=1}^{n} \delta(x_i, x_j)$.

3 Experimental Design

In light of related work, we selected three methods of computing test suite diversity: two population diversity methods from the life sciences (i.e., KW and ADW) and the summing up of pair-wise similarities (SUM) as used by others in the software testing field. Since these methods all rely on pair-wise comparisons of individuals in a population, we used the three functions that have been mostly used in the field, namely, Levenshtein distance, Hamming distance and Dice. This results in nine different combinations to experiment with. Because KW and ADW are based on pair-wise dissimilarities rather than similarities we used the original version of all these three pair-wise measures without converting them to similarity measures.

The main research question we were trying to answer was: what is the effect of using different methods of computing similarity on cost and effectiveness of test suites generated by our multi-objective genetic algorithm? To be more specific, we broke down this question into the following research questions: (i) What is the effect of using different pair-wise dissimilarity measures on cost, or on effectiveness? (ii) What is the effect of using different methods of computing diversity on cost, on effectiveness?

The rest of this section discusses the case study system we used (Sect. 3.1) and the procedure we followed during the case study (Sect. 3.2).

3.1 Case Study

We selected a case study system for which we have a state model (FSM) and source code: a simple Cruise Control (Fig. 1).

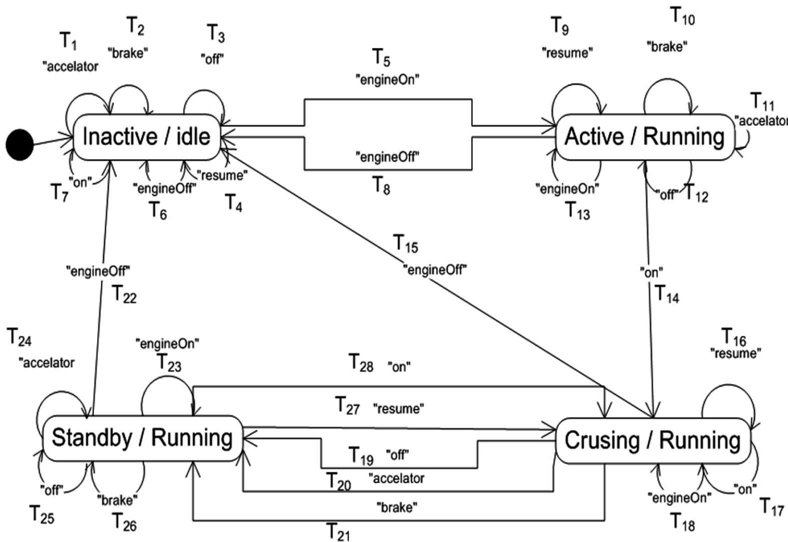


Fig. 1. FSM for the Cruise Control system

We selected this system because, although it does not have guards or actions and therefore any traversal of the state machine graph is a feasible test path (as a consequence we have really three fitness functions), it allowed us to check the correctness of our approach and focus on the three other fitness functions, and especially diversity/similarity measurement. Its simplicity, while remaining complex enough (28 transitions, seven events) and representative of a large number of similar FSMs that are used for testing purposes, allowed us to control some factors in the case study: For example, we know guards and counters in guards are difficult to handle [6]; We use an FSM (without guards) which allows us to remove the adverse effect of guards. Our four-objective optimization turns into a three objective one. With only three fitness functions we can furthermore plot results (not shown here due to space limits) and learn from those results in a qualitative analysis: e.g., we can visually observe the result of the competition between the fitness functions.

3.2 Procedure

We designed the following analysis procedure to answer our research questions.

- We ran each of the nine different versions of the genetic algorithm 100 times. Recall we have three pair-wise similarity measures and three ways to aggregate those pair-wise computations to compute a test suite diversity measure, resulting in nine possible measures of test suite diversity, which we implemented in nine versions of the genetic algorithm (all other characteristics of the genetic algorithm, e.g., seed population, operators, cross-over and mutation rates, remained the same). Since a genetic algorithm is inherently non-deterministic, we must account for possible variations in results. With 100 executions we can perform statistical analyses;
- We randomly selected 20 out of the 100 executions and collected their resulting Pareto front (i.e., final solutions found by the genetic algorithm). Also, to be fair in our comparisons, from each of the 20 Pareto front sets, this random selection only kept the (one) adequate test suite with the lowest cost. This means we picked the best solutions in terms of cost and adequacy for all different flavours of our genetic algorithm. We could not analyse all generated solutions and had to filter. We obtained 180 (20 times nine) different test suites;
- We compared the different versions of the genetic algorithm, through those 180 test suites, in terms of cost using Analysis of Variance (ANOVA) [15]. The characteristic that separates different treatments or populations from one another is referred to as the *factor* under study and each treatment or population is called a *level* of the factor. The goal of ANOVA is investigating if differences between different samples are due to chance or are caused by different treatments. In our case, we have two factors under study. The first factor is the pair-wise similarity (dissimilarity) measure, which has three different levels, i.e., Dice, Hamming distance and Levenshtein distance. The second factor is the method for measuring diversity within a whole test suite, which has three levels: KW, ADW, and SUM. We performed multiple single factor ANOVAs as well as a Two-way ANOVA for dependent variable cost. Our surrogate measure of cost is to count the cumulative number of transitions in all the test cases of a test suite;

- We executed these test suites on mutants and measured mutation scores, i.e., the proportion of seeded mutants (faults) that are killed (revealed) by a test suite. Mutation analysis is a well-known method commonly used for the evaluation of testing strategies [1, 2]. We used *MAJOR* [28], a mutation analysis tool for Java, to seed faults and perform the mutation analysis. We used all the mutation operators available with MAJOR and created 198 mutants for Cruise Control. When identifying whether a mutant is killed or alive, we need an oracle for each test case. Our detailed oracle was to check the target state, transition triggered as well as comparing the value of state variables for each transition in the transition sequence defining a test case. Comparisons took place between executions against mutants and executions against un-mutated code. Similarly to other studies of this kind reported in literature, we considered mutants that are not killed by any of the (180) test suites that are part of the experiment as equivalent. Similarly to cost, we used ANOVA (one-way, two-way) to compare the different versions of the genetic algorithm, through those 180 test suites, for dependent variable effectiveness.

4 Results

As mentioned in the previous section we had two different factors under study and each of them had three different levels. Cost and effectiveness were the two dependent variables. Following the statistical procedure described above resulted in three different single factor ANOVA tests with different methods of computing pair-wise similarity as the factor under study and cost of generated test suites as dependent variables. It also resulted in three single factor ANOVA tests with different methods of computing diversity within a test suite as the factor under study and cost of generated test suites as dependent variable. Replacing cost with the effectiveness at detecting faults (mutation score) as the dependent variable resulted in six additional single factor ANOVAs (three for pair-wise similarity measure as the ANOVA factor and three with method of computing diversity within a test suite as the factor under study). Combining two factors resulted in two different two-way ANOVAs, one of them with cost and the other with effectiveness at detecting faults as the dependent variables. Detailed results of all twelve different one-way ANOVA tests and the two two-way ANOVA tests are not provided here due to space constraints. They can be found publicly elsewhere [3].

Two assumptions about the data when using ANOVA are that samples have a normal distribution and standard deviations are close enough (i.e. the largest standard deviation is not greater than twice the smaller one). Figures 2 and 3 show there is no major skewness in samples. Standard deviations of samples are close enough to each other: Tables 1 and 2. Obviously, such assumptions are usually not met together in practice but ANOVA is known to be very robust to departures from these assumptions [24]. Therefore, we conclude that our data are amenable to the use of ANOVA.

In this section we separately study the effect of different configurations of our genetic algorithm on cost (Sect. 4.1) and effectiveness at finding faults (Sect. 4.2).

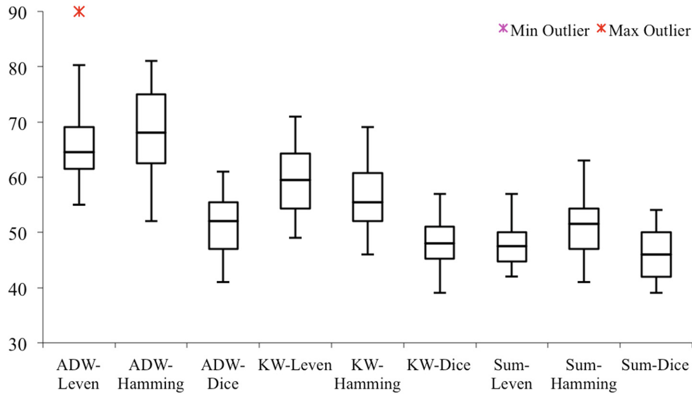


Fig. 2. Box plot and descriptive statistics comparing costs

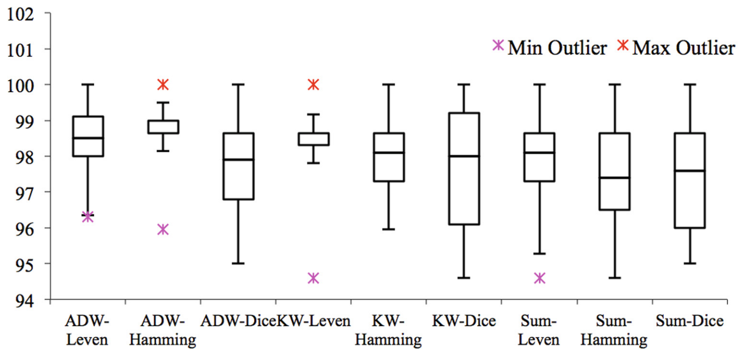


Fig. 3. Box plot and descriptive statistics comparing mutation scores

Table 1. Mean and variance of cost values

	Hamming			Levenshtein			Dice		
	ADW	KW	SUM	ADW	KW	SUM	ADW	KW	SUM
Average	69.1	56.3	51.1	65.9	59.2	48.3	51.8	47.9	44.9
Variance	108.7	48.2	30.0	91.3	44.2	23.8	29.0	27.4	32.7
SD (σ)	10.4	6.9	5.8	9.5	6.6	4.8	5.3	5.2	5.7

Table 2. Mean and variance of mutation scores

	Hamming			Levenshtein			Dice		
	ADW	KW	SUM	ADW	KW	SUM	ADW	KW	SUM
Average	98.65	98.31	97.90	98.95	98.24	97.84	98	98.15	97.77
Variance	1.34	1.31	2.96	1.83	2.51	2.57	2.47	2.93	2.16
SD (σ)	1.16	1.14	1.72	1.35	1.58	1.60	1.57	1.71	1.47

4.1 Effect on Cost

Results of the experiment with nine different configurations of our genetic algorithm are presented in Table 1. As mentioned earlier these nine different flavours are results of using different pair-wise similarity measures and aggregation methods. Each column of the table represents one configuration and its effect on cost in terms of average, variance and standard deviation of 20 test suites. Figure 2 presents the same result in the form of box plots representing the distribution of cost values for each implementation of our genetic algorithm.

When comparing different samples of data (i.e. the 20 test suites collected for each method) in terms of cost, the null hypothesis was rejected in all the six one-way ANOVA tests. This means, there is a relation between cost of test suites generated by our genetic algorithm and the method used to compute diversity/similarity within test suites (Dice vs. Hamming vs. Levenshtein), regardless of the method to aggregate pair-wise comparisons for computing diversity for an entire test suite (ADW, KW, SUM).

We also observed that among the three different measures we used to compute pair-wise similarities/dissimilarities, the Dice measure resulted in test suites of lowest cost on average (i.e. average cost of test suites was lower than the two other measures) regardless of the method used to compute diversity/similarity within test suites. This is in accordance with what Hemmati et al. have reported [20].

When comparing average cost of test suites based on the method used to compute diversity/similarity within test suites, summing up all the pair-wise similarities had the lowest cost value regardless of the method used to compute pair-wise similarity/dissimilarity. This means that the way methods from life sciences promote diversity within test suites results in test suites of higher cost.

If we study effect size, which depicts the strength of a treatment effect and is independent of sample size, we come up with the following conclusions. We used the Omega test, depicted as ω^2, which is an unbiased measure of effect size [14]. In general, Omega is considered as a more accurate measure of the effect size compared to other measures like Cohen's d measure. $\omega^2 = .01$ depicts a small effect and $\omega^2 = .06$ and $.14$ depict medium and large effects respectively [14]. The following are effect sizes for the different ANOVA tests we performed to investigate effect of size:

- Independent variable: pair-wise measure, test suite diversity: ADW $\rightarrow \omega^2 = 0.41$
- Independent variable: pair-wise measure, test suite diversity: KW $\rightarrow \omega^2 = 0.35$
- Independent variable: pair-wise measure, test suite diversity: SUM $\rightarrow \omega^2 = 0.16$
- Independent variable: test suite diversity, pair-wise measure: Dice $\rightarrow \omega^2 = 0.20$
- Independent variable: test suite diversity, pair-wise measure: Hamming $\rightarrow \omega^2 = 0.48$
- Independent variable: test suite diversity, pair-wise measure: Levenshtein $\rightarrow \omega^2 = 0.49$
- Two-way ANOVA: $\omega^2 = 0.30$

All the values are greater than 0.14 which means the similarity measurement method has a considerable effect on the cost of test suites.

4.2 Effect on Mutation Score

Similar to the previous section, we have summarized the results of the experiment in Table 2 and Fig. 3. With mutation score as the factor under study the results of one-way ANOVA tests were different. In all the six different scenarios (i.e. three different pair-wise similarity/dissimilarity measures and three different methods to compute similarity/diversity within whole test suites) we were not able to reject the null hypothesis.

The average mutation scores are very close to each other. In other words changing the method of computing diversity/similarity does not change mutation score significantly. Considering the fact that all the test suites selected for our statistical analysis were adequate test suites this was perhaps to be expected: i.e., covering all transitions, all other things being equal, may be enough in the Cruise Control case study to kill a similar number of mutants. In terms of effect size, using the Omega test (see previous section) we observe the following:

- Independent variable: pair-wise measure, test suite diversity: ADW $\rightarrow \omega^2 = 0.04$
- Independent variable: pair-wise measure, test suite diversity: KW $\rightarrow \omega^2 = 0.01$
- Independent variable: pair-wise measure, test suite diversity: SUM $\rightarrow \omega^2 = 0.03$
- Independent variable: test suite diversity, pair-wise measure: Dice $\rightarrow \omega^2 = 0.05$
- Independent variable: test suite diversity, pair-wise measure: Hamming $\rightarrow \omega^2 = 0.02$
- Independent variable: test suite diversity, pair-wise measure: Levenshtein $\rightarrow \omega^2 = 0.05$
- Two-way ANOVA: $\omega^2 = 0.01$

All of the values above are less than 0.06, which means there is a small effect size. Therefore, we can conclude that changing the method of computing similarity does not have a statistically nor a practically significant effect on mutation score. This is a different observation than that of Hemmati et al. [20].

4.3 Qualitative Analysis of Results

As mentioned earlier, Dice, as a pair-wise similarity measure resulted in test suites of lower costs compared to other measures. Also, we observed summing up pair-wise similarities is simply the best way of aggregating those pair-wise measures into a single value representing the diversity of a test suite. As Fig. 4 suggests, in addition to those two observations there is a trend among different methods of computing similarity within test suites. As we move from Hamming to Levenshtein and then Dice there is an overall monolithic decrease in cost of test suites with the exception of KW. Considering the fact that the KW measure uses a maximum distance to match pairs, and that longer tests (and therefore more expensive ones) have more chances to be matched, this is not entirely surprising. Both the Dice measure and the Hamming distance (as implemented in this work) are set-based measures while the Levenshtein distance is a sequence-based measure. Therefore, the Levenshtein typically results in longer sequences of transitions (test paths) to maximize KW distance, which increases cost of generated test suites.

In mutation analysis there were mutants that none of the test suites could kill. This could be due to the fact that we used transition coverage, and mutation operators (like Literal Value Replacement) that do not change the sequence of transitions triggered by a test case could not be killed by any of the test suites. This is the reason that all the mutation scores are high. As mentioned there is not much variation in mutation scores and we hypothesize that this may be due to the characteristics of the case study (FSM and source code) whereby all the test suites we used were adequate and were as effective as each other at detecting faults. Another reason could be our use of very detailed oracles.

To conclude, as changing the method of computing diversity does not have a significant effect on the mutation score, the method which results in test suites of lower cost is a better one in terms of cost and effectiveness. Therefore, the Dice measure is the best measure to compute pair-wise similarities/dissimilarities in terms of cost and effectiveness. Regarding the method of computing diversity within test suites, summing up all the pair-wise values had better results than the two methods from life sciences (ADW, KW) regarding cost and effectiveness.

5 Threats to Validity

Like any empirical evaluation, our work is subject to threats to validity.

Regarding threats to conclusion validity, which is about empirical work that leads someone to reach an incorrect conclusion about a relationship in one's observations, there are two points to consider. The first threat is due to the stochastic nature of genetic algorithms. To avoid this threat we executed each different version of our multi-objective genetic algorithm 100 times before starting the statistical analysis. As mentioned earlier, we sampled the set of 100 results and each sample had 20 Pareto front solutions in it. Also to be fair in our comparisons, from each Pareto front we selected the adequate test suites with the lowest cost. This means we picked the best solution in terms of cost and coverage for all different flavours of our genetic algorithm.

The second threat can be the size of the case study. Although the case study represents a typical case where a state machine is used to model behaviour, it is admittedly modest in size. However, behaviour of a similar size and complexity is usually modeled using state machines in UML-based development [12, 17] and in industrial case studies reported in the literature [10, 13, 21]. Additionally, it is very uncommon in practice to model subsystems or entire systems using state machines, as this is far too complex in realistic cases; one rather typically models communicating state machines and each separate communicating state machine is simple; there is one exception though, when one abstracts out from low level behavioural details of a (sub-) system in order to simplify the (E)FSM, in which case the state model used for testing purposes is (much) simpler than the actual implemented state-based behaviour. This is the case in our Cruise Control case study: the FSM does not model (abstraction) how the car speed evolves (up and down) as the driver (events triggering transitions in the FSM) breaks or presses on the gas pedal.

Construct validity relates to test measures and whether they are actually measuring what they are supposed to. We tried to minimize construct threats by the way we

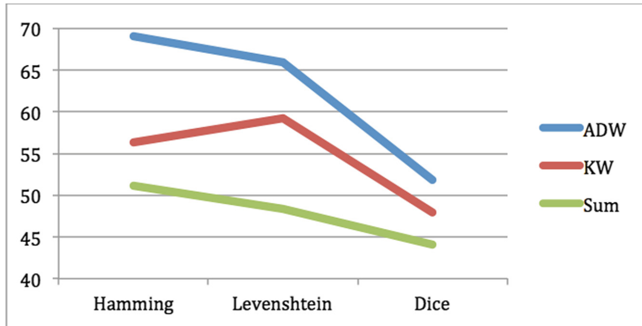


Fig. 4. Relation between cost of test suites (y-axis) and method used to compute similarity

designed our experiment. We ran our GA 100 times and used ANOVA to perform statistical analysis. As mentioned earlier our data seem to satisfy ANOVA assumptions and ANOVA has proved to be quite robust to deviations from these assumptions.

A threat to internal validity of our experiment is the fact that we used mutation score as a surrogate measure of effectiveness at detecting faults, and that we use the cumulative number of transitions as a surrogate measure of test suite cost. As a test suite's mutation score has proved to be correlated with its real fault detection rate [1, 2, 27] the probability that this threat affects the validity of our results should not be significant. A complementary study whereby state-based faults are simulated would nevertheless be interesting. Although test cost can include test set up, execution time and many other aspects, our measure is one typically used in the field. Another possible threat to internal validity is our use of precise oracles: we checked states but also other variable values, we checked after each transition and not only at the end of tests; this could explain why mutations scores are similar and factors/treatments are not discriminating.

Regarding external validity, which can limit the extent to which results of our experiments can be applied to similar situations, we need to consider the fact we performed our experiment in the context of our multi-objective genetic algorithm. Someone studying the relation between diversity and cost and fault detection in the context of another test generation approach may get different results. We tried to minimize this threat by using the Cruise Control FSM in which all the paths are feasible so that feasibility (one of the four fitness functions of our multi-objective genetic algorithm) didn't affect results of our experiment. The fact that we use only one case study, and that this case study may only be representative of a certain kind of a larger set of software systems, is also a threat to external validity, which limits our capability to generalize.

6 Conclusion

In this paper we investigated the effect of using different methods of computing diversity in a test suite made of several test cases. We specifically focussed on test suites that are constructed by our multi-objective genetic algorithm [3–6], which, while

maximizing test suite diversity (i.e., reducing similarity between its test cases), attempts to also reach adequacy (all-transitions criterion), to maximize test case feasibility (i.e., capability to find inputs that can make test cases executable), and to reduce test suite cost.

When computing the diversity of a test suite we relied on a mechanism to compare test cases in the test suite in a pair-wise manner and a mechanism to aggregate pair-wise comparisons to obtain a measure of test suite diversity. We considered three pair-wise comparison measures of tests that have been used by others: Dice, Hamming, Levenshtein. We used two methods from the life sciences to aggregate pair-wise values: Kosman's diversity and the average dissimilarity within population; and added one measure (sum of pair-wise comparisons) that others in software testing have used. We experimented with nine possible ways to compute test suite diversity, embedded in our multi-objective genetic algorithm, on one case study, and compared cost (cumulative number of transitions in a test suite) and effectiveness at finding faults (in fact mutants) of generated test suites. We used analysis of variance (ANOVA) to analyze the results.

We observed that changing the similarity computation method, either the pair-wise comparison or the aggregation mechanism, affects the cost of generated test suites without having a major effect on their fault detection rate. Among the nine possible combinations, our algorithm produces the best results in terms of cost and effectiveness at finding faults when we use the Dice measure to perform pair-wise comparisons of test cases in a test suite and we sum up pair-wise comparisons to obtain a measure of test suite diversity. These results partially conform to what others have observed before us [20] (Hemmati et al. use Dice and SUM), though not in the context of creating an entire test suite with an optimization procedure: our results confirm their results for cost but not for effectiveness at finding faults.

These results must be taken with care since our study should first be replicated on other case study systems, of varying size of complexity (as perceived from the state model as well as source code) to address threats to validity. That would additionally allow us to better understand the scalability of our multi-objective genetic algorithm solution: the multi-objective algorithm returns a Pareto front, that is a set of test suites out of which one should typically be selected. Also, we believe it would be interesting to study these alternative ways of measuring diversity with other test suite construction solutions (multi-objective or single-objective) than ours; do these measures have a similar (or different) effect when used with other test suite construction techniques?

Nevertheless, and we believe more importantly, we argue these results are to some extent counter-intuitive. Specifically, it is counter-intuitive (at least to us) that summing up pair-wise comparisons of test cases is a measure of test suite diversity. Specifically, in light of the large body of knowledge of diversity measurement in the life sciences, our results are surprising; it is not clear how summing up pair-wise comparisons of test cases does actually measure diversity. Is this a matter of measurement threat? Are we measuring the right things? The fact that the Dice pair-wise comparison is good in this context is also counter-intuitive, and this also relates to the notion of diversity. The Dice measure is a set-based measure; When using it on two state-based test cases (pair-wise comparison), which are transition paths in the FSM, the order of transitions and their repetitions are lost in the computation of diversity. On the other hand, our

intuition is that the sequence of transitions in a state model does matter from a testing point of view. It does matter at least from a modeling point of view and this was the initial reason for modeling the behaviour as an FSM. In other words, the notion of state, the notion of history is lost in the computation of diversity. Empirical results show that pairs of consecutive transitions, round trip paths, specific pairs of states and transitions do matter in terms of fault detection [9–11, 20, 22, 29, 30, 36]. Yet, in our experiments and others where diversity is studied [20], this information is lost while some relation between the measured notion of diversity and fault detection is observed. In other words, we have empirical work that shows transition and state sequences matter for fault detection and at the same time this does not matter when computing diversity which we assume is good for fault detection. Is our measure of diversity on par with our intuition of what diversity is?

We therefore believe much more work is warranted to try to explain how measures of test suite diversity actually relate to our intuitive idea/notion of test suite diversity.

Acknowledgements. This work was performed under the umbrella of a Collaborative Research and Development Grant with the Natural Sciences and Engineering Research Council of Canada (NSERC), with support from NSERC, CRIAQ (Consortium for Research and Innovation in Aerospace in Québec), CAE, CMC Electronics, and Mannarino Systems & Software. We would like to thank Professor Root Gorelick from the department of Biology at Carleton University for fruitful discussions on diversity.

References

1. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: Proceedings of the IEEE ICSE, pp. 402–411 (2005)
2. Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S.: Using mutation analysis for assessing and comparing testing coverage criteria. IEEE TSE **32**(8), 608–624 (2006)
3. Asoudeh, N.: Test generation from an extended finite state machine as a multiobjective optimization problem, thesis, Carleton University (2016)
4. Asoudeh, N., Labiche, Y.: A multi-objective genetic algorithm for generating test suites from extended finite state machines. In: Ruhe, G., Zhang, Y. (eds.) SSBSE 2013. LNCS, vol. 8084, pp. 288–293. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39742-4_26
5. Asoudeh, N., Labiche, Y.: Multi-objective construction of an entire adequate test suite for an EFSM. In: Proceedings of the IEEE ISSRE (2014)
6. Asoudeh, N., Labiche, Y.: On the effect of counters in guard conditions when state-based multi-objective testing. In: Proceedings of the IEEE Software Quality, Reliability and Security-Companion (2015)
7. Beizer, B.: Software Testing Techniques. International Thomson Computer Press, New York (1990)
8. Bellman, R., Cooke, K.L., Lockett, J.A.: Algorithms. Academic Press, New York (1970)
9. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools`, Object Technology. Addison-Wesley, Boston (1999)
10. Briand, L.C., Di Penta, M., Labiche, Y.: Assessing and improving state-based class testing: a series of experiments. IEEE TSE **30**(11), 770–793 (2004)

11. Briand, L.C., Labiche, Y., Wang, Y.: Using simulation to empirically investigate test coverage criteria. In: Proceedings of the IEEE/ACM ICSE, pp. 86–95 (2004)
12. Bruegge, B., Dutoit, A.H.: Object-Oriented Software Engineering Using UML, Patterns, and Java. Prentice Hall, Upper Saddle River (2004)
13. Chevalley, P., Thévenod-Fosse, P.: Automated generation of statistical test cases from UML state diagrams. In: Proceedings of the COMPSAC (2001)
14. Cohen, J.: Statistical Power Analysis for the Behavioral Sciences. Routledge, Abingdon (1988)
15. Devore, J.L.: Probability and Statistics for Engineering and the Sciences, 5th edn. Duxbury Press, Scituate (1999)
16. Feldt, R., Poulding, S., Clark, D., Yoo, S.: Test set diameter: quantifying the diversity of sets of test cases. In: Proceedings of the IEEE ICST (2016)
17. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison Wesley, Boston (2000)
18. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
19. Harman, M., Lakhotia, K., Singer, J., White, D.R., Yoo, S.: Cloud engineering is search based software engineering too. JSS **86**(9), 2225–2241 (2013)
20. Hemmati, H., Arcuri, A., Briand, L.: Achieving scalable model-based testing through test case diversity. ACM TOSEM **22**(1), 6:1–6:42 (2013)
21. Holt, N.E., Anda, B.C.D., Asskildt, K., Briand, L., Endresen, J., Frøystein, S.: Experiences with precise state modeling in an industrial safety critical system. In: Proceedings of the Models Workshop on Critical Systems Development Using Modeling Languages (2006)
22. Holt, N.E., Torkar, R., Briand, L., Hansen, K.: State-based testing: industrial evaluation of the cost-effectiveness of round-trip path and sneak-path strategies. In: Proceedings of the IEEE ISSRE, pp. 321–330 (2012)
23. Inverardi, P., Autili, M., Di Ruscio, D., Pelliccione, P., Tivoli, M.: Producing software by integration: challenges and research directions. In: Proceedings of the FSE (2013)
24. Iversen, G.R., Norpoth, H.: Analysis of Variance. Sage publications, Thousand Oaks (1987)
25. Jost, L.: Partitioning diversity into independent alpha and beta components. Ecology **88**, 2427–2439 (2007)
26. Jost, L.: GST and its relatives do not measure differentiation. Mol. Ecol. **17**, 4015–4026 (2008)
27. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing. In: Proceedings of the FSE (2014)
28. Just, R., Schweiggert, F., Kapfhammer, G.M.: MAJOR: an efficient and extensible tool for mutation analysis in a Java compiler. In: Proceedings of the ASE (2011)
29. Kalaji, A.S., Hierons, R.M., Swift, S.: An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models. IST **53**(12), 1297–1318 (2011)
30. Khalil, M., Labiche, Y.: On the round trip path testing strategy. In: Proceedings of the IEEE ISSRE, pp. 388–397 (2010)
31. Kosman, E.: Measuring diversity: from individuals to populations. Eur. J. Plant Pathol. **138**, 467–486 (2014)
32. Kosman, E., Leonard, K.J.: Conceptual analysis of methods applied to assessment of diversity within and distance between populations with asexual or mixed mode of reproduction. New Phytol. **174**, 683–696 (2007)
33. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - a survey. Proc. IEEE **84**(8), 1090–1123 (1996)

34. Legeard, B.: Model-based testing: next generation functional software testing. In: Dagstuhl Seminar Proceedings of Practical Software Testing: Tool Automation and Human Factors (2010)
35. Mondal, D., Hemmati, H., Durocher, S.: Exploring test suite diversification and code coverage in multi-objective test case selection. In: Proceedings of the IEEE ICST (2015)
36. Mouchawrab, S., Briand, L.C., Labiche, Y., Di Penta, M.: Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments. IEEE TSE **37**(2), 161–187 (2011)
37. Nei, M.: Analysis of gene diversity in subdivided population. Nat. Acad. Sci. U.S.A. **70**, 3321–3323 (1973)
38. Neto, A., Travassos, G.H.: Surveying model based testing approaches characterization attributes. In: Proceedings of the ACM/IEEE ESEM, pp. 324–326 (2008)
39. Patrick, M., Jiab, Y.: KD-ART: should we intensify or diversify tests to kill mutants? IST **81**, 36–51 (2017)
40. Rao, C.R.: Diversity and dissimilarity coefficients - a unified approach. Theor. Popul. Biol. **21**, 24–43 (1982)
41. Ricotta, C.: Bridging the gap between ecological diversity indices and measures of biodiversity with Shannon's entropy. Ecol. Model. **152**, 1–3 (2002)
42. Ricotta, C., Szeidl, L.: Towards a unifying approach to diversity measures: bridging the gap between Shannon's entropy and Rao's quadratic index. Theor. Popul. Biol. **70**, 237–243 (2006)
43. Saifan, A., Dingel, J.: A survey of using model-based testing to improve quality attributes in distributed systems. In: Elleithy, K. (ed.) Advanced Techniques in Computing Sciences and Software Engineering, pp. 283–288. Springer, Dordrecht (2010). https://doi.org/10.1007/978-90-481-3660-5_48
44. Shafique, M., Labiche, Y.: A systematic review of state-based test tools. STTT **17**(1), 59–76 (2015)
45. Shi, Q., Chen, Z., Fang, C., Feng, Y., Xu, B.: Measuring the diversity of a test set with distance entropy. IEEE Trans. Reliab. **65**(1), 19–27 (2016)
46. Tuomisto, H.: An updated consumer's guide to evenness and related indices. Oikos **121**, 1203–1218 (2012)
47. Utting, M., Legeard, B.: Practical Model-based Testing. Morgan Kaufmann, Los Altos (2006)
48. Wang, R., Jiang, S., Chen, D., Zhang, Y.: Empirical study of the effects of different similarity measures on test case prioritization. Math. Probl. Eng. **2016**, 19 p. (2016). Article ID 8343910

EMF Patterns of Usage on GitHub

Johannes Härtel, Marcel Heinz, and Ralf Lämmel[✉]

Software Languages Team, University of Koblenz-Landau,
Universitätsstraße 1, 56072 Koblenz, Germany
lammel@uni-koblenz.de
http://www.softlang.org/

Abstract. Mining software repositories is a common activity in software engineering with diverse use cases such as understanding project quality, technology usage, and developer profiles. Such mining activities involve, more often than not, a phase for data extraction from the source code in the repository with recurring tasks such as processing the folder structure (possibly on the timeline), classifying repository artifacts (e.g., in terms of the languages or technologies used), and extracting facts from the artifacts by parsing or otherwise. We describe a new approach for such data extraction; its key pillar is a declarative rule-based language for the uniform, inference-based extraction of facts from the repository (the file system), the artifacts in the repository (their content), and previously extracted facts. All inferred facts are maintained in a triple store. We describe a case study for the purpose of understanding the usage of *EMF*. To this end, we describe an emerging catalog of patterns of using *EMF* in repositories and we detect these patterns on *GitHub*. In our implementation, we use *Apache Jena* for which we provide dedicated language support tailored towards mining software repositories.

1 Introduction

Our long-term research objective is to apply megamodeling [4,5] to the problem of documenting software-technology usage in software projects. In our previous work [9,14,17,19,29], we focused on case studies, basic aspects of language support for such megamodeling, some forms of verification of a megamodel to correspond to a proper system abstraction, the axiomatization of the involved megamodeling expressiveness, the methodology for discovering megamodels, and surveying related concepts in the literature. We also use the term (models of) 'linguistic architecture' for such megamodels.

In this paper, we apply a mining- (or reverse engineering-) oriented view on documenting (or modeling) usage of technologies. We aim at extracting (inferring) facts uniformly from a software repository such that these facts classify artifacts in the repository and describe relationships, for example, related to dependencies, conformance, and correspondence. In particular, we aim at detecting patterns of technology usage. This problem is somewhat similar to design-pattern detection [24,25,34] and architecture recovery [2,16,18,27].

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 216–234, 2018.
https://doi.org/10.1007/978-3-319-92997-2_14

In the case study of this paper, we are concerned with *EMF*. We aim to better understand how *EMF* is used 'in the wild' in *GitHub* projects. To this end, we also describe an emerging catalog of patterns for *EMF*. There are, for example, patterns dealing with the more or less consistent and complete presence of interrelated artifacts: metamodel versus derived *Java* code versus generator model. In this paper, we do not study the evolution history of projects [40].

Our approach is original in that we use a declarative rule-based language for the uniform, inference-based extraction of facts from the repository (the folder structure), the artifacts in the repository (their content), and previously extracted facts. All inferred facts are maintained in a triple store. To give the reader an idea, consider the following trivial rule drawn from the case study:

```
1 (?x sl:manifestsAs sl:File) (?x sl:elementOf sl:XML) Extension(?x,"ecore") →
2    (?x sl:elementOf sl:Ecore).
```

Listing 1. Sample rule classifying Ecore files.

The body of the rule (i.e., the condition left to '→') quantifies over artifacts `?x` that are files with extension 'ecore' and readily known to be of 'type' (language) *XML*. For each such artifact, the head of the rule (right to '→') states that the artifact is also of 'type' (language) 'Ecore'. Thus, the rule infers triples for artifacts to be classified as metamodels.

Our rule-based approach is declarative and modular, when compared to the common use of problem-specific custom functionality for processing folder structure and file content (e.g., [9,21]). Our approach leverages an extensible suite of accessor primitives for interacting with standardized formats and structures such as *Java*, *XML*, and the file system (the folder structure) in a uniform manner. The rule-based approach helps in manifesting only the facts that are actually needed, as opposed to operating on complete ASTs or similar structures (as in, e.g., [1,34,35,39]); it is up to the rules and the accessor primitives to selectively extract and infer more facts. Such inference is similar to the event-condition-action paradigm [10].

Summary of the Paper's Contributions

- We develop a rule-based approach for uniform, inference-based data extraction from source-code repositories. While we leverage existing techniques known in the semantic web context and as specifically supported by *Apache Jena*, we provide dedicated language support on top of—tailored towards mining software repositories.
- We initiate work towards a structured catalog of *EMF* repository patterns. Each pattern captures a particular situation in a repository such as the presence of a certain kind of artifact and a potential or definite symptom of incompleteness or inconsistency (e.g., a missing or an unsynchronized artifact). This catalog calls for future work.
- We design and execute a case study for mining instances of *EMF* repository patterns in projects on *GitHub*. In this large-scale case study, we examine 5759 repositories. In this paper, we limit ourselves to only studying the most recent version of each project, leaving an evolutionary analysis to future work.

Roadmap of the Paper. Section 2 develops the rule-based approach for data extraction in mining software repositories. Section 3 develops the catalog of *EMF* repository patterns. Section 4 describes the design of the case study for *EMF* and the results. Section 5 discusses threats to validity. Section 6 discusses related work. Section 7 concludes the paper. The rules and the dataset for the case study and the implementation of the rule-based language are available online[1].

2 Rule-Based Data Extraction from Repositories

In our approach, the result of data extraction is a 'model'—a set of triples as inferred by the successful application of rules. A rule is of the form '*body* → *head*.' where *body* is the condition part and *head* corresponds to the inferred triples. That is, a rule matches the body against the current set of triples and for each resulting match, the head adds new triples to the model. Inference is a monotone process of inferring triples until a fixed point is reached, i.e., no more successful rule applications for new triples are applicable. The rules may use 'primitives', as discussed below, to access the repository (the folder structure and the content of files). The rule-based approach provides full control over materializing just the 'repository content of interest' in the model. In this section, we may occasionally use rules from the case study for illustrative purposes.

2.1 The Triple Model

Fact extraction infers triples or, in fact, labeled edges of a model (a graph). Nodes are URIs (Unified Resource Identifiers) or literals (such as strings). Each triple consists of a *subject* node (a URI), a *predicate* (a URI) and an *object* node (a URI or literal) where the predicate can be viewed as the edge label. In the earlier rule in Listing 1, we use `sl:File` as an object for classifying a repository artifact `?x` in the subject position with `sl:manifestsAs` as the predicate for the form of classification needed, i.e., an artifact to manifest as, for example, a folder or file. `sl` (for 'software language') represents a custom prefix.

Fact extraction starts from a graph with the following triple:

```
1 repository:/ sl:manifestsAs sl:Folder.
```

Listing 2. The initial model containing one triple.

The subject URI `repository:/` is special in that it refers to the root of the actual repository folder. Inference discovers folders and files and content thereof, as we discuss below.

2.2 A Scheme for Referencing Repository Content

Figure 1 illustrates the straightforward scheme that we assume for referring to repository content. This is the foundation for treating folders, files, and content particles (fragments of content) for files of different languages in a uniform

[1] https://github.com/softlang/qegal.

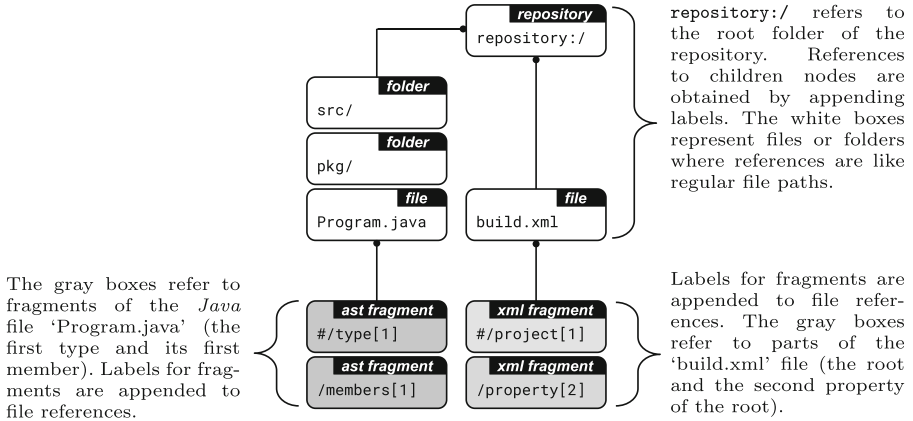


Fig. 1. URIs for referring to repository content for a small sample project containing an *ANT* and a *Java* file.

manner. Whether or not all the conceivable nodes are materialized in the model depends on the fact extractor, i.e., on the design of the rules. Typically, we materialize the catalog of the file system completely, but we materialize file content selectively, as we will discuss in more detail in a second.

2.3 Repository Accessors

The rule-based approach relies on primitives for accessing the repository (folder structure and content of files). In particular, the body of a rule may use such primitives to match or bind repository data such as file names and file content in terms of different representations (e.g., ASTs). Primitives may be also used in heads of rules for the purpose of data manipulation or for expanding given bindings into sets of inferred triples.

The primitives needed in the case study are shown in Table 1 with some omissions for brevity. The primitives are free of side effects to the repository. A primitive takes one or more arguments. Each argument is either a URI, a literal, or a placeholder to be bound. The application of a primitive may fail, if the given arguments are not in the corresponding relationship or placeholders cannot be bound to valid results.

2.4 Materialization of the Folder Structure

The following rule is the starting point for decomposing the repository folder:

```
1 (?x, sl:manifestsAs, sl:Folder) → DecFs(?x, "/*", sl:partOf, ?x).
```

Listing 3. Recursive folder decomposition into parts.

For the initial model (Listing 2), the placeholder `?x` in the body of the rule in Listing 3 is matched with `repository:/`. The head uses the `DecFs(folder, subj, pred, obj)` primitive, as described in Table 1, to

Table 1. Primitives for accessing folder structure (in general) and file content for *XML* and *Java* (in the case study).

Primitive	Parameters and Description
`IsFile`	`(artifact)` Matches if the `artifact` URI can be accessed as a file
`IsFolder`	`(artifact)` Matches if the `artifact` URI can be accessed as a folder
`Extension`	`(file, extension)` Matches if the `file` URI has a given extension. (When `extension` is a placeholder, it will be bound.)
`XmlWellformed`	`(file)` Parses the content referred to by the `file` URI and matches if the content is well-formed *XML*
`Children`	(`uri`, $\texttt{part}_1$,...,$\texttt{part}_n$) Decompose a `uri` into its parts split at '/'; parts filled in are matched; variable parts are bound
`DecFs`	`(folder, xpath, result)` Decompose references to file system by applying an XPath expression `xpath` on the repository starting from the given `folder`; assigns the URI of the first result to `result`
`DecFs`	`(folder, subj, pred, obj)` A variation on the previous primitive. It infers a *set* of triples, when used in the head of a rule. The inferred triples vary in the subject based on the argument `subj` which corresponds to the XPath expression in the basic form of `DecFs`. The arguments `pred` and `obj` are assigned to regular URIs. For instance, `DecFs(repository:/,"/*",sl:partOf, repository:/)` adds a `sl:partOf` triple for all first-level repository children (XPath '/*' for subject) of the repository (for the object)
`DecJava`,`DecXml`	Variations on `DecFs` working on *Java* ASTs or *XML* trees as opposed to the file system
`StrXml`, `StrJava`, `UriXml`, `UriJava`	Variations on the decomposition primitives for use in rule bodies, as described above. These variations perform data lookup as opposed to constructing a reference. The `Str...` primitives look up a string (e.g., an attribute in XML) and return it as a result. Likewise, the `Uri...` primitives look up a string which is a URI

decompose file paths, to compose subject URIs, and to infer triples with the `sl:partOf` predicate and `?x` as the object. Listing 4 presents the resulting triples.

```
1 repository:/ sl:manifestsAs sl:Folder. // Initial repository root classification.
2 repository:/src/ sl:partOf repository:/. // src is part of the repository.
3 repository:/build.xml sl:partOf repository:/. // build.xml is part of the repository.
```

Listing 4. Evolved model after applying the rule for folder decomposition.

We should enable the recursive application of the previous rule. To this end, we need to infer triples with the manifestation types `sl:Folder` and `sl:File` of discovered artifacts. The following rules match all (newly added) `sl:partOf`

triples, check whether the part is a folder or a file (using the corresponding primitives), and, if so, add suitable triples with the `sl:manifestsAs` predicate.

```
1 (?x, sl:partOf, ?parent) (?parent, sl:manifestsAs, sl:Folder) IsFolder(?x) →
2    (?x, sl:manifestsAs, sl:Folder). // Classifies folders.
3 (?x, sl:partOf, ?parent) (?parent, sl:manifestsAs, sl:Folder) IsFile(?x) →
4    (?x, sl:manifestsAs, sl:File). // Classifies files.
```

Listing 5. Classifying files and folders.

We would like to classify files by languages, as this is a prerequisite for diving deeper into the content, e.g., at the level of parse trees or ASTs. The following rules classify files by a language adding an `sl:elementOf` relationship between the file and the language at hand. We use `(?x sl:manifestsAs sl:File)` to select potential candidates `?x`. Listing 6 illustrates the rules for language classification.

```
1 (?x sl:manifestsAs sl:File) Extension(?x,"java") → (?x sl:elementOf sl:Java).
2 (?x sl:manifestsAs sl:File) XmlWellformed(?x) → (?x sl:elementOf sl:XML).
3 (?x sl:manifestsAs sl:File) (?x sl:elementOf sl:XML) Extension(?x,"ecore") →
4    (?x sl:elementOf sl:Ecore).
5 (?x, sl:manifestsAs, sl:File) Children(?x, _, "META−INF", "MANIFEST.MF") →
6    (?x, sl:elementOf, sl:Manifest).
7 (?x, sl:manifestsAs, sl:File) (?x, sl:elementOf, sl:XML) Children(?x, _, "build.xml") →
8    (?x, sl:elementOf, sl:Ant).
9 (?x, sl:manifestsAs, sl:File) Children(?x, _,"build.gradle") →
10    (?x, sl:elementOf, sl:Gradle).
11 (?x, sl:manifestsAs, sl:File) (?x, sl:elementOf, sl:XML) Children(?x, _, "pom.xml") →
12    (?x, sl:elementOf, sl:Pom).
```

Listing 6. Rules for basic language classification.

2.5 Pluggable Primitives

In our implementation, we rely on the extensibility of the rule engine. That is, *Apache Jena* allows us to plug Java code for new primitives into the engine. In the common semantic web-like use case, primitives are used for basic string or data manipulation. In our mining context, primitives correspond to significant functionality involving repository access, parsing, and more complex analyses. Consider the following implementation of the `Extension` primitive:

```
1 public class Extension extends QegalBuiltin {
2    @Override
3    public boolean trackedBodyCall(Node[] args, int length, RuleContext context) {
4       BindingEnvironment env = context.getEnv();
5       String file = getArg(0, args, context).getURI();
6       Node extension = getArg(1, args, context);
7       return env.bind(extension, NodeFactory.createLiteral(iolayer.extension(file)));
8    }
9 }
```

Listing 7. Implementation of the extension builtin.

This primitive for extension matching or extraction would be typically used in the body of a rule and thus, we need to implement a method `trackedBodyCall` which receives the bound or free arguments `args` and returns true if the primitive completes successfully, i.e., matching or binding succeeds. `Extension(file, extension)` takes two arguments: the `file` argument which must be given and the `extension` argument which is matched if present or bound if it is a placeholder. That is, the method `env.bind(current, expected)` returns true if the `current` and `expected` assignments match or, in the case that `current` is an open placeholder, it binds it to the `expected` value and returns true. For brevity, we omit the discussion of implementing primitives for head usage.

2.6 Dedicated Language Support

Our implementation leverages the *Apache Jena*[2] implementation for rule-based inference and triple processing. Our experience with the rule-based approach in case studies such as the one of Sect. 4 led us to advance the Jena approach by adding language support addressing the complexities of mining software repositories. In addition to the specific primitives needed, as discussed earlier, there are these aspects:

```
1  import org.softlang.qegal.buildins.decompose.*
2  import org.softlang.qegal.buildins.*
3  import org.softlang.qegal.buildins.string.*
4  import org.apache.jena.reasoner.rulesys.builtins.*
5
6  @prefix sl: <http://org.softlang.com/>.
7
8  (?file, sl:manifestsAs, sl:File) Children(?file, ?project, "META-INF", "MANIFEST.MF") ->
9      (?y, sl:elementOf, sl:Manifes
10
11 (?file, sl:elementOf, sl:Manifest
12     // Replace all strings.
13     ReplaceAll(?x, '("[^"]*)"', "
14     // Replace all details.
15     ReplaceAll(?xi, "(;[^,]*)|\\s
16     SplitToUri(?xii, ?file, sl:re
```

```
org.softlang.qegal.buildins.Children

Body: Children(uri, rest, part_1, ... part_n)

Decomposes the uri into fragments separated by slash. The last 1 to n fragi
n. The first fragments are bound to the rest parameter.
```

Fig. 2. IDE support for working with rules and primitives.

IDE support. Based on XText, [3][3] provide editing, syntax highlighting, auto-completion, code navigation, and other IDE support (see Fig. 2 for an illustration) also subject to JVM integration for the pluggable primitives of the rule-based language;

[2] https://jena.apache.org/.
[3] http://www.eclipse.org/Xtext/.

Logging and profiling. Log the execution of primitives with appropriate context and runtimes to enable debugging of the rule-based system and to check for performance hogs, thereby guiding optimization of primitives and rule set;

Exception handling. Recover from and log exceptions thrown by accessor primitives to enable completion of repository processing and post-mortem analysis, subject to a dedicated interface for primitives and housekeeping;

Virtualized access. Be able to switch between actual file-system-based access to artifacts (development mode) and immediate repository access without manifestation on the local file system (production mode);

Testing. Use a combination of parametrized and instance-based tests on white- and blacklisted repositories and the derived models, also using redundant repository processing (e.g., based on grep) to obtain reasonable baselines.

3 Towards an EMF Repository Pattern Catalog

EMF can be used in different ways in projects, subject to the *presence* of different types of artifacts, possibly with different *multiplicities* and in different *combinations*. In this paper, we begin work towards a catalog for *EMF* which covers these basic 'artifact' types: *Ecore Package*, as identified in '.ecore' files where one such file can possibly define several such packages; *Java Package* – an actual Java package containing derived classes according to a metamodel, a factory, and a package description; and *Generator Package* as identified in '.genmodel' files.

Artifacts of these types can be related in certain ways in a project. In fact, by checking on certain relationships, e.g., by determining the *absence* of certain artifacts or elements thereof, we may infer cases of *incompleteness* or *inconsistency*, where these are either *potential* or *definite* problems of usage or, in fact, of maintaining *EMF* usage in the repository. Table 2 lists patterns organized along these different dimensions (artifact type, presence, incompleteness, inconsistency, potential versus definite). We group by cardinalities of artifacts: single, double, and triple artifact patterns. For brevity, we exclude patterns related to XMI-based persistence in this paper. Generally, further work is needed to arrive at a more comprehensive catalog for *EMF*.

4 An MSR Case Study on EMF

We located projects with traces of *EMF* usage on *GitHub*. We assessed these projects in terms of some basic architectural characteristics to prepare a selection of well-understood project layouts for which a mining process is assumed to provide more comprehensible insights. Eventually, we detected *EMF* repository patterns as introduced in Sect. 3. We describe these phases here and summarize our findings.

Table 2. An *EMF* repository pattern catalog (some of the corresponding detection rules are discussed in Sect. 4. All of the rules are available online.)

Id	Cls.	Artifacts	Description and cause
Single artifact patterns			
E	Pres	- Ecore Pkg.	The presence of an Ecore Pkg. in '.ecore' files as root or subpackage
J	Pres	- Java Pkg.	The presence of a Java Pkg.
G	Pres	- Genmodel Pkg.	The presence of a Genmodel Pkg. in '.genmodel' files as root or subpackage
C	Pres	- Customized Java Pkg.	The presence of a Java Pkg. with customized interface or implementation
Double artifact patterns			
EJ1	Pot. Incomp.	- Ecore Pkg. - Java Pkg.(m[a])	A Java Pkg. cannot be found for a given nsURI as extracted from some Ecore Pkg. This is only a potential incompleteness, because a Java Pkg. could be potentially derived, if no customization is intended
EJ2	Def. Incomp	- Ecore Pkg. (m) - Java Pkg.	An Ecore Pkg. cannot be found for a given nsURI as extracted from some Java Pkg. This is a definite because the Java Pkg. is derived and thus, the underlying primary artifact (the Ecore Pkg.) should also be in the repository
EJ3	Pres	- Ecore Pkg. - Java Pkg.	The presence of a Java Pkg. and Ecore Pkg. with the same nsURI. One Ecore Pkg. can correspond to many Java Packages.
EE	Def. Incons	- Ecore Pkg. - Ecore Pkg.	An Ecore Pkg. with at least one competing Pkg. with the same nsURI
EJc1	Def. Incons.	- Ecore Pkg. - Java Pkg.	A Java class that is part of the Java Pkg. with a corresponding Ecore Pkg., but without a corresponding Ecore classifier (based on name comparison). For instance, one may have forgotten to remove a Java class derived from an earlier version of the metamodel
EJc2	Def. Incons.	- Ecore Pkg. - Java Pkg.	An Ecore classifier contained in an Ecore Pkg. with a corresponding Java Pkg., but without a corresponding Java classifier (based on name comparison). The Java Pkg. is thus out of sync with the Ecore Pkg. in the repository
Triple artifact patterns			
EJJ	Pres.	- Ecore Pkg. - Java Pkg. - Java Pkg.	An Ecore Pkg. with at least two corresponding Java Packages
EJG	Pot. Incomp	- Ecore Pkg. - Java Pkg. - Generator Pkg. (m)	For a corresponding pair of Java Pkg. and Ecore Pkg., a corresponding Generator Pkg. cannot be found

4.1 Locating Repositories

We used the *GitHub* search API to locate all recently indexed *Ecore*, *Generator Model* and *Java Model* files on *GitHub* as an indication of *EMF* usage in repositories. The corresponding queries are listed in Table 3.

Table 3. Queries for locating repositories through *GitHub* API.

Evidence	Query	Extension
Java model	"extends EObject {"	java
Ecore model	GenModel	ecore
Generator model	EClass	genmodel

(For what it matters, the API search limit is circumvented by recursive query segmentation which splits a query by setting an upper and lower bound in file size based on the returned number of total results. This process may miss some results.) The search API only considers heads of default branches and files smaller than 384 KB. A list of 5759 *GitHub* repositories was extracted from the query results.

4.2 Selection of Repositories

We applied the rule-based mining approach to recover the repository layout in terms of usage of build systems, project dependencies, and other aspects. We developed the following classifiers for repositories as an extension of the pattern catalog of Sect. 3:

Homogeneous versus **heterogeneous build system.** We search for traces of Manifest, POM, Gradle, and ANT, as modeled by the rules in Listing 6. In the homogeneous case, only one such technology is used; otherwise we apply the heterogeneous classifier. We assume that the heterogeneous situation is harder to understand in terms of project dependencies.

Single component versus **multiple components.** Based on an analysis of project dependencies, as described in more detail below, we determine the number of components. We assume that repositories with multiple components are special. Such a repository may be, for example, a 'zoo' [28]. Note that a single component can still imply the presence of multiple (dependent) projects.

Variants. This classifier applies when we locate different versions of the same project in a repository based on the analysis of project dependencies. We assume again that repositories with variants are special. Such a repository may capture, for example, versions in a migration process.

EMF's default is the use of Manifest files for defining OSGi projects and dependencies. We decided to only include homogeneous repositories using Manifest files for mining. The analysis of dependencies is based on 'Bundle-SymbolicName' elements in Manifest files. Listing 8 presents the rules for

inferring the occurrence of declarations (predicate `sl:decOccures`) and references (predicate `sl:refOccures`) and OSGi dependencies between Manifest files (predicate `sl:dependsOn`):

```
1  // Extraction of Bundle−SymbolicName declaration.
2  (?file, sl:elementOf, sl:Manifest) StrManifest(?file, "Bundle−SymbolicName", ?x)
3     ReplaceAllToUri(?x, "(;[^,]∗)|\\s", "", ?declaration) → // Replace all details.
4     (?file, sl:decOccurs, ?declaration).
5  // Extraction of Bundle−SymbolicName references.
6  (?file, sl:elementOf, sl:Manifest) StrManifest(?file, "Require−Bundle", ?x)
7     ReplaceAll(?x, '("[^"]∗)"', "", ?xi) // Replace all strings.
8     ReplaceAll(?xi, "(;[^,]∗)|\\s", "", ?references) → // Replace all details.
9     SplitToUri(?references, ?file, sl:refOccurs, ',').
10 // Creating dependency structure
11 (?a, sl:elementOf, sl:Manifest) (?b, sl:elementOf, sl:Manifest)
12    (?a, sl:decOccurs, ?deca) (?a, sl:refOccurs, ?decb) (?b, sl:decOccurs, ?decb) →
13    (?deca, sl:dependsOn, ?decb).
```

Listing 8. Rules for extracting OSGi declarations, references and dependencies.

The primitive `StrManifest(file, property, value)` is a specialized decomposition of a Manifest file, comparable to `StrJava` in Sect. 2.3; it binds `value` to a Manifest property in literal form. In the rules shown above, it binds `?x` to the required or defined bundles in string representation. The chains of `ReplaceAllToUri`, `ReplaceAll` and `SplitToUri` process `?x` in that it can be added to the model as declaration or reference. We exclude repositories with duplicated declarations (`sl:decOccurs`) for the same URI (classifier *Variants*). We apply an algorithm for detecting connected components to the `sl:dependsOn` triples, as inferred by the last rule shown above. We exclude repositories with multiple components.

The results of the selection steps are depicted in Fig. 3. In what follows, we only consider repositories with a single component, Manifest usage only, and no variants. We refer to these repositories as 'Vanilla *EMF* repositories'. There are 1438 such projects. These are the projects considered for mining below.

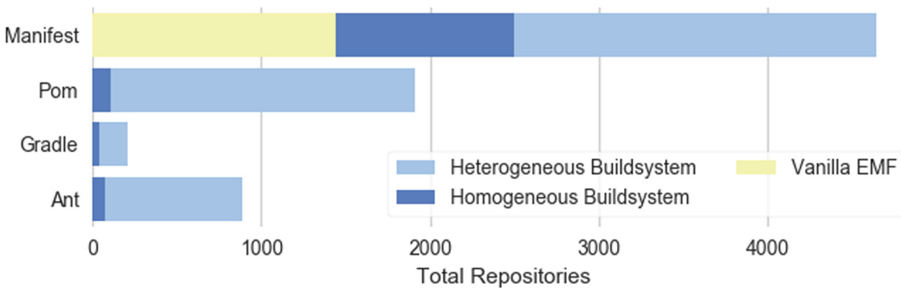


Fig. 3. Number of repositories with a particular build system further partitioned into homogeneous versus heterogeneous case.

4.3 Pattern Detection

For brevity, we only discuss here the detection of correspondence between Java and Ecore models; we show the decomposition of an Ecore model; we omit the rules for *Java* model detection in the set of available *Java* files; we also omit handling Ecore sub-packages.

Consider the beginning of a small Ecore sample file as follows:

```
1 <ecore:EPackage ... name="fsml" nsURI="http://www.softlang.org/metalib/emf/
      Fsml" nsPrefix="fsml">
2   <eClassifiers xsi:type="ecore:EClass" name="FSM">
3       ...
```

Listing 9. The first lines of a sample Ecore file.

The following rules decompose an Ecore model into root package and nested classifiers:

```
1 // Decomposition of the Ecore file into ...
2 (?x, sl:elementOf, sl:Ecore) → // ... the root package.
3    DecXml(?x, "/ecore:EPackage", sl:partOf,?x)
4    DecXml(?x, "/ecore:EPackage", sl:elementOf, sl:EcorePackageXMI).
5 (?x, sl:elementOf, sl:EcorePackageXMI) → // ... the nested classifiers in a package.
6    DecXml(?x, "/eClassifiers", sl:partOf, ?x)
7    DecXml(?x, "/eClassifiers", sl:elementOf, sl:EcoreClassifierXMI).
8 // Extracting URI and nsURI, necessary for detecting correspondence.
9 (?x, sl:elementOf, sl:EcorePackageXMI) →
10   UriXml(?x, ?x, sl:nsUri, "/@nsURI"). // NsUri for a package as URI.
11 (?classifier, sl:elementOf, sl:EcoreClassifierXMI)
12   (?classifier, sl:partOf, ?package) (?package, sl:nsUri, ?nsUri) // Get package's
         nsURI.
13   StrXml(?classifier, "/@name", ?classifierName) // Get the classifier's name as
         string.
14   UriConcat(?nsUri, '#//', ?classifierName, ?uri) → // Build a compound ?uri.
15   (?classifier, sl:uri, ?uri). // Uri for a classifier, i.e., nsURI with appended name.
```

Listing 10. Decomposing Ecore into classifiers appending a nsURI.

That is, a `sl:partOf` relationship is inserted along the nesting structure and fragments are classified by `sl:EcorePackageXMI` and `sl:EcoreClassifierXMI` respectively. This decomposition is handled by the first two rules using the `DecXml` to construct URIs in the repository referencing scheme. In contrast, the last two rules extract the attributes 'nsURI' and 'name' using `UriXml` and `StrXml`. The primitives return the recovered content directly as string or URI, as we need the actual attribute values 'FSM' (name) and 'http://www.softlang.org/metalib/emf/Fsml' (nsURI).

The following rules establish the correspondence between the elements of `sl:EcorePackageXMI` and `sl:EcorePackageJava` by matching the nsURI.

```
1 // Correspondence between XMI and Java Packages.
2 (?xmi, sl:elementOf, sl:EcorePackageXMI) (?java, sl:elementOf, sl:EcorePackageJava)
3    (?xmi, sl:nsUri, ?nsUri) (?java, sl:nsUri, ?nsUri) →
4    (?xmi, sl:correspondsTo, ?java).
5 // Correspondence between XMI and Java Classifiers.
6 (?xmiClassifier, sl:uri, ?uri) (?javaClassifier, sl:uri, ?uri)
7    (?xmiClassifier, sl:elementOf, sl:EcoreClassifierXMI)
8    (?javaClassifier, sl:elementOf, sl:EcoreClassifierJava) →
9    (?xmiClassifier, sl:correspondsTo, ?javaClassifier).
```

Listing 11. Rules recovering the correspondence.

The *Java* model extraction underlying the latter classification is based on accessing the nsURI property in the *Java* AST by a similar primitive `UriJava` (not shown here). Corresponding classifiers are aligned by comparing the nsURI concatenated with the classifier name.

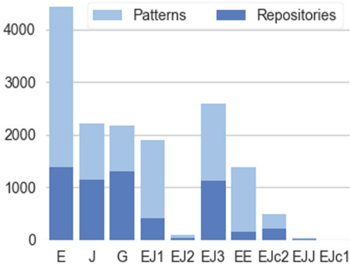


Fig. 4. Overall pattern sum and the number of repositories a pattern occurs in.

	E	J	G	EJ1	EJ2	EJ3	EE	EJc2	EJJ
Sum	4427	2217	2181	1894	96	2598	1376	496	45
Repo	1389	1152	1294	404	43	1127	157	223	18
mean	3.1	1.5	1.5	1.3	0.1	1.8	1.0	0.3	0.0
std	9.8	3.2	2.2	6.2	0.5	7.3	8.3	1.6	0.5
25%	1.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
50%	1.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
75%	2.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0
max	261	71	28	103	10	248	251	26	16
cv	3.2	2.1	1.5	4.7	8.2	4.0	8.6	4.5	15.2

Fig. 5. The distribution of detecting a pattern in the repositories (where the minimum is always 0.0). Row 'cv' lists the coefficient of variation.

4.4 Results

The results of the case study applied on 1438 Vanilla *EMF* repositories are shown in Figs. 4, 5 and 6. The discussion is not comprehensive. Overall, the online corpus features additional results. At the most basic level, we show numbers of repositories per pattern and numbers of pattern instances (Fig. 4). The median pattern occurrence of Ecore (E), Java (J) and Genmodel (G) packages and the regular correspondence (EJ3) in a repository is 1 (Fig. 5). This indicates that common usage is concerned with only one package. The coefficient of variation for measuring the relative variability 'cv' is the highest for EJ2, EE and EJJ — the first two patterns indicate problems; the latter pattern represents a very rare case (1% of the Vanilla repositories). We expect corner cases and problems to have

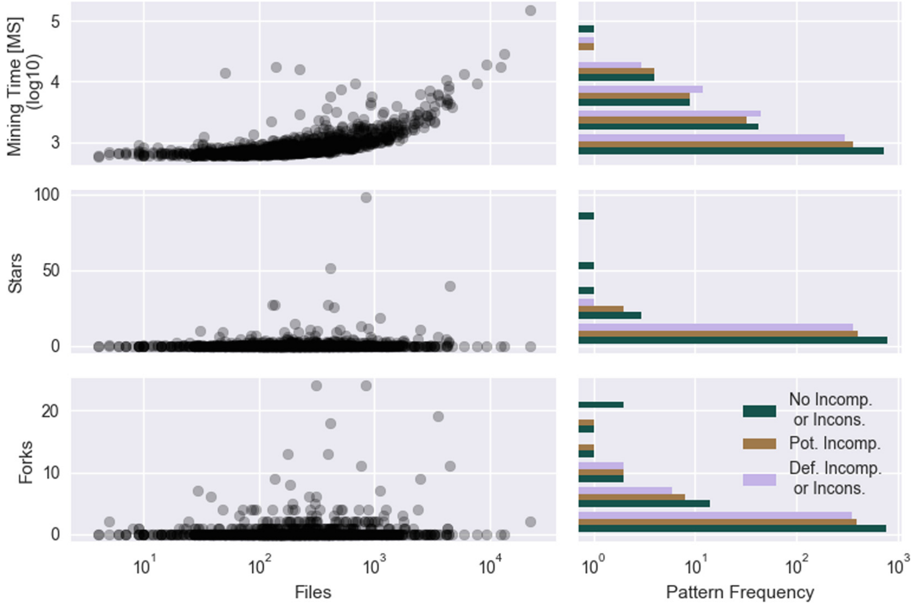


Fig. 6. Different repository and mining characteristics shown with respect to the amount of files in a repository. The right column shows how often different types of patterns occur for some histogram buckets and for the different characteristics.

high variations in pattern occurrence. Having no package correspondence (EJ1) or a regular package correspondence (EJ3) can both be considered as common usage. Forgetting to remove Java classifiers (EJc2) can also be considered a common usage despite being a definite inconsistency. On the contrary, we detected no repository missing a Java classifier for an Ecore classifier (EJc1). In Fig. 6, we examine the projects on scales for different characteristics (forks, stars, and mining time) to see how the size of projects relates to these characteristics and also how the relative frequency of pattern-based problems or the absence thereof relates to these characteristics. For instance, we can observe (right-bottom and right-middle charts in Fig. 6) that definite incompleteness and inconsistency is of much less or no concern with increasingly more forked or stared repositories.

5 Threats to Validity

The initialization and filtering of the repository list towards 'Vanilla EMF' can be seen as a threat to external validity. We might miss 'important' repositories and thereby produce biased results. Further, due to the complexity of EMF and the diversity of possible repository layouts, we might potentially miss particular cases in the rules. We extensively tested our rules in an instance- and parameter-based manner to cover this internal threat, but some rules are incomplete (e.g., regarding the considered build systems) or approximative (e.g., in assuming a

very strict naming scheme for generated classifiers). Our pattern catalog and the rating of patterns, i.e., *potential* or *definite incompleteness* or *inconsistency*, are potentially subjective, even though we have extensive experience with *EMF*.

6 Related Work

The reported research is original in terms of (i) leveraging an inference- and rule-based approach (as opposed to using any computational approach which would operate on more 'complete' representations) and (ii) developing a pattern catalog for EMF usage in repositories. However, there is related research in the broader areas of mining software repositories, program comprehension, and reverse engineering which we group accordingly below.

Pattern Detection. This may concern design or API-usage patterns. For instance, in [36], API-usage patterns are mined based on structural, semantic, and co-usage similarity for the accessed API methods and fields; in [34], logic metaprogramming is used to detect patterns in a logic layer on top of Java ASTs using JDT.

Classification of Artifacts. In [21], language usage trends are analyzed in 22 open source software projects by counting files with language-specific file extensions, e.g., '.py', and certain file-name patterns, e.g., 'README'. In [26], the use of different Eclipse-based MDE technologies is examined on *GitHub* repositories. The approach combines search based on technology-specific extensions and string-based content search, just like in our case study (Table 3). In [33], a large dataset of UML models is collected from *GitHub* by script-based inspection of downloaded images (e.g., '.jpg'), standard UML formats (e.g., '.uml') and tool specific formats (e.g., '.argo'). In [32], the number of languages used and their relatedness to each other is analyzed in a random set of 1150 *GitHub* repositories relying only on metadata and version history.

Linking Artifacts. In [41], traces between XML documents are analyzed using an imperative rule language and XML technology such as XPath. This work exceeds ours by considering references encoded in text fragments that are then mined using NLP-techniques. In [8], the distribution frequency at *GitHub* is empirically compared to CRAN for R packages. Further, package dependencies between projects' 'DESCRIPTION' files are mined to explore inter-repository dependency problems. In [30], a system's ground truth architecture is recovered by analyzing dependencies in the build configuration. To raise accuracy, the folder layout is considered. Arguably, our work relates to traceability recovery [7,15,22,31,38].

Fact Extraction. In [20], metrics are computed in multi-language repositories by reusing existing parser technology that is part of Eclipse. In [37], API usage in Java-based *GitHub* repositories is analyzed by parsing the code and resolving method calls to APIs. Their approach exceeds ours in using JDT type resolution which we may want to incorporate into our rule-based language model. In [1],

OO-specific ASTs (i.e., Java) are converted into RDF triples. This enables data extraction through SPARQL-queries on a triple store. In [39], a rule-based approach similar to ours is presented that relies on parsing code into a knowledge-discovery-model (KDM). It is used to mine dependencies in Java EE application. While we also employ AST structures, we rely on selective fact extraction as opposed to full materialization of the involved artifacts. In [11–13], the infrastructure, the domain specific language and applications of Boa, a framework for structured data extraction targeting repositories, is described. While the actual purpose and the surrounding infrastructure of our approach is highly comparable, the computation substantially differs in that Boa's fact extraction is not driven by previously inferred facts. The Boa language is compiled to a map-reduce framework to be executed in parallel. Such distribution is difficult to align with our rule-based inference mechanism.

Analysis of Changes. In [6], the frequency of code changes in Java-projects using Hibernate is traced while differentiating between data model, mapping, performance configurations, and query calls. In [23], the dependencies between projects considering build files specific to JavaScript, Ruby, and Rust are examined and their evolution is traced. In our future work, we will take version history into account.

7 Conclusion

In this paper, we have provided some insight into basic variation points and potential completeness and consistency problems with using EMF and detecting or maintaining such usage in repositories. We have used a rule-based approach to detect patterns of usage.

In future work, we would like to analyze evolution of repositories in terms of layout and pattern instances. Further, we would like to increase precision with regard to some aspects of correspondence, completeness, and consistency by integrating type resolution with Java classpath recovery. Moreover, we also work on a more profound generalization of referring to and accessing 'arbitrary' code or model elements across technological spaces: codename URA (unified resource accessor). Ultimately, we would like to move from (small) patterns of technology usage to inferring usage of more complex and modular megamodels for technology documentation [17].

References

1. Atzeni, M., Atzori, M.: CodeOntology: RDF-ization of source code. In: d'Amato, C., Fernandez, M., Tamma, V., Lecue, F., Cudré-Mauroux, P., Sequeda, J., Lange, C., Heflin, J. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 20–28. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_2
2. Berger, B.J., Sohr, K., Koschke, R.: Extracting and analyzing the implemented security architecture of business applications. In: Proceedings of the CSMR, pp. 285–294. IEEE (2013)

3. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, Birmingham (2013)
4. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003-2004. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005). https://doi.org/10.1007/11538097_3
5. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop (2004)
6. Chen, T., Shang, W., Yang, J., Hassan, A.E., Godfrey, M.W., Nasser, M.N., Flora, P.: An empirical study on the practice of maintaining object-relational mapping code in Java systems. In: Proceedings of the MSR 2016, pp. 165–176 (2016)
7. Cleland-Huang, J., Gotel, O., Zisman, A. (eds.): Software and Systems Traceability. Springer, Heidelberg (2012). https://doi.org/10.1007/978-1-4471-2239-5
8. ADecan, A., Mens, T., Claes, M., Grosjean, P.: When GitHub meets CRAN: an analysis of inter-repository package dependency problems. In: SANER, pp. 493–504 (2016)
9. Di Rocco, J., Di Ruscio, D., Härtel, J., Iovino, L., Lämmel, R., Pierantonio, A.: Systematic recovery of MDE technology usage. Springer, LNCS (2018)
10. Dittrich, K.R., Gatziu, S., Geppert, A.: The active database management system manifesto: a rulebase of ADBMS features. In: Sellis, T. (ed.) RIDS 1995. LNCS, vol. 985, pp. 1–17. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60365-4_116
11. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In: ICSE, pp. 422–431. IEEE Computer Society (2013)
12. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: ultra-large-scale software repository and source-code mining. ACM Trans. Softw. Eng. Methodol. **25**(1), 7:1–7:34 (2015)
13. Dyer, R., Rajan, H., Nguyen, H.A., Nguyen, T.N.: Mining billions of AST nodes to study actual and potential usage of java language features. In: ICSE, pp. 779–790. ACM (2014)
14. Favre, J.-M., Lämmel, R., Varanovich, A.: Modeling the linguistic architecture of software products. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 151–167. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_11
15. Galvão, I., Goknil, A.: Survey of traceability approaches in model-driven engineering. In: Proceedings of the EDOC, pp. 313–326. IEEE (2007)
16. Han, M., Hofmeister, C., Nord, R.L.: Reconstructing software architecture for J2EE web applications. In: Proceedings of the WCRE, pp. 67–79. IEEE (2003)
17. Härtel, J., Härtel, L., Lämmel, R., Varanovich, A., Heinz, M.: Interconnected linguistic architecture. Program. J. **1**(1), 3 (2017)
18. Hassan, A.E., Jiang, Z.M., Holt, R.C.: Source versus object code extraction for recovering software architecture. In: Proceedings of the WCRE, pp. 67–76. IEEE (2005)
19. Heinz, M., Lämmel, R., Varanovich, A.: Axioms of linguistic architecture. In: Proceedings of the MODELSWARD 2017 (2017)
20. Janes, A., Piatov, D., Sillitti, A., Succi, G.: How to Calculate software metrics for multiple languages using open source parsers. In: Petrinja, E., Succi, G., El Ioini, N., Sillitti, A. (eds.) OSS 2013. IAICT, vol. 404, pp. 264–270. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38928-3_20

21. Karus, S., Gall, H.C.: A study of language usage evolution in open source software. In: MSR, pp. 13–22. ACM (2011)
22. Keenan, E., Czauderna, A., Leach, G., Cleland-Huang, J., Shin, Y., Moritz, E., Gethers, M., Poshyvanyk, D., Maletic, J.I., Hayes, J.H., Dekhtyar, A., Manukian, D., Hossein, S., Hearn, D.: TraceLab: an experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In: Proc. ICSE, pp. 1375–1378. IEEE (2012)
23. Kikas, R., Gousios, G., Dumas, M., Pfahl, D.: Structure and evolution of package dependency networks. In: MSR, pp. 102–112 (2017)
24. Kniesel, G., Binun, A.: Standing on the shoulders of giants - a data fusion approach to design pattern detection. In: Proceedings of the ICPC, pp. 208–217. IEEE (2009)
25. Kniesel, G., Binun, A., Hegedüs, P., Fülöp, L.J., Chatzigeorgiou, A., Guéhéneuc, Y., Tsantalis, N.: DPDX-towards a common result exchange format for design pattern detection tools. In: Proceedings of the CSMR, pp. 232–235. IEEE (2010)
26. Kolovos, D.S., Matragkas, N.D., Korkontzelos, I., Ananiadou, S., Paige, R.F.: Assessing the use of Eclipse MDE technologies in open-source software projects. In: Proceedings of the MODELS, pp. 20–29 (2015)
27. Koschke, R.: Architecture reconstruction. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006-2008. LNCS, vol. 5413, pp. 140–173. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-95888-8_6
28. Kusel, A., Schoenboeck, J., Wimmer, M., Retschitzegger, W., Schwinger, W., Kappel, G.: Reality check for model transformation reuse: the ATL transformation zoo case study. In: Proceedings of the AMT 2013, volume 1077 of CEUR Workshop Proceedings. CEUR-WS.org (2013)
29. Lämmel, R., Varanovich, A.: Interpretation of linguistic architecture. In: Cabot, J., Rubin, J. (eds.) ECMFA 2014. LNCS, vol. 8569, pp. 67–82. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09195-2_5
30. Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidovic, N., Kroeger, R.: Comparing software architecture recovery techniques using accurate dependencies. In: Proceedings of the ICSE, pp. 69–78 (2015)
31. Mäder, P., Egyed, A.: Do developers benefit from requirements traceability when evolving and maintaining a software system? Empir. Softw. Eng. **20**(2), 413–441 (2015)
32. Mayer, P., Bauer, A.: An empirical analysis of the utilization of multiple programming languages in open source projects. In: Proceedings of the EASE, pp. 4:1–4:10 (2015)
33. Robles, G., Ho-Quang, T., Hebig, R., Chaudron, M.R.V., Fernández, M.A.: An extensive dataset of UML models in GitHub. In: Proc. MSR, pp. 519–522 (2017)
34. Roover, C.D.: A logic meta-programming foundation for example-driven pattern detection in object-oriented programs. In: Proceedings of the ICSM, pp. 556–561. IEEE (2011)
35. Roover, C.D., Lämmel, R., Pek, E.: Multi-dimensional exploration of API usage. In: Proceedings of the ICPC 2013, pp. 152–161. IEEE (2013)
36. Saied, M.A., Sahraoui, H.A.: A cooperative approach for combining client-based and library-based API usage pattern mining. In: Proceedings of the ICPC, pp. 1–10 (2016)
37. Sawant, A.A., Bacchelli, A.: A dataset for API usage. In: Proceedings of the MSR, pp. 506–509 (2015)

38. Seibel, A., Hebig, R., Giese, H.: Traceability in model-driven engineering: efficient and scalable traceability maintenance. In: Cleland-Huang, J., Gotel, O., Zisman, A. (eds.) Software and Systems Traceability, pp. 215–240. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2239-5_10
39. Shatnawi, A., Mili, H., El-Boussaidi, G., Boubaker, A., Guéhéneuc, Y., Moha, N., Privat, J., Abdellatif, M.: Analyzing program dependencies in java EE applications. In: Proceedings of the MSR (2017)
40. Stevens, R., Roover, C.D., Noguera, C., Kellens, A., Jonckers, V.: A logic foundation for a general-purpose history querying tool. Sci. Comput. Program. **96**, 107–120 (2014)
41. Zisman, A.: Using rules for traceability creation. In: Cleland-Huang, J., Gotel, O., Zisman, A. (eds.) Software and Systems Traceability, pp. 147–170. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2239-5_7

Towards Efficient Loading of Change-Based Models

Alfa Yohannis[1(✉)], Horacio Hoyos Rodriguez[1], Fiona Polack[2], and Dimitris Kolovos[1]

[1] Department of Computer Science, University of York, York, UK
{ary506,dimitris.kolovos}@york.ac.uk,
horacio_hoyos_rodriguez@ieee.org
[2] School of Computing and Maths, Keele University, Keele, UK
f.a.c.polack@keele.a.cuk

Abstract. This paper proposes and evaluates an efficient approach for loading models stored in a change-based format. The work builds on language-independent change-based persistence (CBP) of models conforming to object-oriented metamodelling architectures such as MOF and EMF, an approach which persists a model's editing history rather than its current state. We evaluate the performance of the proposed loading approach and assess its impact on saving change-based models. Our results show that the proposed approach significantly improves loading times compared to the baseline CBP loading approach, and has a negligible impact on saving.

1 Introduction

Conventional approaches for file-based model persistence in metamodelling architectures such as MOF [1] and EMF [2] are state-based – saving the current state of a model. In these approaches, version control and change detection are delegated to external systems. State-based persistence is computationally expensive, as a whole model must be saved and loaded; this can particularly affect large models and collaborative developments.

In [3], we proposed change-based persistence (CBP), an approach that persists the full sequence of *changes* made to a model instead of persisting the state. Compared to state-based approaches, CBP supports fast detection of changes, which can speed up model comparison and merging, as well as fast incremental model validation and transformation [4,5]. However, saving the change history of a model results in large, and ever-growing, CBP files. Loading times are also significant, as the loading process has to reconstruct a model's current state from its history [3]. This paper proposes and evaluates an approach that reduces CBP model loading time by avoiding the replaying of historical changes that have no impact on the final state of the model.

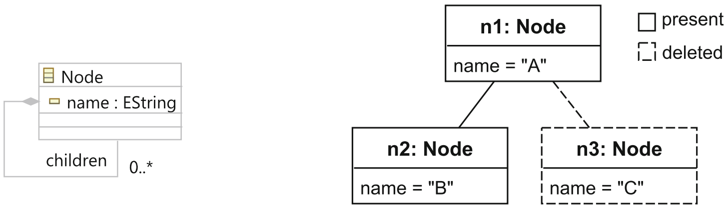
The rest of the paper is structured as follows. Section 2 introduces a running example and provides a brief introduction to CBP. Section 3 presents the approach to speed up model loading and its supporting data structures. Section 4

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 235–250, 2018.
https://doi.org/10.1007/978-3-319-92997-2_15

presents experimental results and evaluation. Section 5 provides an overview of related work, and Sect. 6 concludes with a discussion on directions of future work.

2 Running Example

To explain model CBP, we use a minimal tree metamodel and an example tree model in Fig. 1a and b. The metamodel is expressed in the Eclipse Modelling Framework (EMF) Ecore metamodelling language, the de-facto standard for object-oriented metamodelling. The example is contrived to avoid unnecessary repetition, whilst providing adequate coverage of the core features of Ecore (classes, single/multi-valued features, references). In this example, a tree model consists of named nodes which can – optionally – contain other nodes (*child* reference).



(a) The tree metamodel (EMF/Ecore).

(b) A tree model that conforms to the metamodel. Node n3 is created and then deleted.

Fig. 1. Running example of a metamodel and a conformant model.

The current state of the model in Fig. 1b has two nodes, $n1$, $n2$. The model was constructed by firstly creating the three nodes ($n1$, $n2$ and $n3$) and then nodes $n2$ and $n3$ were then added as children of $n1$. Finally, node $n3$ was deleted.

Listing 1. State-based tree model.

```
1  <Node id="n1" name="A">
2  <children id="n2" name="B"/>
3  </Node>
```

Listing 2. Change-based tree model.

```
1  create n1 of Node
2  set n1.name to "A"
3  create n2 of Node
4  set n2.name to "B"
5  create n3 of Node
6  set n3.name to "C"
7  add n2 to n1.children
8  add n3 to n1.children
9  remove n3 from n1.children
10 delete n3
```

Listing 1 shows the state-based representation of the model, using simplified XMI. Listing 2 shows the change-based representation, using the CBP syntax introduced in [3]. Lines 1–6 of Listing 2 record the creation and naming of the three nodes; lines 7–8 record the addition of $n2$ and $n3$ as children of $n1$; lines 9–10 capture the deletion of $n3$ (the *remove* command removes f $n3$ from its

container; the *delete* command completely removes $n3$ from its model). Changes in a CBP representation can be uniquely identified by their line numbers.

The example model history illustrates a case where earlier events (creating *n3* in line 5, naming it in line 6, making it a child of *n1* in line 8, removing it from the container in line 9) are superseded by a subsequent event (deletion of *n3* in line 10). Loading of the current model would arguably be faster if the events in lines 5, 6, 8, 9 and 10 could be ignored.

3 Towards Efficient Loading of Change-Based Models

The flowchart in Fig. 2 provides an overview of the editing lifecycle of a CBP model [3], with the proposed extensions shown as starred blocks. A model is loaded (1), edited (2) and saved (3). During editing, the changes made to the model are recorded in a memory-based data structure, serialised and with the latest events appended at the end (4). The change events are persisted into a CBP file every time the model is saved (5). When a model is re-loaded, the current model state is recreated by replaying the events stored in the CBP file (6).

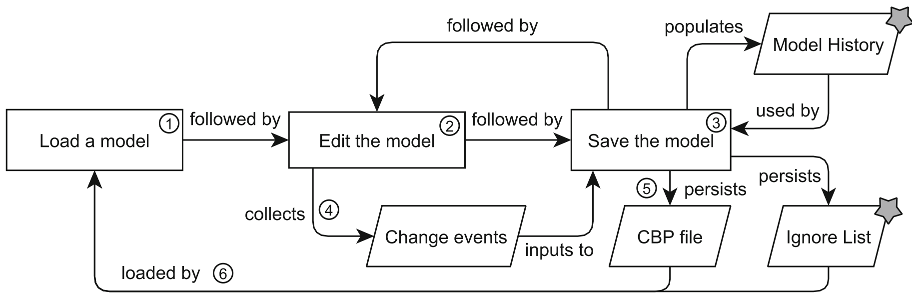


Fig. 2. CBP workflow, with optimised loading elements indicated by starred blocks.

A key principle of CBP is that the editing history is immutable, as this is essential for supporting incremental model management operations. As such, superseded events cannot be simply removed from the CBP file. Therefore, the proposed approach adds two artefacts: a in-memory $ModelHistory$ data structure which aggregates change events per model element, and an $IgnoreList$ file, which persists the position (i.e. line numbers) of superseded events so that the events can be ignored the next time the model is loaded. The Ignore List is saved alongside the CBP file. The rest of this section presents how the Model History is used to detect superseded events and generate the Ignore List.

3.1 Model History

The Model History data structure stores events and their line numbers in a CBP representation. The data can be used to reason about the events of a particular element and to determine which events are superseded. We refer to the line

number in the CBP representation as the *event number*. The proposed data structure is defined in Fig. 3 using a class diagram.

A *ModelHistory* has a *URI* attribute to identify the model for which it records changes. A *ModelHistory* can link to many *ElementHistory* objects, each identified by its *element* field which is queried from the model. An *ElementHistory* can link to many *FeatureHistories*, representing the editing histories of individual features – either references or attributes of the element. A *FeatureHistory* has a *type* (attribute or reference) and a *name*, identifying the feature.

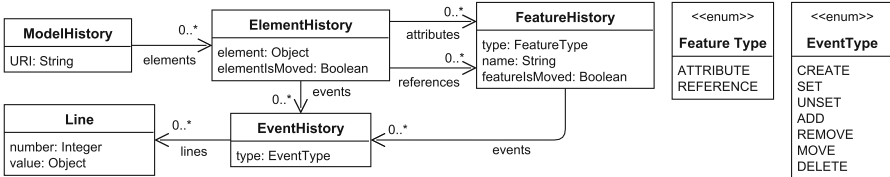


Fig. 3. The class model defining Model History.

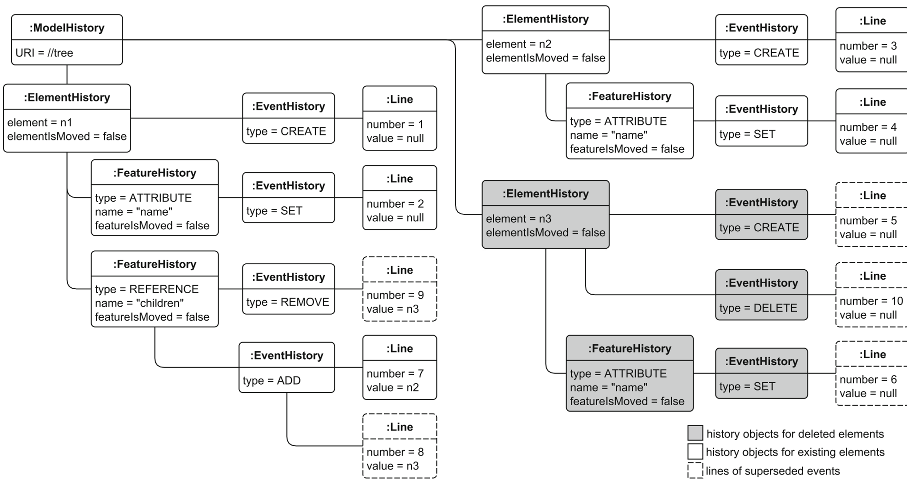


Fig. 4. The object diagram of the CBP model history in Listing 2.

An *EventHistory* represents series of events of the same type; it has an attribute *type* to identify the events' type and can have many *Lines*. A *Line* has a *number* attribute, to record the event number and a *value* that records the element involved in the event (Value is only used for events with types *ADD*, *REMOVE* and *MOVE*). Each *FeatureHistory* can have many *EventHistories*, to represent the events that modify the values of the features. Each *ElementHistory* can have many *EventHistories* to represent events that affect the state of the elements (life-cycle and relations to multivalued features). Figure 4 shows

an object diagram corresponding to the model in Fig. 3 that captures the model history shown in Listing 2. The grey rectangles are *History* objects related to the deleted node *n3*. The rectangles with the dashed outline are *Line* objects that represent superseded changes.

Next, we present the different strategies used to identify superseded events that will be added to the Ignore List.

3.2 Set and Unset Events

During the lifecycle of a model, a single-valued feature can have its value set (assigned) or unset many times. Each event is persisted, but only the last assigned value needs to be considered. For example, in Listing 3, the feature *name* is set to the value "A", unset, and finally set to the value "B". In the final state of the model, *n1.name* = "B". Thus, only line 4 is significant for the model's final state and therefore lines 2 and 3 can be ignored when loading the model. For a *set* event, all preceding *set* and *unset* events can be ignored, but for an *unset* event, all *set* and *unset* events can be ignored. Executing it does not have any effect on the final state of a model if all the preceding events also have been ignored.

Listing 3. A CBP representation of attribute *name* assignments.

```
1  create n1 of Node
2  set n1.name to "A"
3  unset n1.name
4  set n1.name to "B"
```

Listing 4. A CBP representation of attribute *name* assignments.

```
1  create n1 of Node
2  set n1.name to "A"
3  set n1.name to "B"
4  unset n1.name
```

Based on the Listing 3, our approach creates an instance of $ElementHistory$ $n1$ which contains an instance of $FeatureHistory$ *name*. The $FeatureHistory$ *name* consists of two $EventHistory$ instances, with types *SET* and $UNSET$ (the instances are named *set* and *unset* respectively for brevity). The *set* records the *Line* instances that hold the event numbers where the *set* events, and similarly for *unset*.

From Listing 3, we can thus infer that *name.set.lines* $= \{2, 4\}$ and *name.unset. lines* $= \{3\}$. The event numbers in both lists are used to determine that the events represented by lines 2 and 3 are superseded by that in line 4, which is a *set* event, giving an *ignoreList* $= \{2, 3\}$. By the same process, for Listing 4, we can reason that *name.set.lines* $= \{2,3\}$ and *name.unset.lines* $= \{4\}$. However, this case, the highest-numbered event is an *unset*, all so line numbers are put into the *ignoreList* (*ignoreList* $= \{2, 3, 4\}$) (*unset* event can be ignored along with all preceding *set* and *unset* events).

3.3 Add, Remove, and Move Events

For a multi-valued feature, add, remove, and move events can be called many times, to modify the feature. If an element is added to the feature, moved multiple times, and finally removed, then all the element's preceding events can be ignored, as long as the order of the feature's elements is not changed.

Listing 5 shows an example without a *move* event. In the Listing, nodes $n1$, $n2$, and $n3$ are added to the *children* feature of p (lines 5–7), In the latest state of the model, *children* only contains $n1$ and $n3$. As a result, the loading process could ignore the events that represent the *add* and *remove* events on $n1$.

Listing 5. A CBP of add and remove operations.

```
1  create p of Node
2  create n1 of Node
3  create n2 of Node
4  create n3 of Node
5  add n1 to p.children
6  add n2 to p.children
7  add n3 to p.children
8  remove n2 from p.
       children
```

Listing 6. A CBP representation of add, move, and remove operations.

```
1  create p of Node             //children=[]
2  create n1 of Node            //children=[]
3  create n2 of Node            //children=[]
4  create n3 of Node            //children=[]
5  add n1 to p.children         //children=[n1]
6  add n2 to p.children         //children=[n1,n2]
7  add n3 to p.children         //children=[n1,n2,n3]
8  move 0 to 1 in p.children    //children=[n2,n1,n3]
9  remove n2 from p.children    //children=[n1,n3]
```

To create the Ignore List for the Listing 5, we can deduce that $children.add.lines = \{\{5, n1\}, \{6, n2\}\ \{7, n3\}\}$ (5 is the line number and $n1$ is the value) and $children.remove.lines = \{\{8, n1\}\}$. Since $n2$ is removed from its containing feature (line 8), then executing its preceding add and remove events is unnecessary. Note that we retain the *create* event (line 3) as $n2$ has not been deleted from the model – only removed from its containing feature. We can iterate through the add and move structures to identify the events on $n2$ that should be removed, resulting in the $ignoreList = \{6, 8\}$.

Listing 6 shows an example with a *move* event[1]. A *move* event is inserted at line 8 thus makes the *remove* event of $n2$ moves to line 9. With the introduction of this *move* event, we now have the $children.add.lines = \{\{5, n1\}, \{6, n2\}\ \{7, n3\}\}$, $children.move.lines = \{\{8, n1\}\}$, and $children.remove.lines = \{\{9, n2\}\}$. In the final state of the model, the *children* should have the $n1$ and $n3$ in order, $children = [\text{n1, n3}]$.

However, executing the previous strategy naively leads to an erroneous final state. Using $ignoreList = \{6, 8\}$ produced by the naive strategy leads to a different order of $n1$ and $n3$ in the final state of the model where $children = [\text{n3, n1}]$ as shown by the naive optimised CBP in Listing 7. To overcome this problem, $*IsMoved$ flags in Fig. 3 is used to sign features and elements if they have been moved – the flags are set to *true*. If an element's $*IsMoved$ flag is true then all of its line numbers related to *add*, *move*, *remove* events cannot be put into the *ignoreList*. The flags are set to $false$ if the feature is empty.

Listing 7. A naive optimised CBP representation of original CBP representation in Listing 6 .

```
1  create p of Node           // children = []
2  create n1 of Node          // children = []
3  create n2 of Node          // children = []
4  create n3 of Node          // children = []
5  add n1 to p.children       // children = [n1]
6  add n3 to p.children       // children = [n1, n3]
7  move 0 to 1 in p.children  // children = [n3, n1]
```

[1] The commented parts show the end states of *children* after each event.

3.4 Create and Delete Events

When an element is deleted, it is completely removed from the model. Therefore, all previous events (*create*, *set*, *unset*, *move*, *add*, *remove*, *delete*) on features of element can be ignored, along with all events on the element's features. For example, when node *n3* in Listing 2 is deleted, the events in lines 5–6 and 8–10 are superseded. If the Listing 2 is optimised – some of its events are ignored – when loading, it runs as if the Listing 8 are executed.

Listing 8. Change-based representation of the model in Fig. 1b after removal of node $n3$.

```
1  create n1 of Node
2  set n1.name to "A"
3  create n2 of Node
4  set n2.name to "B"
5  add n2 to n1.children
```

Using the Listing 2, we can construct the structure of histories that are related to element $n3$ as follows: $n3.create.lines = \{5\}$, $n3.name.set.lines = \{6\}$, $n1.children.add.lines = \{\{7,\ n2\},\ \{8,\ n3\}\}$, $n1.children.remove.lines = \{\{9, n3\}\}$, and $n3.delete.lines = \{10\}$. Thus, when element $n3$ is deleted, by iterating through all these history structures, all line numbers associated with $n3$ can be identified and added to *ignoreList* producing $ignoreList = \{5\ 6, 8, 9, 10\}$ so they can be ignored in the next model loading.

4 Performance Evaluation

We developed the proposed efficient loading approach on top of the original CBP implementation[2] from [3] and evaluated our approach's model loading performance, as well as its memory footprint and its impact on the time required to save changes made to CBP models. The evaluation was performed on Intel® Core™ i7-6500U CPU@2.50 GHz 2.59 GHz, 12 GB RAM, and the Java™ SE Runtime Environment (build 1.8.0_162-b12).

Given that CBP is a very recent contribution and we are not aware of any existing datasets containing real-world models expressed in a change-based format, we have used synthetic change-based models for the evaluation of our experiments. The synthetic models were derived from real-world cases: the BPMN2 [6,7] and Epsilon [8,9] software projects, and the United States article [10] on Wikipedia (the article is further referred as Wikipedia). For the first two projects, for each version of the cases, we used MoDisco [11] to generate a UML2 [12] model that reflects its source code. For the Wikipedia article, a model that conforms to the Modisco XML metamodel [13] was generated. Since these cases have many versions – represented by commits/revisions, different models of the versions can be generated, and to some degree, they reflect the time-ordered changes of the cases. The synthetic change-based model for each case was derived by

[2] The prototype, tests, and data used in the evaluation are available under https://github.com/epsilonlabs/emf-cbp and https://goo.gl/1zUBQC for reproducibility.

comparing an initially-empty running model to different versions of the case's models sequentially. All identified differences were then reconciled by performing a unidirectional merging to the running model. All changes made to the running model during the merging process were captured and persisted into a CBP file. EMF Compare was used [14] to perform the comparison and merging.

Using the synthetic models, we performed performance evaluation on loading time, saving time, and memory footprint for both loading and saving. To compare the loading time, we ran the optimised and original (baseline) CBP algorithms to reconstruct the current state of each of the three models (the results are shown in Fig. 5). As discussed in Sect. 3, optimised CBP also does extra work when saving the changes to a model, in order to save time (relative to original CBP) when loading a model. To analyse the performance effect of optimisation activities, we, therefore, compared the overall time required to save a new version of the models described above, after one single change has been made (The results are shown in Fig. 6). We also compare the memory footprints for both loading and saving since the optimised CBP approach also requires the maintenance of an additional in-memory data structure that keeps track of element and feature editing histories (see Figs. 7 and 8 for the results).

For each combination of dimensions (loading time, saving time, loading memory footprint, saving memory footprint), persistence types (original CBP, optimised CBP, and XMI), and cases (BPMN2, Epsilon, and Wikipedia), we performed measurement 22 times. The results of the measurement enabled us to perform the Welch's t-test [15] to find the significance of the comparisons for each case. We used a significance level of 5%. If t-test' *p-value* < 0.05, we rejected the null hypothesis – the *means* of the compared persistence types are equal (H_0) – and accepted the alternative hypothesis – the *means* of the compared persistence types are not equal (H_1).

For loading and saving time, we measured the delta time required to complete the loading and saving. For memory footprint, we measured the delta of memory used before and after loading and saving completes. The results are presented below.

4.1 Data Description

Table 1 summarises events, elements and saved versions for the Epsilon, BPMN2, and Wikipedia cases. *Total Events* is the numbers of events that were produced by our approach in generating a change-based model for each case. *Ignored Events* is the number of superseded events that do not need to be replayed when reloading the models. *Elements* is the number of elements contained in each model. *Total Versions* is the number of commits/revisions made to the cases, taken from the git repositories or Wikipedia at the time this evaluation performed. *Processed Versions* is the number of commits/revisions that were processed to produce change-based models: since the comparison between versions takes considerable time, not all versions are processed here.

Table 1. Description of change-based models generated for evaluation.

Model	Total events	Ignored events	Elements	Total versions	Processed versions
BPMN2	1.2 million	1.1 million	62,062	192	192 (100.0%)
Epsilon	2.6 million	1.8 million	79,459	3,037	727 (23.9%)
Wikipedia	11.5 million	7.8 million	12,144	37,996	3,100 (8.2%)

4.2 Model Loading Time

This subsection presents the results of the loading time measurement of change-based models for each pair of the persistence types and cases, and the t-test results of their comparisons (Table 2 and Fig. 5).

Table 2. The t-test results of loading time comparison between original CBP (CBP), optimised CBP (OCBP), and XMI.

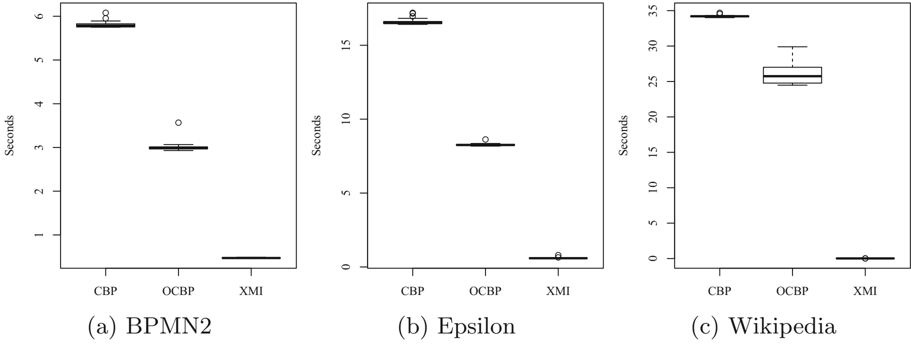
Group	Mean	SD	Comparison	t	df	p-value
BPMN2 Load Time (s)			*BPMN2 Load Time*			
CBP	5.81	0.08	CBP vs. XMI	315.95	21.46	<0.05
OCBP	3.02	0.13	CBP vs. OCBP	87.67	35.10	<0.05
XMI	0.47	0.47	OCBP vs. XMI	93.86	21.18	<0.05
Epsilon Load Time (s)			*Epsilon Load Time*			
CBP	16.60	0.23	CBP vs. XMI	324.18	22.78	<0.05
OCBP	8.28	0.09	CBP vs. OCBP	160.06	27.48	<0.05
XMI	0.60	0.05	OCBP vs. XMI	354.52	42.06	<0.05
Wiki Load Time (s)			*Wikipedia Load Time*			
CBP	34.23	0.145	CBP vs. XMI	1,110.10	21.00	<0.05
OCBP	26.14	1.583	CBP vs. OCBP	23.90	21.35	<0.05
XMI	0.02	0.001	OCBP vs. XMI	77.37	21.00	<0.05

$Mean$ = average, SD = standard deviation, t = t-test's *t-value*, *df* = degree of freedom, *p-value* = significance, s = the unit is seconds

These loading times show a considerable time saving for optimised CBP: BPMN2 was 48.02% faster, Epsilon 50.12% faster, and the Wikipedia page 23.63% faster than in the original CPB implementation (all optimised CBP's *means* are smaller than all original CBP's *means*), which has a positive correlation to the number of ignored events. All the t-test results also show that loading times for all the persistence types are significantly different (all the *p-values* < 0.05).

For reference, we also compare CBP loading with the execution time for loading the equivalent state-based model in XMI. Figure 5 shows that, even with

the improvements delivered by the new algorithm, loading change-based models is still significantly slower than loading a state-based model (all XMI's means are smaller than other persistence types' means).



(a) BPMN2 (b) Epsilon (c) Wikipedia

Fig. 5. Results for loading a model in original CBP (CBP), optimised CBP (OCBP), and for loading a state-based (XMI) representation.

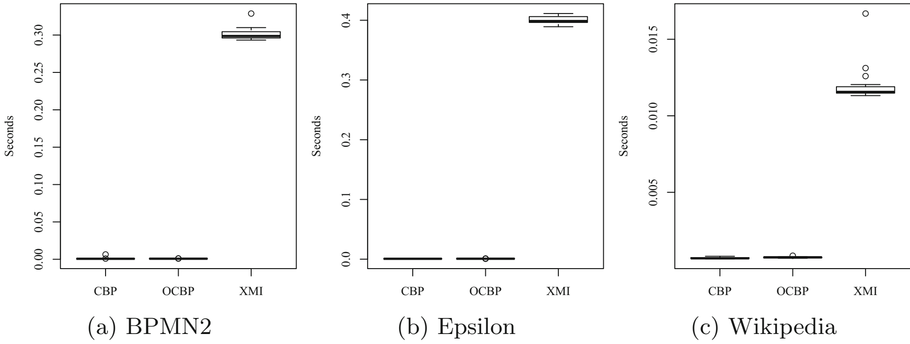
4.3 Model Saving Time

This subsection presents the results of the saving time measurement of change-based models for each pair of the persistence types and cases, and the t-test results of their comparisons (Table 3 and Fig. 6). As discussed in [3], CBP loading

Table 3. The t-test results of saving time comparison between original CBP (CBP), optimised CBP (OCBP), and XMI.

Group	Mean	SD	Comparison	t	df	p-value
BPMN2 Save Time (s)			*BPMN2 Save Time*			
CBP	0.00097	123e−5	CBP vs. XMI	−175.58	22.01	<0.05
OCBP	0.00081	12e−5	CBP vs. OCBP	0.62	21.38	0.54
XMI	0.30122	793e−5	OCBP vs. XMI	−177.76	21.01	<0.05
Epsilon Save Time (s)			*Epsilon Save Time*			
CBP	0.00069	3.4e−5	CBP vs. XMI	−6.01	21.00	<0.05
OCBP	0.00080	8.0e−5	CBP vs. OCBP	160.06	28.24	<0.05
XMI	0.40025	595e−5	OCBP vs. XMI	−314.80	21.01	<0.05
Wiki Save Time (s)			*Wikipedia Save Time*			
CBP	0.00071	4.9e−5	CBP vs. XMI	−46.19	21.08	<0.05
OCBP	0.00075	4.1e−5	CBP vs. OCBP	−3.48	40.77	<0.05
XMI	0.01195	114e−5	OCBP vs. XMI	−46.01	21.06	<0.05

Mean = average, *SD* = standard deviation, *t* = t-test's *t-value*, *df* = degree of freedom, *p-value* = significance, *s* = the unit is seconds



(a) BPMN2 (b) Epsilon (c) Wikipedia

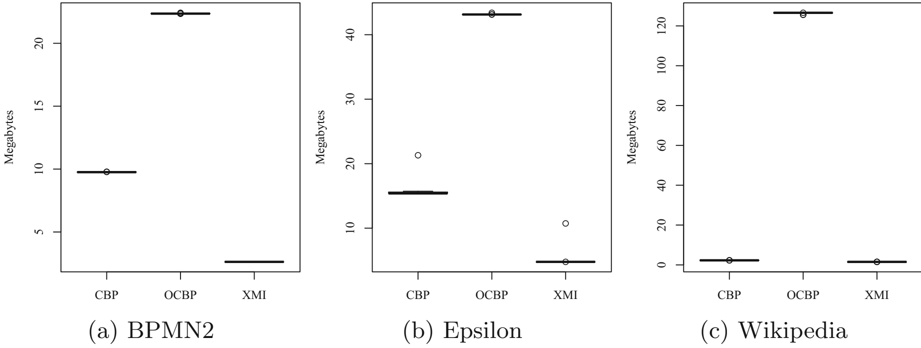
Fig. 6. A comparison on time required for persisting an event between original CBP (CBP), optimised CBP (OCBP), and XMI.

time penalties are balanced against the benefits that CBP brings, in terms of persisting changes (saving time).

As shown in Table 3 and Fig. 6, the performance of the two CBP implementations is not very different. Since the significance level is 5%, only the BPMN2 case that fails. However, the difference between the *means* of its original CBP (0.97 ms) and optimised CBP (0.81 ms) is small. This indicates that the cost of the extra work in the optimised CBP algorithm is negligible. On the other hand, both CBP implementations are significantly faster at saving changes than state-based XMI (the *means* of both CBP implementations are smaller than XMI's *means*, and both CBP implementations have *p-values* < 0.05 when compared to XMI). This is expected, as the CBP implementations only need append the last changes to the existing model file (their performance is thus relative to the number of changes since the last save), while the XMI implementation needs to reconstruct an XML document for the entire state of the model, and replaces the contents of the model file every time (and hence its performance is relative to the size of the entire model).

4.4 Memory Footprint

Here we present the results of measuring the memory footprint after loading models (Table 4 and Fig. 7) and persisting single changes (Table 5 and Fig. 8) using the models from the three cases. The results show the significant memory overhead of the extra data structure when loading models (all the *means* of optimised CBP are greater than all the *means* of original CBP and all comparisons between both CBPs show *p-values* < 0.05, Table 4). Both CBPs are also outperformed by XMI in terms of memory footprint when loading models (all the *means* of XMI are smaller than all the *means* of both CBPs and all comparisons against XMIs show all *p-values* < 0.05, Table 4). In loading, XMI uses significantly less memory than the optimised CBP representation and performs slightly better than the original CBP.



(a) BPMN2 (b) Epsilon (c) Wikipedia

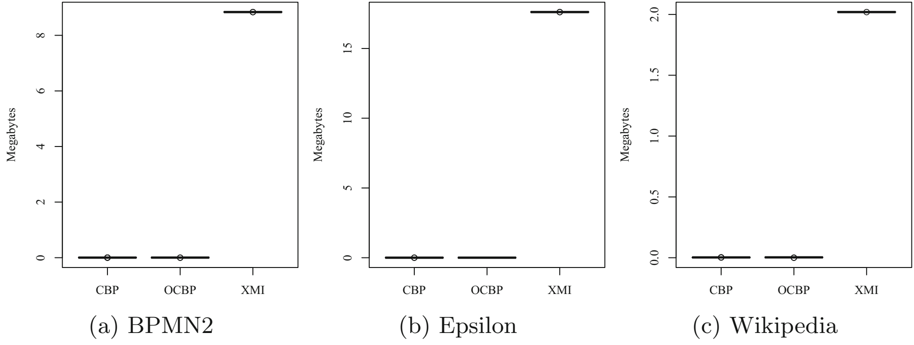
Fig. 7. A comparison on memory footprint after loading a model between original CBP (CBP), optimised CBP (OCBP), and XMI.

Table 4. The t-test results of memory footprint comparison after loading a model between original CBP (CBP), optimised CBP (OCBP), and XMI.

Group	Mean	SD	Comparison	t	df	p-value
BPMN2 Load Memory (M)			*BPMN2 Load Memory*			
CBP	9.76	76.0e−4	CBP vs. XMI	4,392.5	21.22	<0.05
OCBP	22.36	0.015	CBP vs. OCBP	−3,695.7	32.28	<0.05
XMI	2.63	5.5e−4	OCBP vs. XMI	6,572.4	21.06	<0.05
Epsilon Load Memory (M)			*Epsilon Load Memory*			
CBP	15.74	1.248	CBP vs. XMI	28.16	41.99	<0.05
OCBP	43.15	0.056	CBP vs. OCBP	−102.9	21.08	<0.05
XMI	5.05	1.271	OCBP vs. XMI	140.49	21.08	<0.05
Wiki Load Memory (M)			*Wikipedia Load Memory*			
CBP	2.29	2.4e−4	CBP vs. XMI	4,523.5	25.16	<0.05
OCBP	126.48	0.29	CBP vs. OCBP	−2,009.3	21.00	<0.05
XMI	1.52	7.6e−4	OCBP vs. XMI	2,021.8	21.00	<0.05

Mean = average, *SD* = standard deviation, *t* = t-test's *t-value*, *df* = degree of freedom, *p-value* = significance, *M* = the unit is megabytes

In terms of saving, both CBP implementations persist a single change faster than XMI indicated by their *means* that are smaller than the *means* of XMI, and all the CBPs' t-tests with XMI show that their differences are significant at *p-value* < 0.05 (Table 5). The optimised CBP has a larger memory footprint than the original CBP since the means of the optimised CBP for all cases are greater than the means of the original CBP. However, their memory footprints are not very different. Even though the BPMN2 and Epsilon cases have *p-values* < 0.05, the differences of the *means* of their original and optimised CBPs are small, and the Wikipedia case also shows *p-value* > 0.05 on its original CBP vs. optimised CBP comparison.



(a) BPMN2 (b) Epsilon (c) Wikipedia

Fig. 8. A comparison on memory footprint after persisting an event between CBP, optimised CBP, and XMI.

Table 5. The t-test results of memory footprint comparison after saving an event between original CBP (CBP), optimised CBP (OCBP), and XMI.

Group	Mean	SD	Comparison	t	df	p-value
BPMN2 Save Memory (M)			*BPMN2 Save Memory*			
CBP	0.0023	6.3e−5	CBP vs. XMI	−489,170	41.49	<0.05
OCBP	0.0029	80e−5	CBP vs. OCBP	−3.22	21.26	<0.05
XMI	8.84	5.6e−5	OCBP vs. XMI	−51,180	21.21	<0.05
Epsilon Save Memory (M)			*Epsilon Save Memory*			
CBP	0.0025	18.8e−6	CBP vs. XMI	−4.3e+6	21.00	<0.05
OCBP	0.0031	279.9e−6	CBP vs. OCBP	−10.131	21.19	<0.05
XMI	17.61	2.4e−6	OCBP vs. XMI	−295,090	21.00	<0.05
Wiki Save Memory (M)			*Wikipedia Save Memory*			
CBP	0.0025	1.9e−5	CBP vs. XMI	−391,970	40.52	<0.05
OCBP	0.0028	84.1e−5	CBP vs. OCBP	−1.75	21.02	0.094
XMI	2.0194	1.5e−5	OCBP vs. XMI	−11,245	21.01	<0.05

Mean = average, *SD* = standard deviation, *t* = t-test's *t-value*, *df* = degree of freedom, *p-value* = significance, *M* = the unit is megabytes

4.5 Threats to Validity and Limitations

In this work, we have only tested the algorithms on synthesised models which may not be representative of the complexity and interconnectedness of models in other domains. Diverse characteristics of models in different domains can affect the effectiveness of the algorithm and therefore yield different outcomes. So far, CBP optimisation only supports ordered and unique features. Support for duplicate values means that removal of an item does not necessarily result in the item not being present in the feature value. Additional information must be captured to persist the number of copies and positions of the feature members to properly generate the ignore list.

4.6 Discussion

For the original CBP loading, the total time required to load a model is $T_{CBP} = T_E + T_O$, where T_E is the total time required to complete executing all events, and T_O is the total time needed to complete other required routines (e.g. initialisation, reading files). For the optimised CBP loading, the total time to load a change-based model is reduced by the total time saved-up by ignoring superseded events T_I, that is $T_{OCBP} = T_E + T_O - T_I$. Thus, it is expected that optimised CBP can load a model faster than original CBP. This statement is in accordance with our finding in Sect. 4.2 that the total saved-up loading time corresponds to the number of ignored events. However, it still requires more investigation to determine the degree of their correlation, which will be addressed in our future work.

5 Related Work

There are several non-XMI approaches to state-based model persistence, using relational or NoSQL databases. For example, EMF Teneo [16] persists EMF models in relational databases, while Morsa [17] and NeoEMF [18] persist models in document and graph databases, respectively. None of these approaches provides built-in support for versioning and models are eventually stored in binary files/folders which are known to be a poor fit for text-oriented version control systems like Git and SVN. Connected Data Objects (CDO) [19], provides support for database-backed model persistence as well as collaboration facilities, but its adoption necessitates the use of a separate version control system in the software development process (e.g. a Git repository for code and a CDO repository for models), which introduces fragmentation and administration challenges [20]. Similar challenges arise in relation to other model-specific version control systems such as EMFStore [21].

6 Conclusions and Future Work

This paper proposes an efficient algorithm and supporting data structures for loading change-based models. Performance is evaluated on synthesised models, with comparison against the existing change-based implementation, and state-based XMI. Our results show considerable savings in terms of loading time with a negligible impact on saving time, but at the cost of a higher memory footprint. In future, we intend to evaluate CBP against state-based persistence on real complex models. We also plan to investigate the impact of change-based model persistence on the performance of change detection, model merging, and conflict resolution in the context of collaborative modelling. Meanwhile, the CBP approach can be further optimised to consume less memory and speed up parsing, such as using binary format instead of text. We are also exploring a hybrid persistence representation that offers a combination of state-based and change-based persistence.

Acknowledgements. This work was partly supported by through a scholarship managed by *Lembaga Pengelola Dana Pendidikan Indonesia* (Indonesia Endowment Fund for Education).

References

1. OMG: Metaobject facility. http://www.omg.org/mof. Accessed 21 Feb 2018
2. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Eclipse Series. Pearson Education, London (2008)
3. Yohannis, A., Polack, F., Kolovos, D.: Turning models inside out. In: Proceedings of the 3rd Workshop on Flexible Model Driven Engineering Co-located with ACM IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2017) (2017)
4. Ráth, I., Hegedüs, Á., Varró, D.: Derived features for EMF by integrating advanced model queries. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 102–117. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_10
5. Ogunyomi, B., Rose, L.M., Kolovos, D.S.: Property access traces for source incremental model-to-text transformation. In: Taentzer, G., Bordeleau, F. (eds.) ECMFA 2015. LNCS, vol. 9153, pp. 187–202. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21151-0_13
6. Eclipse: MDT/BPMN2. http://wiki.eclipse.org/MDT/BPMN2. Accessed 15 Jan 2018
7. Eclipse: BPMN2 git. https://git.eclipse.org/c/bpmn2/org.eclipse.bpmn2.git/. Accessed 19 Feb 2018
8. Eclipse: Epsilon. https://www.eclipse.org/epsilon/. Accessed 12 Feb 2018
9. Eclipse: Epsilon git. https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git. Accessed 19 Feb 2018
10. Wikiedia: United States. https://en.wikipedia.org/wiki/United_States. Accessed 19 Feb 2018
11. Brunelire, H., Cabot, J., Dup, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014)
12. Eclipse: MDT/UML2. http://wiki.eclipse.org/MDT/UML2. Accessed 15 Jan 2018
13. Eclipse: Xml metamodel. http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.modisco.xml.doc%2Fmediawiki%2Fxml_metamodel%2Fuser.html. Accessed 19 Feb 2018
14. Eclipse: EMF Compare. https://www.eclipse.org/emf/compare/. Accessed 15 Jan 2018
15. Welch, B.L.: The generalization of 'student's' problem when several different population variances are involved. Biometrika **34**(1/2), 28–35 (1947)
16. Eclipse: Teneo. http://wiki.eclipse.org/Teneo. Accessed 15 Oct 2017
17. Espinazo-Pagán, J., Sánchez Cuadrado, J., García Molina, J.: Morsa: a scalable approach for persisting and accessing large models. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 77–92. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_7
18. Daniel, G., Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., Cabot, J.: NeoEMF: a multi-database model persistence framework for very large models. In: Proceedings of the MoDELS 2016 Demo and Poster Sessions Co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, 2–7 October 2016, pp. 1–7 (2016)

19. Eclipse: CDO the model repository. https://www.eclipse.org/cdo/. Accessed 15 Oct 2017
20. Barmpis, K., Kolovos, D.S.: Evaluation of contemporary graph databases for efficient persistence of large-scale models. J. Object Technol. **13**(3), 3:1–26 (2014)
21. Koegel, M., Helming, J.: EMFStore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, Cape Town, South Africa, 1–8 May 2010, vol. 2, pp. 307–308 (2010)

Towards a Framework for Writing Executable Natural Language Rules

Konstantinos Barmpis[1(✉)], Dimitrios Kolovos[1], and Justin Hingorani[2]

[1] Department of Computer Science, University of York, York, UK
{konstantinos.barmpis,dimitris.kolovos}@york.ac.uk
[2] JC Chapman Ltd., Suite 4, 12 Jenner Avenue, London W3 6EQ, UK
justin@jcchapman.com

Abstract. The creation of domain-specific data validation rules is commonly performed by the relevant domain experts. Such experts are often not acquainted with the low-level technologies used to actually execute these rules and will hence document them in some informal form, such as in natural language. In order to execute these rules, they need to be transformed by technical experts into a relevant executable language, such as SQL. The technical experts in turn are often not familiar with the business logic these rules are depicting and will thusly have to collaborate with the business experts to gain insight into the semantics of the rules. This paper presents an approach for writing financial data validation rules in constrained natural language, that can then be automatically transformed and executed against the data they are referring to. In order to achieve this, we use the Xtext framework for creating the editor where business experts can create their rules that can then be transformed into executable constraints. We evaluate this approach in terms of its extensibility, coverage and verboseness with respect to the business rules sent to specific UK banks submitting data under one of the Bank of England's annual reviews.

1 Introduction

Organizations will commonly communicate their policies in natural language, be it internally to their staff and stakeholders or externally to interested parties. As natural language is inherently vague, for achieving consistency and amenability to computer-based processing it needs to be either written in or converted to a formal notation. A common approach is to introduce domain experts (in the domain the data these policies are written against is stored) to convert the natural language documents into the appropriate executable form. This introduces another level of risk as the domain experts will have to interpret these documents, introducing formal meaning to an inherently informal description. As such, the domain experts may have to consult the business experts themselves in order to gain a better understanding, leading to a large increase in both company resources used as well as error-prone cross-domain knowledge transfer.

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 251–263, 2018.
https://doi.org/10.1007/978-3-319-92997-2_16

Another approach is to write the policies themselves in a form more amenable to automation, but also retaining the ability for them to be written by business experts. In this paper we introduce a constraint natural language (CNL) for expressing constraints providing the benefits of machine-readable content whilst also being closely resemblant of natural language itself. This allows data validation rules to be written by non-technical stakeholders without the need for the technical experts; this allows each expert to focus on their respective fields, avoiding any additional risk.

The remainder of the paper is structured as follows: Sect. 2 discusses tools and methodologies of a CNL-based approach to business rules and Sect. 3 introduces the Open Rules Platform (ORP), a framework for writing validation rules in CNL. ORP is a commercial product from JC Chapman[1] that was produced as part of a knowledge transfer partnership (KTP) program[2] in collaboration with the University of York. Section 4 discusses the results obtained by using ORP in an industrial use-case and finally Sect. 5 concludes and mentions future lines of work.

2 Background and Related Work

The development and use of constrained natural language, also referred to as controlled natural language or controlled language, has been extensively investigated over the past decade. This section presents the main state-of-the-art practices and technologies and discusses the approach taken by the Open Rules Platform. As it is assumed that the reader is familiar with model-driven engineering practices like model transformation and domain-specific languages and editors, such information is omitted.

2.1 Constrained Natural Language

One of the most popular standards used to create such business rules in constrained natural language is the Semantics of Business Vocabulary and Business Rules (SBVR). This specification by the Object Management Group (OMG) covers two aspects: Vocabulary (natural language ontology) and Rules (elements dictating policy) [1]. Rules are composed of facts that rely on concepts which are made up of terms. Each term expresses a business concept and a fact can make assertions regarding this concept. Since SBVR does not use any specific language to express these concepts in a concrete fashion (it uses the notion of a "semantic formulation" to describe structure), it is left to the creator to decide the scope and expressiveness of any language conforming to this standard. As SBVR supports both formal and informal expressions, covering both aspects of the formalist vs naturalist approach to constrained natural language [2], it is down to the SBVR-based languages to decide which way they lean towards. For

[1] http://www.jcchapman.com/.
[2] http://ktp.innovateuk.org/.

example SBVR Structured English leans heavily towards the formal side whilst RuleSpeak[3] leans towards a more natural form of constrained natural language [1].

There are various tools with languages conforming to the SBVR standard, such as RuleCNL [3] and others [4–6], that vary in their expressiveness and ease of use. As good as SBVR is at introducing structure to constrained natural languages, its inherent complexity means that for smaller dialects the overhead of following the standard may overshadow its usefulness. As such, even though the Open Rules Platform is heavily inspired by SBVR as well as languages using it, it does not formally abide by the standard, instead deciding to keep a more minimalist language metamodel, more amenable to extension.

3 Executable Natural Language Rules

This section presents the architecture and design of the OR platform. Since the platform uses multiple tools and technologies, we briefly introduce them, focusing on how each technology contributes to the system. Finally we discuss the various metamodels used by the OR platform, in order to provide more insight into how the system behaves.

3.1 Architecture

Figure 1 shows an overview of the OR platform and below we detail the technologies used:

- Eclipse Modeling Framework (EMF) [7]. This framework is used to facilitate the creation of meta-models that encode the abstract syntax of the developed CNL.
- Xtext [8]. This framework is used to define a textual concrete syntax for the CNL. It offers various (semi-)automatically generated artefacts such as a rich editor for writing statements in this syntax as well as an API for this functionality, which can be used to create web-based editors offering similar capabilities.
- Epsilon [9]. This framework is used to transform CNL models into executable representations (e.g. SQL, EVL). Epsilon comprises a family of languages for performing various model management operations, underpinned by a common model connectivity layer that can access various modeling technologies (like EMF, relational databases or spreadsheets).
- CNL/Mapping metamodels. These metamodels (defined in EMF) are provided to the CNL Xtext parser in order to provide the structure for generating rules models as well as a mapping model from the relevant CNL documents. More details on these artefacts are given in Sect. 3.2.

[3] http://www.rulespeak.com/en/.

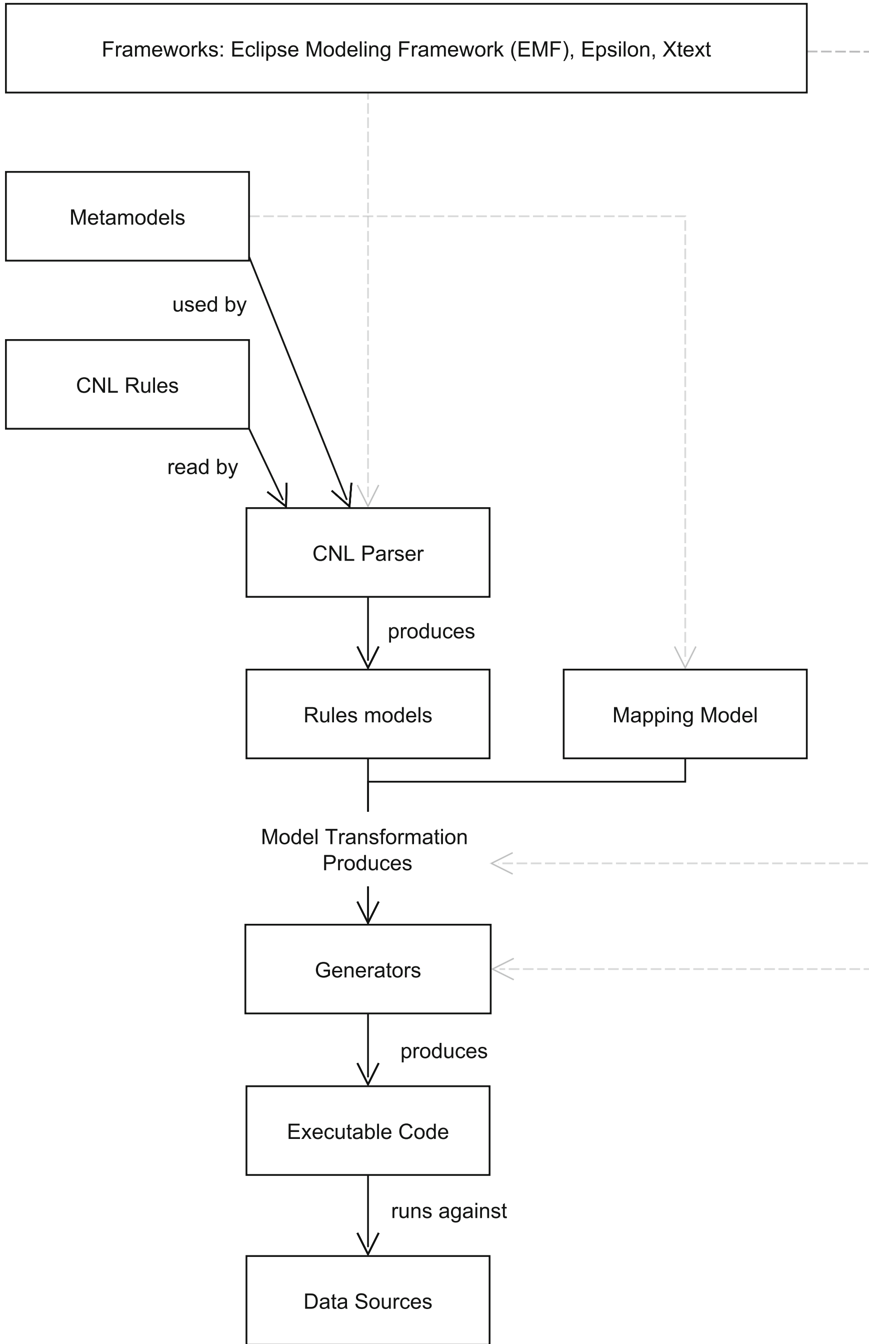


Fig. 1. Architecture of the Open Rules Platform

- Rules/domain/mapping CNL documents. These structured text documents, adhering to the CNL grammar, encode rules/constraints for a particular domain/metamodel of interest. The domain document will contain the terminology used by that domain, which can then be used by the CNL document to write rules for that domain. As such, changing to another domain is as simple as creating a new domain document, and will not affect any part of the CNL syntax other than offering a new domain to write rules against.
- CNL parser. This component parses CNL rules expressed in the language's concrete syntax (aka CNL documents), into in-memory models that conform to the CNL metamodel, that will then be consumed by the transformation engine (Epsilon).
- Generators. Generators consume models and produce executables for a specific back-end and its configuration. These models can be either:
  - The rules/domain/mapping models themselves, whereby a model to text transformation is performed to produce executable code from the models.
  - Back-end specific models representing the technologies used to store the data (such as an SQL model that would conform to a metamodel of the Sequel language). Back-end metamodels will be used for a model to model transformation of the rules/domain/mapping models into a back-end specific model that can then be consumed by a generator to produce executable code. This second approach has not been implemented but can have merit, as discussed in Sect. 5.
- Executables. These are specific to the runtime environment of the end-user such as a MySQL relational database with a specific runtime configuration.
- Back-ends. This is the actual data against which the generated rules will be run against.

3.2 Design

Two metamodels underpin the OR platform: that of the CNL itself and that of the various configurations and mappings required to trace elements from the data itself to the rules written in CNL.

CNL Metamodel. The CNL metamodel captures the abstract syntax of the language. As such it does not contain any information about *how* the CNL will look like but *what* types of elements it can contain. As such, it is the responsibility of the Xtext parser to convert CNL documents into models conforming to this metamodel, which can then be automatically consumed (in this case by Epsilon) to produce executables. Figure 2 shows a simplified version of the metamodel; important metaclasses are briefly introduced below:

- ConstrainedNaturalLanguageRules. Contains a list of validation rules and/or a list of CNL metadata. This root element allows any CNL document to either contain the rules, the metadata (domain the rules are written against) or both.

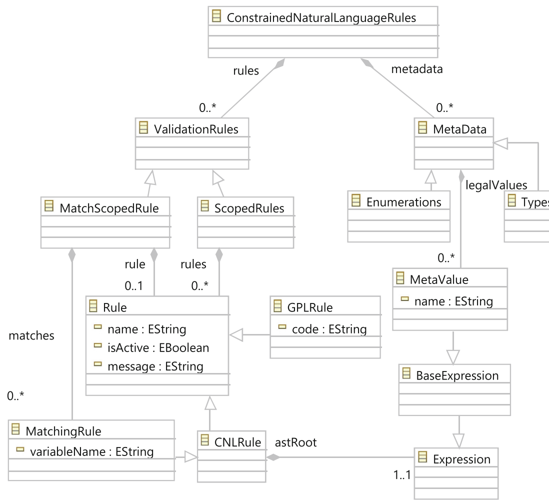


Fig. 2. Simplified CNL metamodel

– ScopedRules. The first type of rule, where one or more rules are written against a single scope (domain element, found in the metadata document).
– MatchScopedRules. The second type of rule, where a single rule is written against sub-collections of data defined in the matches. These collections can be of the same or different scopes, allowing rules to link queries of multiple domain elements together into a single rule.
– CNLRule. A rule written against elements in the metadata domain, in constrained natural language. It contains a root element of the abstract syntax tree used to define it (after it is parsed into such a tree by the CNL parser).
– GPLRule. A rule written against elements in the metadata domain, in a programming language. It contains a String with the relevant syntax conforming to the language (in an unaltered state from the original CNL document). The responsibility of correctly defining this rule lies with the person writing the document as this rule is directly passed on to a relevant parser without change. This type of rule allows for arbitrarily complex expressions which would otherwise not be expressible in CNL to be written in the same document as the natural language rules themselves.
– Expression. The common supertype for all abstract syntax elements a rule can be made up of, such as comparison, arithmetic (summation, difference, multiplication and division), logical (and/or), unary and other simple binary expressions.
– MetaValue. Common supertype for defining domain metadata.

Mapping Metamodel. As different back-ends can require slight variations of similar concepts (such as wrapping identifiers or escaping special characters), a set of configuration options allows abstracting the most commonly found ones

from having to be hard-coded into the generators themselves. These include, for example, converting names to lowercase, specifying special characters that enclose identifiers or strings, other special characters that may need to be substituted, etc.

Since it is unlikely that the domain used to create the CNL rules will be a perfect match with the actual data it is validating, the mapping metamodel is also tasked to map various CNL-domain elements to their appropriate data-domain siblings:

- 1 to 1 mappings. Such mappings denote a single feature in the CNL is mapped onto a single feature in the back-end. This mapping is handled directly by the generator, without the need to introduce any new transformation or other overhead.
- 1 to n mappings. Such mappings denote that a single feature in the CNL needs to be mapped to a list of features in the back-end. The values from this list will be then aggregated (in the order provided by the list), using one of the AggregationOperations available in the mapping model, returning a single result.
- Logical Mappings. As different back-ends may use different symbols for denoting equality, negation or other logical and comparison operators, the technology-specific versions can be provided here, in case they differ from the defaults (for example if SQL uses = instead of == for equality).

These manipulations are used by any Epsilon Generation Language (EGL) generator that generates executables (such as the EVL or the SQL generators provided in the use-case presented in Sect. 4). Any time an identifier (type, feature, variable) is passed to the EGL generator (through the CNL model it is using as its source for code generation), the following may occur, for example:

- A simple mapping (from the mapping model) to replace the identifier with a new one.
- Replacing or removing special characters.
- Converting strings to lowercase.

Generators. These components take a CNL model (after it has been mapped using the appropriate mapping model conforming to the metamodel presented above) and generate appropriate execution-level code to be run against the stored data. Such generators can be produced either using a model to model or a model to text transformation:

Using a model to text transformation is recommended when there are a lot of static text regions that need to be frequently repeated. This provides the freedom to create optimal execution-level code, but may end up being non-trivial to maintain, should the static regions end up becoming too verbose.

Using a model to model transformation is relevant when an appropriate metamodel and unparser of the execution language is available.

Generic EVL Generator. This generator produces code written in the Epsilon Validation Language. The generation mainly comprises transforming the rule abstract syntax tree into the appropriate expression in EVL. This generator can run against any data format supported by the Epsilon framework, such as EMF models, XML documents, Spreadsheets, Relational Databases, etc. Nevertheless, since it is a generic layer it may not be able to be fully optimized against all such technologies and alternative generators may need to be used for performance reasons.

Optimized Native SQL Generator. Since SQL has a substantially different structure to EVL, a native generator provides full control over how a CNL model is converted to executable code to be run against a relational database. This allows tackling of issues such as type correctness by using the database metadata (instead of having to compare each relevant data item to a type), allows effective use of derived tables and merging etc. Preliminary tests have shown that for certain classes of validation rules this can greatly outperform a naïve EVL generator, and that it can be up to an order of magnitude faster than writing inefficient SQL (further investigation onto this is required, as detailed in Sect. 5).

Web-Based Validation. Xtext offers a generated web-based API for writing CNL rules on a web-client and executing a program on these rules on the server side, returning a document containing relevant execution information. Using one of the above mentioned generators, we can perform validation of the provided rules against data stored on the server. This would execute the following (on the server side, after receiving the CNL document from the client):

- The input CNL model is transformed using the relevant mapping model to an in-memory transformed CNL model.
- The transformed CNL model is used to generate the appropriate execution-level code to run against the stored data.

Currently the EVL generator is used to produce EVL code which will run against data stored on an Excel spreadsheet, but both the generator used and the data connector can be replaced if necessary with the appropriate ones. The EVL code is executed against the data, getting back violations for each of the rules. The violations are formatted into a report document which is then returned to the client, providing feedback on offending elements (if any).

4 Evaluation

In this section we present the empirical results obtained when evaluating the OR platform against a domain-specific use-case.

4.1 Use-Case

Annually the Bank of England (BoE) produces a large set of rules that specified UK banks follow to submit data as part of one of the BoE's annual reviews. These rules help ensure the submitted data is consistent with the BoE's expectations and cover a variety of different aspects such as retail risk, commercial risk, operational risk, etc. These rules are provided in a mixture of natural language and procedural statements and are written so that domain experts can understand them, hence are not amenable to machine consumption in any way. In order for these rules to be executed against the actual data the bank holds, they have to be understood by a domain expert and then a technical expert will have to write the appropriate low-level code representing these rules. This process requires stakeholders with different expertise to collaborate and can introduce further risk as the two interpretation steps need to be in line with one another.

4.2 Coverage

As a first criterion for evaluating the OR platform for this use-case, we classified BoE's rules into 8 categories and then analyzed the coverage of the OR platform with respect to the total number of rules. This was done to estimate the actual coverage of the OR platform as it would not have been feasible within the scope of this project to convert all 3668 rules into CNL to execute them; as such random sampling of each category has been performed.

Table 1. Classification of business rules

Category	Description	Count
Type/enum check	This rule only contains a single type or enumeration check	1556
Comparison	This rule only contains a single comparison	315
Comparison+logic	This rule only contains a combination of comparison and logical operators	40
Using variables/functions	This rule requires comparison/logical operations across tables	1534
Duplicate check	This rule requires all values of a field to be different	17
Multi-key match	This rule requires multiple fields to be treated as a key to a search	99
Enumeration sub-matching	This rule requires that the legal values of a field are determined by the current value of another field	27
Complex rule	This rule is too complex to classify	80
		3668

Table 1 presents these categories in more detail. Here, we can see that the large majority of rules fall under either simple type/enumeration checks or elaborate rules requiring the use of variables and functions (matching rules), often across different domain elements. from the remaining rules, the 80 complex rules are noteworthy as it was decided that attempting to further classify them or to convert them to CNL was not efficient. Instead, these rules are flagged as complex and meant to be executed through the use of GPL rules that are written in the target language used to execute against the data itself. Finally, the category of enumeration sub-matching is not yet supported, even though such a feature can be added in further iterations of the tool, as mentioned in Sect. 5.

As such, we achieve 97% coverage (as we don't currently offer CNL expressiveness for complex rules and enumeration sub-matching rules), whilst opening the possibility (through the use of GPL rules) for any rule to be written in the CNL document regardless, in order to ensure that a single document contains all the rules that need to be executed, regardless of whether they can be actually expressed in CNL.

4.3 Verboseness

The second criterion used to evaluate the OR platform is the verboseness of the rules, when written in CNL. Should the CNL form of the rules be disproportionate to the complexity of the rule (the size of the rule written in the execution language) then it may be unreasonable to expect them to be written by domain experts as it will become tedious to write extremely long CNL rules. As such, we compare the size in characters (ignoring whitespaces) of various rules written in CNL with the rule written in both EVL as well as SQL, as a representative sample of verboseness.

Table 2. Rule character count ignoring spaces

Category	cnl^1	cnl^2	evl^1	evl^2	sql^1	sql^2
Type/enum check	33	57	86	111	202	215
Comparison	51	184	118	327	116	405
Comparison+logic	131	139	209	243	139	154
Using variables/functions	314	447	522	703	568	740
Duplicate check	55	58	226	241	116	119
Multi-key match	63	89	487	627	163	187

Table 2 shows the relevant character count for two representative rules written for each of the categories the tool supports. CNL written in the ORP framework is much less verbose than the SQL it would require to execute against data in relational databases (in this case a MySQL database) and less verbose than EVL constraints written in Epsilon. Considering both the EVL and SQL were

generated by the tool and as such attempt to be as minimal as possible (as they do not care about human readability at all), we have gained confidence that writing rules in CNL will require less effort than the same rule written by the relevant expert in EVL or SQL.

Below we see how the type/enum check constraint annotated as cnl[1] looks like:

```
1  in a Branch the country must be in Europe
```

Similarly for the comparison+logic cnl[1] constraint:

```
1  in a MortgageAgreement
2    when the beginningDate exists and the initialEndingDate exists
3    then the beginningDate must be before or by the initialEndingDate
```

4.4 Extensibility

The final criterion used to evaluate the OR platform is its extensibility. Since the system claims to offer domain-agnostic CNL capabilities for writing rules in any domain, we need to gain some confidence that this can be feasible. As such, extensibility can be broken down into three distinct categories:

- Language extensibility. Since the OR platform offers a constrained form of English for expressing rules, this category considers how easy it is to alter this constrained subset should a new type of (English) expression be required or should a new type of executable expression be required (such as the example of the enumeration sub-matching rules in the coverage example, which are not currently expressible in CNL).
  - Regarding extending the subset of English supported by the CNL, this would require adding the new expressions in the Xtext parser that reads the CNL document and creates the relevant model. Since adding new English phrases is unlikely to affect the model itself but rather only the parser, we believe that the OR platform is extensible in this regard as only one component of the system needs to be adapted to add this functionality.
  - Regarding the extending of the semantic expressions offered by the OR platform, this would require the extension of the CNL metamodel to include these new concepts, as well as the extension of the Xtext parser to include a way to express these rules in English. As the adaption of two different interconnected components is required to achieve this, we consider this to be a task of moderate difficulty for an extender of the tool.
- Domain extensibility. If rules need to be written in a different domain, a domain document will have to be created detailing the various concepts in that domain and their relevant features. These concepts will then be usable in the CNL document describing the rules written for that domain. Since the CNL document is not bound to a specific set of concepts but to another

document which will describe these concepts, we believe it is natural to change from one domain to another without much effort.

– Execution technology extensibility. This category considers whether it is possible to change the data storage technology and still be able to execute CNL rules. The execution layer of the OR platform is de-coupled from the language itself, as the data can be accessed either through the Epsilon model connectivity layer (whilst generating EVL rules), or through the use of a new generator that takes the rule document alongside the domain and mapping documents and produces executable code for the required storage technology. As such, we have gained confidence that the OR platform is extensible with respect to use of other data storage technologies.

Overall extending the OR platform for these three categories will require adding/changing only one component in most cases (with two components needing to be changed when new types of semantic expressions need to be added).

5 Conclusions and Further Work

Concluding, we have presented ORP and its CNL, aimed at offering executable validation rules written in natural language. We have evaluated this framework in a real-world case-study using a subset of the Bank of England's business rules and have obtained promising results in both the areas of coverage and verboseness, whilst qualitative evaluation of extensibility is also promising.

The tool can be extended to provide advanced features to cover even more types of rules, such as: n to 1 mappings; such mappings denote multiple fields in the CNL needing to be mapped onto a single field in the data schema (that needs to be disaggregated appropriately). Simple numerical disaggregations (such as each CNL field containing an equal (numerically) subdivision of the target field) can be performed within the CNL itself without the need for further information, but any complex expression-based mapping will need to be presented and incorporated into the mapping model. Reference resolution; to tackle data normalization, references need to be navigated using unique identifiers of elements. This navigation may require extra information such as naming conventions of the target object (such as using a foreign key with a different column name to the original, in a relational database, etc.).

Finally, investigating the applicability of using model-to-model transformations to various back-end technologies through an appropriate metamodel and unparser (for example using an SQL metamodel and an SQL unparser to convert a CNL model into an executable SQL model) can provide insight into this alternative approach.

Acknowledgments. This research was part supported by Innovate UK through its Knowledge Transfer Partnership (KTP) program and JC Chapman LTD.

References

1. Spreeuwenberg, S., Healy, K.A.: SBVR's approach to controlled natural language. In: Fuchs, N.E. (ed.) CNL 2009. LNCS (LNAI), vol. 5972, pp. 155–169. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14418-9_10
2. Clark, P., Murray, W.R., Harrison, P., Thompson, J.: Naturalness vs. predictability: a key debate in controlled languages. In: Fuchs, N.E. (ed.) CNL 2009. LNCS (LNAI), vol. 5972, pp. 65–81. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14418-9_5
3. Njonko, P.B.F., Cardey, S., Greenfield, P., El Abed, W.: RuleCNL: a controlled natural language for business rule specifications. In: Davis, B., Kaljurand, K., Kuhn, T. (eds.) CNL 2014. LNCS (LNAI), vol. 8625, pp. 66–77. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10223-8_7
4. Aiello, G., Bernardo, R.D., Maggio, M., Bona, D.D., Re, G.L.: Inferring business rules from natural language expressions. In: 2014 IEEE 7th International Conference on Service-Oriented Computing and Applications, pp. 131–136, November 2014
5. Njonko, P.B.F., El Abed, W.: From natural language business requirements to executable models via SBVR. In: 2012 International Conference on Systems and Informatics (ICSAI 2012), pp. 2453–2457, May 2012
6. Feuto, P.B., Cardey, S., Greenfield, P., El Abed, W.: Domain specific language based on the SBVR standard for expressing business rules. In: 2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops, pp. 31–38, September 2013
7. Paternostro, M., Steinberg, D., Budinsky, F., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Boston (2008)
8. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA 2010, pp. 307–309. ACM, New York (2010)
9. Kolovos, D.S., Rose, L., Garcia, A.D., Paige, R.F.: The Epsilon Book (2008)

Model-Driven Re-engineering of a Pressure Sensing System: An Experience Report

Atif Mashkoor[1,2](✉), Felix Kossak[1], Miklós Biró[1], and Alexander Egyed[2]

[1] Software Competence Center Hagenberg GmbH, Hagenberg, Austria
{atif.mashkoor,felix.kossak,miklos.biro}@scch.at
[2] Johannes Kepler University, Linz, Austria
{atif.mashkoor,alexander.egyed}@jku.at

Abstract. This article presents our experience in re-engineering a pressure sensing system – a subsystem often found in safety-critical medical devices – using the B formal method. We evaluate strengths and limitations of the B method and its supporting platform Atelier B in this context. We find that the current state-of-the-art of model-oriented formal methods and associated tool-sets, especially in automatic code generation, requires further improvement to be amenable to a wider deployment to industrial applications for model-driven engineering purposes.

1 Introduction

One of the ways to promote the use of formal methods for model-driven engineering of industrial applications is to demonstrate the ability of formal methods to automatically generate executable source code from "correct by construction" software models [10]. However, automatic generation of code from a formal specification such that it requires no further human intervention or post-processing before deployment is a weak link in the development chain [3].

Dataflow-oriented frameworks, such as Simulink[1] or Safety-Critical Application Development Environment (SCADE)[2], are already popular for their model-driven engineering capabilities in safety-critical domains such as avionics and automotive systems [27]. The appeal of these frameworks stems from their graphical notation and simulation capabilities. However, as compared to model-oriented formal methods [12], these frameworks lack sophisticated verification techniques which guarantee *correctness* [10] and also suffer from scalability issues [28]. Model-oriented formal methods lend themselves better to abstraction

The writing of this article is supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

[1] http://www.mathworks.com/products/simulink.
[2] http://www.estereltechnologies.com/products/scade-suite.

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 264–278, 2018.
https://doi.org/10.1007/978-3-319-92997-2_17

and reduction techniques, the key ingredients in modeling and proving correctness properties, and are supported by a variety of model checkers and theorem provers.

In practice, however, many projects are not about developing new software systems from scratch but improving on existing software systems or porting them to new platforms. Thus, it would also be desirable to be able to use formal methods in maintenance and re-engineering projects. Thereby it should also be possible to transform only parts or individual modules. Doing so would improve the quality of software systems and this would also increase the potential for automation such as automatic code generation.

In this article, we report about our experience with the development of a control software for a Pressure Sensing System (PSS) – a subsystem of a hemodialysis machine [21]. A typical pressure sensing system reads sensor data, transforms the data into meaningful information, saves results, checks whether different values are within certain ranges (which depend on certain modes of running the machine), and raises different types of alarm if defined thresholds are violated.

Based on our experience that stems from the application of formal methods on several industrial and academic projects, for example, hemodialysis machines [23,24], aircraft landing gear system [16], machine control systems [25], and stereoacuity measurement system [5], we decided to use the B method [2] for the task. The B method enjoys extensive tool support, covers all the necessary development phases (e.g., support for code generation), and the developers of the PSS system already had experiences with it. Related model-oriented formal methods either do not cover all phases of development, e.g., there is no automatic code generator available for Alloy [14], Temporal Logic of Actions (TLA+) [18] and Z [29], or have limited code generation capabilities, e.g., Abstract State Machines (ASMs) [8], Event-B [4] and the Vienna Development Method (VDM) [15]. A detailed comparison of various model-oriented formal methods concerning their modeling and code generation capabilities is available in [17].

The main objective of the development was the automatic generation of C language code from a formal requirements model that was, in turn, developed through a re-engineering process. Due to space limitation (and also a nondisclosure agreement with the case study provider), we do not include artifacts, such as model and code, in the paper. We also deliberately omit a detailed discussion on the "traditional" use of formal methods, e.g., requirements modeling, property verification, assessment of code complexity, and proof statistics. For such a discussion, interested readers may consult the work by Mashkoor [22] that contains a detailed account of our effort of model-driven engineering of various components of a hemodialysis machine including a discussion on verification and validation. Here, we rather focus on the modeling and code generation experience with the B method.

In the following, we first briefly describe the B method in Sect. 2. Then we present the case study including its description, overall aim and objectives in Sect. 3. In Sect. 4, we highlight the undertaken re-engineering process. In Sects. 5

and 6, we report the experiences and challenges we met in the course of modeling and code generation with the B method respectively. Section 7 discusses some related work. The paper is concluded in Sect. 8 with a discussion on deployed methods and tools.

2 B Method

B is a refinement- and state-based formal method. A B model describes data structures and operations thereupon. A state is defined by particular values of the variables of the data structure, and operations describe state transitions. A so-called *machine* captures a part of the data structure, constraints on the data structure, and operations on the values of the data structure. Requirements are basically described either by constraints (this includes, e.g., safety requirements) or by operations. In a way, a machine resembles programming code for a software module, but it is more abstract, is not tailored to a specific platform, captures requirements more directly, and is suitable for logical analysis.

Modeling in B starts with one or more *abstract machines* which are supposed to capture the most basic requirement(s) in a concise way. These machines are then refined by adding constraints and detailing the actions of operations, according to additional requirements. This is done in separate files which are called "refinements." This way, the consecutive development of the formal specification is well documented. Verification includes proving that each refinement preserves the specified properties of the abstract machine or of the previous refinement.

A special case of refinement is called "implementation." This does not add further requirements to the model, but transforms the final specification model into a form which is suitable for code generation.

The three different machine types – abstract machines, refinements, and implementations – must obey different constraints regarding the language in which they can be expressed. The language for implementations even has its own name, B0 ("B zero").

Tool support for modeling and analyzing in B is available in the form of the Atelier B platform[3]. Atelier B includes an editor, syntax and type checkers, a proof obligation generator, automatic provers and an interactive proving environment, as well as code generators for different target languages. A standalone model checker and animator, ProB [20], is available for B and can be used together with Atelier B.

3 Pressure Sensing System Case Study

3.1 Case Study Description

Pressure sensors are important ingredients of modern diagnostic and therapeutic devices such as dialysis machines, respiratory devices, drug-delivery systems and

[3] http://www.atelierb.eu.

patient monitors. Pressure sensors provide either gauge or differential pressure that is used for various purposes, e.g., extracting and measuring volumetric flow rates, and total fluid volume transferred. This allows to monitor drug administration and to detect anomalies, in which case alarms can be raised. Consequently, pressure sensors enhance the capabilities of medical devices by providing physicians with the ability to measure blood pressure, administer precise quantities of drugs or oxygen, and track patient compliance.

A PSS is safety-critical with respect to human health and life. For instance, in dialysis machines, PSS are vital to ensure that the dialyzed fluid is pumped back into the human body with the right pressure. If the pressure is not within a certain range, an alarm must be raised and the flow must be disconnected from the patient. Both admissible range and type of alarm are thereby dependent on operation mode and other circumstances. Raising an alarm is part of the tasks of PSS software, whose code may attain several thousands LoC (much of which is dedicated to interfaces and data structures).

The PSS is not large but still a nontrivial system. Its implementation uses relatively few programming constructs; in particular, no loops and no recursion are used. Thus we could test only a roughly estimated half of B's major language constructs, and even less of common programming constructs. Yet this is not uncommon for hardware control software, and the restriction of constructs used facilitates safe modeling and programming – or code generation – as well as verification. At the same time, complex data structures with many fields are used in the interface, and many and often nested case distinctions have to be made, in particular for determining whether different values are within permissible limits and how to react if not, depending on various factors, including the operation mode. The interface also requires many type casts to be performed, amongst others. With an order of magnitude of a thousand lines of code, all this makes the software liable to error and thus formal analysis of the code, and even more correct construction by design through the use of formal methods, are bound to improve correctness and thereby safety. We used this opportunity to conduct a pilot project for evaluating the feasibility of methods and tools for eventual deployment.

3.2 Aim and Objectives of the Case Study

The overall aim of the case study was to re-engineer the control software for a PSS – a subsystem of a hemodialysis machine – piece-wise so that the subsystem can be transformed into a form which allows for proving certain correctness properties, e.g., by verification (by automatic theorem proving), validation (by simulation), and automatic test case generation. More generally, the quality, certifiability, and maintainability of existing software should be gradually improved in this way. The primary objectives of the case study were that

- C code should be automatically generated from a formal specification,
- it should be possible to directly integrate the resulting C code in the existing software (without further human intervention or post-processing), and

– the generated code should perform exactly the same externally visible actions as the original one. (This objective should be validatable through inspection of the generated code by developers of the original C code.)

Thus, the project crucially involved re-engineering. This is certainly not an exceptional situation as we have ourselves experienced other cases that required re-engineering of often *completely* undocumented legacy code before. There exists also much safety-critical software for which only *informal* specifications are available.

3.3 System Interface

The interface of the system's most important procedure has the form,

```
void do_sensor(const SHARED_DATA *data_access)
```

where `SHARED_DATA` is a structure containing two other structures, one for data which must not be changed (read-only) and another for data which may be changed (write access) by the software; values of the writable structure are also read within the module, e.g., old values for comparison with the new values (before the old values are updated). In total, the two structures contain several dozen data fields, most of which are writable. It should be noted that the respective procedure parameter, `data_access`, serves for both input and output; there is *no formal* output parameter, although output is actually produced (as a "side effect.") Access control for the shared data is not implemented within the system in question.

4 Re-engineering Process

In order to re-engineer the PSS, we first converted its code into an abstract model in the language of a formal method (and the respective tool platform). Then, we tried to automatically generate code out of this model such that the result can again be integrated into the whole targeted software system.

There was no complete requirements specification available for the respective PSS except for a document that briefly explained a large number of parameters and modes of operation and gives a brief, pseudo-code outline of the required actions. The detailed specification had to be extracted from the provided C code (approx. 1K LoC).

To this end, we abstracted from the C code by means of the ASM method. This method is more suitable for reverse engineering as it allows for n-to-m refinement, where the number of algorithmic steps in the refinement (m) may actually be smaller than the original number of steps (n). B, in contrast, allows only for 1-to-1 refinement (through the definition of originally abstract submachines). On the other hand, tool support for B is much better than for ASMs, especially for verification and code generation, so we chose to use both methods to exploit their relative strengths where appropriate.

One of the important cornerstones of the specification process is the representation of requirements at various abstraction levels using the notion of refinement. By following this technique, requirements are easy to specify, analyze and implement. In this style of specification writing, requirements are incrementally added to the model until the model is detailed enough to be effectively implemented. If the refinement model of the method is flexible enough, such as that of the ASM method, it is possible to reverse this process for the sake of increasing abstraction.

It is not possible to directly translate an ASM model to a B model with an off-the-shelf tool or method. However, both B and the ASM method are state-based methods and corresponding models are similar enough to allow for manual translation with a high degree of confidence, provided that the theoretically more expressive ASM method is exploited only to the point where it remains compatible with the expressive capabilities of B.

Once the informal requirements were formally specified, the next step was to make sure that the requirements conformed to verification standards, i.e., requirements are consistent and verifiable. During this process, it was determined that a specification conformed to some precisely expressed properties that the model is intended to fulfill such as well-definedness, invariant preservation and other safety conditions. For verification of the model, we used two well-established approaches of theorem proving and model checking. The former helped us to reason about defined properties using a rigorous mathematical approach. The latter helped us to verify dynamic properties of the model by exploring its whole state space. While theorem proving is helpful in ensuring

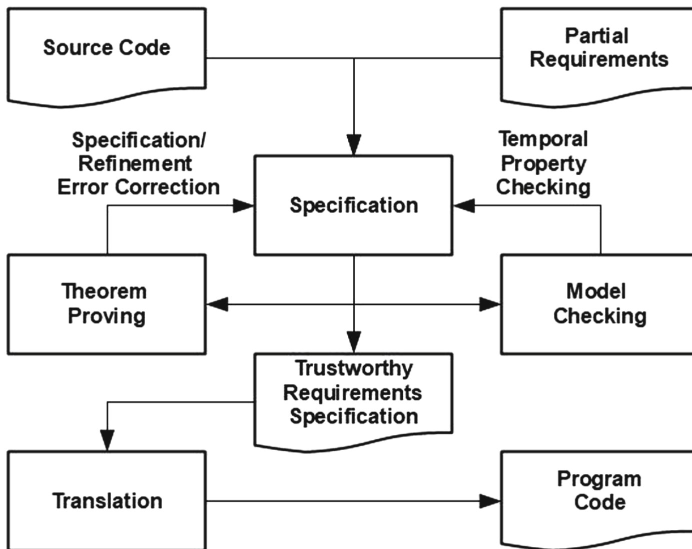


Fig. 1. Model-driven re-engineering process

safety constraints of the system, model checking is effective in verifying temporal constraints of the system such as liveness and fairness properties and also help in validating the specification against requirements.

The last step of the formal development process is the translation of the requirements specification into program code. This last refinement step is, in fact, already very detailed and close to the implementation stage. The whole re-engineering process is depicted in Fig. 1.

As the main objective of the work was code generation, the rest of the paper will focus only on our experiences and challenges with the B method. However, for a detailed comparison between the ASM and B methods, please see [17].

5 Modeling Experience

In the following, we present our modeling experience with the B method. We first describe what we particularly liked about the method, and later, what was limiting about it.

5.1 Strengths of the B Method

Composition

Support for composition and decomposition in a modeling method is important for any domain of application when it comes to "larger-than-toy" systems [9]. Without decomposition, large complex models cannot be effectively over-viewed and handled. Decomposition is also of great value while proving the correctness of a large system. Composition is also important for model reuse.

Composition and decomposition are supported by the B method by allowing to call operations of other machines and to access, e.g., data structures from other machines. This works basically like calling procedures in procedural programming languages, but B additionally provides a few options regarding the visibility and accessibility of elements of other machines. Every machine has its own file.

Refinement

Refinement is a way of specifying the requirements of a complex system through a series of models for the same system with increasing depth of detail or, for reverse engineering, with increasing abstractness. Refinement is the central element in the B method. B defines a very powerful and well-supported refinement process. B supports a one-to-one notion of refinement. This is rather strict but eventually results in a higher degree of automatically discharged proofs. B allows to refine a model up to the level of detail required for implementation, or actually right down to programming code.

In practice, refinement relies very much on defining the actions of operations which can initially be the empty action, "`skip`." That is, in the operations of one machine one can call operations of other machines which may

initially be left abstract. For instance, in this project, one machine may call another machine called "`update_shared_data`," for which the algorithmic definition remains "`skip`," i.e., abstract, because those details are not relevant for the specification or for the current level of abstraction.

Nondeterminism

Support for nondeterminism in a modeling method is very useful for keeping models abstract. B supports nondeterminism by allowing for nondeterministic choice of values for variables out of a given set (corresponding to Hilbert's ϵ operator) as well as by operators "`ANY`" (unbounded choice of value) and "`CHOICE`" (nondeterministic choice of alternative substitutions). In this fashion, new concepts can be added to specifications abstractly in the earlier refinements and can be concretized in the later ones. For example, in this project, we initialize a variable, potentially of a complex data type, with an unspecified value from this data type – as in "`l_d_sensor_input :: SENSOR_DATA`." Thereby no assumption about a concrete value is made other than that it is of type `SENSOR_DATA`.

Correctness Assurance

The possibility to express and prove properties, such as consistency, safety and temporal constraints (e.g., termination, deadlock freeness, fairness, and liveness), are integral to reason about the correctness of a safety-critical system. B enables the expression of typical safety properties through invariants. An invariant is a property that the specification is assumed to meet and maintain. Following is a sample requirement from the project:

> If the system is in the preparation mode or if the system is in the therapy mode and if the critical fluid temperature exceeds the maximum temperature of 41 °C, then the software shall disconnect the supply of the critical fluid within 60 s and execute an alarm signal.

The safety properties are specified in terms of invariants as follows:

inv1 $\mathsf{systemMode} = \mathsf{Preparation} \wedge \mathsf{criticalFluidTemperature} > 41 \Rightarrow$
$\mathsf{systemState} = \{\mathsf{CriticalFluid} \mapsto \mathsf{Disconnected}\} \wedge \mathsf{disconnectionTime} < 60 \wedge \mathsf{alarm} = \mathsf{ALM1}$
inv2 $\mathsf{systemMode} = \mathsf{Therapy} \wedge \mathsf{criticalFluidTemperature} > 41 \Rightarrow$
$\mathsf{systemState} = \{\mathsf{CriticalFluid} \mapsto \mathsf{Disconnected}\} \wedge \mathsf{disconnectionTime} < 60 \wedge \mathsf{alarm} = \mathsf{ALM2}$

The support environment Atelier B generates POs to make sure that the system specification is well-behaved, i.e., maintains system invariants.

5.2 Limitations of the B Method

No Loops Except in Implementations

A restriction for abstract machines and refinements except for implementations – the last refinement steps before code generation – is that no loops are allowed (in implementations, `WHILE` loops are possible). While we did not need loops in our case study (so far), this restriction can certainly be critical.

Data Types

In a specification and an associated abstract model, usually very few data types are really needed. For this purpose, the data types provided by B (including booleans, integers, arrays, and structures) are largely sufficient. However, in an associated programming code – which also has to take into account efficient use of resources, amongst others –, we often need ultimately more such as a 16-bit integer, a 8-bit unsigned integer, and strings. In the context of the programming language C, *pointers* are also important constructs whose support is not present in typical modeling languages, so as B. In our case, already the modeling of the *interface* turned out to be a problem due to this nonexistent support of pointers.

Subsets

In B0 - the restricted B "dialect" required as a basis for code generation - all enumerated types have to be redefined as integer intervals. Arbitrary sets of integers are not allowed. But this precludes the definition of subsets (sub-types) whose members are not consecutive members of the base type. For instance, when we have a range of (named) colors as the base set, then we cannot create a greater number of arbitrary subsets of that – say, rainbow colors, RGB-colors, reddish colors, etc. – even with the most fancy ordering of the colors in the base set. (n.b., the result – an *interval* – is not a "set" in the mathematical sense any more because of the imposed ordering!) In general, the limit is two subsets.

In our case study, certain data types are used all of which may have the value `NO_DATA`. The definition of such types is not possible in B0 because (a) more than two of such otherwise disjoint types cannot be defined as sub-types of a common supertype (or `SET`), as stated above, and (b) `NO_DATA` can only be defined once and cannot be a member of different sets (which have to be disjoint when defined in `SETS`). (Note that we are only talking about enumeration sets here – with numerical types (e.g., `INT`), this is not possible at all.) A workaround for the "`NO_DATA`" problem is to define different constants for different types – `NO_DATA_X, NO_DATA_Y, ...` –, but this is not compatible with given interfaces.

Note that this problem only surfaces at the level of *implementations*, i.e., the last refinements before code generation; in B proper, arbitrary subsets can be defined (as constants).

Restrictions on Record Handling

Single record fields cannot be directly set via an operation call. Instead, an auxiliary variable has to be used (i.e., such an assignment requires two lines of code instead of one). We suspect that there may be further, similar restrictions for handling record fields; see also code generation problems regarding structures/records in the next subsection.

Identifiers

The language B imposes restrictions on identifiers: scalar parameters of machines must be lowercase only while set parameters must be uppercase only. While this had no direct influence on our case study (although the use of machine parameters might be considered in further development), this indirectly imposes the *convention* to use only uppercase names for sets in general, and only lowercase names for constants and variables, for instance. This, however, may clash with conventions of existing code (and did so in our case study).

No Pragmas

It seems impossible to model compiler directives (pragmas) in B. Pragmas might be seen as too implementation-specific constructs to earn a place in formal modeling, but they are frequently used in legacy code (including the one we dealt with; e.g. `#if 0 ... #endif`) and may be required when just single modules of a larger system are to be modeled. (Note that `#include` and `#define` statements are automatically generated by Atelier B from respective `IMPORTS` or `SEES` sections or from constant definitions, respectively.)

6 Code Generation Experience

In the following, we present our experience with code generation. For the case study, we used the community version of Atelier B including its code generator that is released for public use after every two years[4].

6.1 Strengths of the Atelier B Suite

The provided tool support is very good. Atelier B comes with code generators for different target languages, including C, C++, Java, and Ada. Although the generated code requires some post-processing, it is a good basis for the implementation of a B specification. The generated code is well-structured, well-documented and legible.

6.2 Limitations of the Atelier B Suite

Identifiers

In order to fit into a given interface, identifiers have to match exactly. However, this is not possible with the code generator of Atelier B. This code generator produces identifiers – most importantly, procedure names – as a combination of the machine name and the operation or set or constant name, separated by a double underscore. This is motivated by the need to avoid identifier clashes and

[4] According to ClearSy, they will fix some of the concerns raised in this paper in the upcoming version of the code generator.

related scoping problems (see [1, p. 6]). However, this renders it impossible to get procedure (and custom type) names prescribed by an existing interface.

This is a *crucial* problem in a project setting where only a part of an existing piece of software shall be transformed to code generated from a formal model. Manually rectifying all these names is tedious and error-prone. Automated post-processing would have to be performed separately for each project[5].

Structures and Records

The translation of structures ("`struct`" types) is actually faulty: the result *cannot be compiled* without post-processing (we did manual corrections).

In the B model, we had a single definition file in which all the `struct` types were defined. But in the generated code, any typing declaration with such a structure was individually expanded to the whole type definition with all fields and their types, and everywhere it was given a different name. In the interface of a procedure with two parameters of the same `struct` type, this `struct` was twice expanded and given two different names. The respective type names (`R_1`, `R_2`, ...) even differed between the header files and the corresponding definition files.

To mend this, in post-processing, one has to make the necessary `struct` definitions once and for all in some header file and then change every occurrence of this type to the respective type name. If one has large structures (in our case, with up to 36 fields), this is hard to automate, even using regular expressions, and error-prone. (The regular expression "`struct R_?` {*}" actually matches *any* occurrence of *any* structure type, and the number attached to `R_` does not give any indication as to which particular structure type is actually given in the respective place.)

Miscellaneous Observations

In B0, the language from which code can be generated, we discovered a strange restriction on expressions, i.e., for `IF` conditions: a statement of the form "`IF a < (b + c) THEN ...`" is not possible (though it is in B in general); instead, one has to add an auxiliary statement: "`aux := b + c; IF a < aux THEN ...`"[6].

An issue related with data type restrictions as well as with identifier restrictions is that the generated code requires the Standard Library for C. This may seem reasonable at first sight, but in practice, the Standard Library is not always used in industrial practice. This is a real problem in particular when the task is to obtain code which fits into given interfaces of a larger, existing system. In

[5] According to ClearSy, in the upcoming version of the code generator, custom identifiers (without prefix) would be possible. This would indeed constitute a major improvement and should be regarded as an important *and feasible* requirement for code generators.

[6] According to ClearSy, this construct eases the proving process.

such a case, there is no realistic possibility to change the libraries used or to rename types (or other identifiers). At the moment, further post-processing is required if, e.g., the Standard Library is not used or not supported in the target environment.

7 Related Work

Bert et al. [7], apart from our work, also critically evaluated the performance of the Atelier B platform in this direction and have suggested several improvements. An experience report involving Atelier B by Beneviste [6] briefly mentions code generation for VHDL. However, both of these aforementioned works were not really useful for our case study.

Event-B is a variant of B for higher-level specification, verification and validation of systems and environments where software systems are supposed to operate. We, alternatively, tried to generate code from an Event-B model of the pressure sensing system but failed. It turned out that none of the available code generators for Event-B was usable for our purpose. The one by Wright [30] was custom-built and only supports a part of the Event-B syntax. The most significant shortcoming is that it does not support contexts and therefore cannot be used when constants and sets are used in a model. The one by Fürst et al. [13] is not publicly available, thus we could not use it at all. EB2ALL [26] explicitly requires the manual alteration of code after generation. This is contradictory to one of the objectives of the case study. The Tasking Event-B tool [11], which appears to be the most mature of all, is compatible only until Rodin 2.8 (the current Rodin version at the time of writing this paper is 3.2) and may only work properly on 32-bit machines; however, such a machine was not available for the case study.

8 Conclusion

The use of formal methods is "highly recommended" for safety-critical software by international standards and can actually also be economic in the long run for other kinds of software. Yet there are still a couple of hurdles for a more widespread use of formal methods in industrial software development, which have to be addressed one by one. Two such issues are the extra effort invested in formal modeling and the potentially unsafe transition from the specification to the implementation. Both issues can be tackled by means of automated code generation from formal models, i.e., from those models which are used in formal specification and analysis.

We have put the current state-of-the-art tool support for code generation to test with a case study drawn from real industrial software development. We have chosen B for this purpose because it is well-suited for ordinary software development, is well-known also in parts of industry, supposed to be mature, has commercial tool support including code generation, and there was already expertise available within the team.

The results of our practical investigations are mixed. First and foremost: yes, generating code from B models, using B's standard tool Atelier B, *does* work. However, there are several restrictions which we have encountered, ranging in severity from inconvenient to critical. Some of these restrictions not only concern code generation, but also modeling in B.

We believe that the necessary improvements are feasible with reasonable effort. For the scenario which the formal methods community typically envisages, i.e., where one starts with a completely new development from requirements engineering via formal specification and analysis to design and coding (etc.), code generation already works quite well when we take aside type restrictions, the problems with records which we encountered, as well as efficiency issues.

But for a scenario which involves re-engineering and improving a single module out of a large system by means of (e.g.,) the B method, there are still major shortcomings, as we have described in detail in this paper. Note, however, that other application examples are likely to identify more issues; to name just a few examples, we did so far not touch upon loops, recursive function calls, float (real) types, or strings. Some of the necessary improvements will probably require more customization of the tool. This concerns enhanced support of (custom) types in particular. It should be noted, though, that the tool is still being improved and certain issues may even have been solved by the time this paper is actually published, or will be solved in the near future.

According to ClearSy [19], almost all safety-critical products using B have their own code generator because of the constraints imposed by the platform running it. However, this further confirms our impression that off-the-shelf code generation is still an open issue (which probably holds for other formal methods as well).

A factor for the successful application of a particular method which cannot be neglected is the existence of a dynamic community wherein experiences can be shared and issues can be discussed. The B method has rich tool support and a sizable community in this regard. On the other hand, more popular methods with larger and more diverse communities, such as Event-B and TLA+, are either not universally applicable, or do not support code generation, or both. This suggests that a new impetus is needed for general-purpose formal methods which support the whole software development cycle, from specification and analysis to code generation and test suite generation to maintenance and versioning/product family development.

Still, it must be noted that, in principle, most important technologies and tools are there and do work. What is desirable now is further improvement and consolidation of these methods and tools.

We end with final remarks for practitioners. When starting to use formal methods in software development, the goals of their introduction must be clear and expectations need to be realistic. Furthermore, it is important to select a suitable method for the chosen goals and with respect to the given organizational environment and similar parameters. It should also be considered to introduce formal methods *incrementally* so as to minimize impact on development time and

costs at each step and thus lowering the economic and psychological thresholds which cannot be completely avoided at the beginning. This article may help to assess what can be expected from the current state-of-the-art of some important aspects of formal methods. This article also explicitly addresses the scenario of incremental introduction, in particular, using formal methods for selected modules out of a larger system.

Acknowledgment. Thanks to Thierry Lecomte (ClearSy) for providing feedback that helped a lot to improve the quality of the presented work.

References

1. Atelier B Translators: User Manual version 4.6. http://tools.clearsy.com/resources/documents. Accessed 28 Feb 2018
2. Abrial, J.R.: The B Book. Cambridge University Press, Cambridge (1996)
3. Abrial, J.R.: Formal methods in industry: achievements, problems, future. In: Proceedings of the 28th International Conference on Software Engineering ICSE 2006, pp. 761–768. ACM, New York (2006)
4. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
5. Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., Riccobene, E.: Formal validation and verification of a medical software critical component. In: 13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, pp. 80–89. Austin, 21–23 September 2015
6. Benveniste, M.: On using B in the design of secure micro-controllers: an experience report. Electron. Notes Theor. Comput. Sci. **208**, 3–22 (2011)
7. Bert, D., Boulmé, S., Potet, M.-L., Requet, A., Voisin, L.: Adaptable translator of B specifications to embedded C programs. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 94–113. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_7
8. Börger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag New York Inc., Secaucus (2003)
9. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. **28**(4), 626–643 (1996)
10. Daskaya, I., Huhn, M., Milius, S.: Formal safety analysis in industrial practice. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 68–84. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24431-5_7
11. Edmunds, A., Butler, M., Maamria, I., Silva, R., Lovell, C.: Event-B code generation: type extension with theories. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 365–368. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_33
12. Fitzgerald, J.S., Larsen, P.G.: Triumphs and challenges for model-oriented formal methods: the VDM++ experience. In: Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006), pp. 1–4, November 2006
13. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code generation for Event-B. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 323–338. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10181-1_20

14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
15. Jones, C.B.: Systematic Software Development Using VDM, 2nd edn. Prentice-Hall Inc., Upper Saddle River (1990)
16. Kossak, F.: Landing gear system: an ASM-based solution for the ABZ case study. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 142–147. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_10
17. Kossak, F., Mashkoor, A.: How to select the suitable formal method for an industrial application: a survey. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 213–228. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_13
18. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
19. Lecomte, T.: Atelier B has turned twenty. In: Keynote of the Fifth International Conference on ASMs, Alloy, B, TLA, VDM, and Z (ABZ 2016). Springer, Heidelberg (2016)
20. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
21. Mashkoor, A.: The hemodialysis machine case study. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 329–343. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_29
22. Mashkoor, A.: Model-driven development of high-assurance active medical devices. Softw. Q. J. **24**(3), 571–596 (2016). https://doi.org/10.1007/s11219-015-9288-0
23. Mashkoor, A., Biro, M.: Towards the trustworthy development of active medical devices: a hemodialysis case study. IEEE Embed. Syst. Lett. **8**(1), 14–17 (2016)
24. Mashkoor, A., Biro, M., Dolgos, M., Timar, P.: Refinement-based development of software-controlled safety-critical active medical devices. In: Winkler, D., Biffl, S., Bergsmann, J. (eds.) SWQD 2015. LNBIP, vol. 200, pp. 120–132. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-13251-8_8
25. Mashkoor, A., Hasan, O., Beer, W.: Using probabilistic analysis for the certification of machine control systems. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E., Xu, L. (eds.) CD-ARES 2013. LNCS, vol. 8128, pp. 305–320. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40588-4_21
26. Méry, D., Singh, N.K.: Automatic code generation from Event-B models. In: Proceedings of the Second Symposium on Information and Communication Technology SoICT 2011, pp. 179–188. ACM, New York (2011)
27. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM **53**(2), 58–64 (2010)
28. Reicherdt, R., Glesner, S.: Formal verification of discrete-time MATLAB/Simulink models using boogie. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 190–204. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_14
29. Spivey, J.M.: Understanding Z: A Specification Language and Its Formal Semantics. Cambridge University Press, Cambridge (1988)
30. Wright, S.: Automatic generation of C from Event-B. In: Workshop on Integration of Model-based Formal Methods and Tools (2009)

Modeling AUTOSAR Implementations in Simulink

Jian Chen[1(✉)], Manar H. Alalfi[2], Thomas R. Dean[1], and S. Ramesh[3]

[1] Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada
{jian.chen,tom.dean}@queensu.ca

[2] Department of Computer Science, Ryerson University, Toronto, Canada
manar.alalfi@scs.ryerson.ca

[3] General Motors R&D, Warren, MI, USA
ramesh.s@gm.com

Abstract. AUTOSAR (AUTomotive Open System ARchitecture) is an open industry standard for the automotive sector. It defines the automotive three-layered software architecture. One layer is application layer, where functional behaviours are encapsulated in Software Components (SW-Cs). Inside SW-Cs, a set of runnable entities represent the internal behaviours and are realized as a set of tasks.To address AUTOSAR's lack of support for modelling behaviours of runnables, other modelling languages such as Simulink are employed. Simulink simulations assume tasks are completed in zero execution time, while real executions require a finite execution time. This time mismatch can result in failures of analyzing an unexpected runtime behaviour during the simulation phase. This paper extends the Simulink environment to accommodate the timing relations of tasks during simulation. We present a Simulink block that can schedule tasks with a non-zero simulation time. This enables more realistic analysis during the model development stage.

Keywords: AUTOSAR · Simulink · Simulation · Scheduling

1 Introduction

Modern automotive systems are software intensive and the complexity of these systems is rapidly growing. There are many critical functions of modern vehicles that rely on software. An example is embedded controllers that coordinate with each other to perform advanced control functions such as autonomous driving, active safety, and infotainment. All of these functions are related to software which indicates cars contain some of the largest pieces of software[1]. A modern high-end car features around 100 million lines of code. To address the challenge of automotive systems, a worldwide development partnership AUTOSAR was formed [1]. AUTOSAR standardizes the entire automotive electronic software architecture and development methodology [16].

[1] http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code.

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 279–292, 2018.
https://doi.org/10.1007/978-3-319-92997-2_18

While there are many tools that support the AUTOSAR process, MATLAB/Simulink (ML/SL) is a popular option for developing automotive software that meets the AUTOSAR standard, providing a tool chain that supports the AUTOSAR development process. We can directly use Simulink blocks to develop AUTOSAR software components. Embedded Coder[2] provides the mapping of the Simulink models to AUTOSAR components and generates the AUTOSAR compliant production code.

Simulation is a process of representing the actions of a real-world system. Through simulation, engineers can evaluate the system design and diagnose problems in the early phase of the design process. However, the Simulink simulation algorithm dose not take every factor of the real world into account such as the time of real-world computation. In other words, software tasks are completed in zero execution time in the simulation stage. In the real world, software tasks take non-zero execution time that varies according the hardware platform. Hence, simulation cannot reflect a real execution at run-time on a specific target hardware platform. Therefore, developing a more realistic model of an AUTOSAR based software application in Simulink is needed.

AUTOSAR supports a modified version of the priority ceiling scheduling [7]. In this approach, when a lower priority process uses a shared resource, and a higher priority process needs access to the shared resource, the priority of the lower priority process is raised to a higher priority so that it may finish using the resource. This is not always the desired behaviour. Ideally, the contention over the shared resource should be minimized during the modelling phase. However, this requires accurate simulation of the timing of each of the software components.

Automotive ECU (Electronic Control Unit) software consists of multiple threads which are often encapsulated in time-triggered tasks and executed on a Real-Time Operating System (RTOS). The use of model-based development in creating the ECU software is limited in that a thread in an ECU is derived from multiple Simulink/StateFlow models that are independently developed, validated and code-generated. The concept of a thread and timing are largely absent at the time of development and validation of the models. Thus there is a large discrepancy between the run-time semantics and the models giving rise to additional work at run-time. This can be avoided if the run-time abstractions of time-triggered tasks can be captured early at the modelling level. This would enable carrying out detailed concurrency and timing analysis early in the cycle thereby reducing the overall time and efforts involved in the development and validation cycle of ECU development. In this research, we propose an approach that can reflect the real system behaviours during the simulation phase and in the future, identify race conditions at the model level.

[2] https://www.mathworks.com/products/embedded-coder.html.

1.1 AUTOSAR

AUTOSAR aims to meet the needs of future cars, provides an open industry standard among suppliers and manufacturers. The best way to achieve this goal is to minimize the coupling of software modules through abstraction. Hence, AUTOSAR defines three main layers: the application layer, the runtime environment (RTE), and the basic software (BSW) [17].

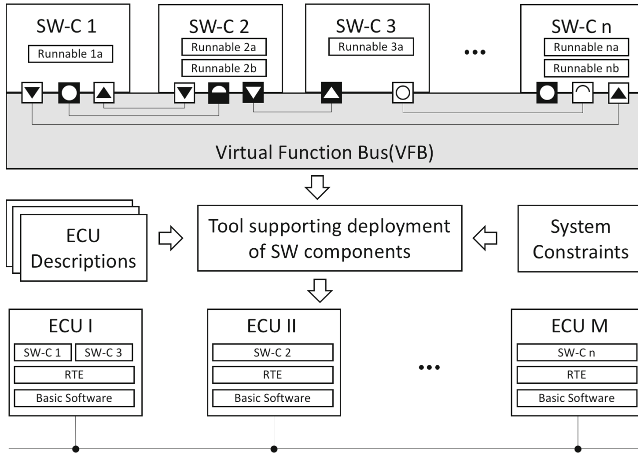


Fig. 1. AUTOSAR components, interfaces and runnables. (Adapted from [1])

The functions in application layer are implemented by SW-Cs, which encapsulate part or all of the automotive electronic functions as shown in Fig. 1. The component communications are via a new concept VFB (Virtual Functional Bus), which is an abstraction of all communication mechanisms of AUTOSAR. Using VFBs, engineers abstract the communication details of software components. Inside the SW-Cs, the internal behaviours are represented by a set of runnables. A runnable is the smallest piece of code that can be independently scheduled either by a timer or an event. Finally, runnables are implemented as a set of tasks on a target platform. Runnables from different components may be mapped into the same task and must be mapped in such a way that ordering relations and causal dependencies are preserved.

1.2 AUTOSAR Support in ML/SL

In fact, ML/SL has supported AUTOSAR compliant code generation since version R2006a. ML/SL and Embedded Coder provide a powerful platform for AUTOSAR software development from behaviour modeling to production code generation. First, each single AUTOSAR concept can be represented by an ML/SL block. Existing ML/SL blocks can be applied to AUTOSAR development and no additional AUTOSAR-specific blocks are required. Table 1 shows

examples of key mappings between AUTOSAR concepts and Simulink concepts [15]. Second, ML/SL provides a *Simulink-AUTOSAR Mapping Explorer* for configuring the mapping of Simulink inports, outports, entry-point functions, data transfers, and lookup tables to AUTOSAR elements. Last, Embedded Coder software supports AUTOSAR-compliant C code generation and AUTOSAR XML(ARXML) description files exporting from an ML/SL model.

Table 1. Examples of ML/SL and AUTOSAR concepts mapping

ML/SL	AUTOSAR
Subsystem	Atomic software component
Function call subsystem	Runnable
Function calls	RTEEvents

1.3 Simulink

ML/SL system models are blocks connected to each other by signals between input and output ports. ML/SL simulation engine determines the execution order of blocks before simulation in a sorted order, called the block invocation order. The block invocation order can be determined by the data dependencies among the blocks. ML/SL uses two kinds of block direct feedthrough and non-direct feedthrough to ensure the simulation can follow the correct data dependencies. A block for which the output ports is directly determined by its input ports is a direct-feedthrough block, while a block for which inputs only affect its state is a non-direct feedthrough block. ML/SL use the following two basic rules to form the sorted order [11]: A block must be executed before any of the blocks whose direct-feedthrough ports it drives; Blocks without direct feedthrough inputs can execute in arbitrary order as long as they precede any block whose direct-feedthrough inputs they drive. All blocks are scheduled in a sorted order and executed in a sequential execution order. The simulink engine maintains a virtual clock to execute each ordered block at each virtual time. Hence, a Simulink block is usually exhibited as a zero execution time behaviour.

Simulink Coder[3] not only supports code generation for ML/SL models, it offers a framework to execute the generated code in a real-time environment. The framework assures the generated code follow the standard of simulation engine and the implementation should preserve the semantics of models. Simulink Coder has two code generation options for periodic tasks: single task and multi-task. Single task implementations can preserve the semantics during the simulation because the generated code is invoked by a simple scheduler in a single thread without preemptions. For multi-task implementations, the generated code is invoked by a rate monotonic (RM) [8] scheduler in a multithreaded RTOS environment, where each task is assigned a priority and preemptions occur

[3] https://www.mathworks.com/products/simulink-coder.html.

between tasks. As a consequence of preemption and scheduling, the implementation semantic can conflict with the model semantic in a multi-rate system. Hence, the Simulink simulation does not always reflect the actual model behaviours in implementation. In this work, we develop a scheduler that can schedule the executions of ML/SL blocks with priorities and preemptions during the simulation.

1.4 Scheduler

ML/SL uses a scheduler mechanism to schedule the execution of Simulink subsystems in a specific order [12]. The scheduler is implemented by Stateflow charts and it implicitly controls the order of execution in a Simulink model. There are three kinds of schedulers that can be implemented using Stateflow including Ladder logic scheduler, Loop scheduler, and Temporal logic scheduler. In this work, we developed a new scheduler to replace the ML/SL scheduler to enable a more realistic simulation.

2 Related Work

Logical Execution Time (LET) [6] was introduced as part of the time-triggered programming language Giotto. It abstracts from the physical execution of a real-time program to eliminate I/O execution time so that a LET model execution is independent from its actual execution. LET uses ports to define a logical task execution, input ports take values at the start of a task and the output ports release the values at the end of the task execution. LET has an assumption that actual task execution should be able to be finished during the logical execution. Derler *et al.* [3] demonstrated that real-time software based on LET paradigm has the ability to exhibit the equivalent behaviour on a specific platform during the simulation phase in ML/SL. However, Naderlinger *et al.* [14] points out that data dependency problems may occur when simulating LET-based software.

In order to keep data consistency and preserving semantics, Ferrari *et al.* [4] discuss the proof of absence of interference, disabling of preemption, communication buffers and semaphores as possibilities on a single-core resource in the context of AUTOSAR. Zeng *et al.* [18] present similar mechanisms for the preservation of communication semantics for a multi-core platform.

TrueTime [5] simulator is an ML/SL based network simulation toolbox and it is good for co-simulation of scheduling algorithms, control algorithms, and network protocols. TrueTime is designed as a research tool that requires a learning curve for system engineers to use this tool. Additionally, tasks cannot be expressed directly using production code and requires a special format for function code.

Cremona *et al.* [2] propose a framework TRES, which is used for a co-simulation of the software model and the hardware execution platform. It adds the schedulers and tasks to Simulink models to model the scheduling delays.

Recently, Naderlinger [13] introduces timing-aware blocks into ML/SL, which consumes a finite amount of simulation time so that simulation behaviour of ML/SL models is equivalent to real-time execution behaviour.

Our work differs in the sense that we aim to bring the impact of real-time execution to the semantics of model simulation in the context of AUTOSAR. Hence, our approach natively support AUTOSAR development in ML/SL and the model scheduler can be integrated into code generation.

3 Model Scheduler

In order to reflect the real-time execution of an AUTOSAR Simulink model on a actual hardware during the simulation process, we propose a customized scheduler, Model Scheduler, which schedules the order of execution of each subsystem at a specific time so that Simulink simulation is able to capture the real behaviour of AUTOSAR applications.

Our model scheduler replaces the Stateflow temporal logic scheduler in the ML/SL model and schedules a set of given tasks with non-zero execution time so that the model can have a real-time behaviour during simulation. The model scheduler is implemented as an S-Function block that can easily substitute for the Stateflow scheduler in an ML/SL model. The model scheduler takes tasks and runnables information as input parameters and outputs scheduled subsystem function call triggers. Inside the model scheduler, we implemented a preemptive scheduling algorithm written in C based on Fixed Priority Scheduling (FPS) [9] algorithm, which computes the scheduling and the model scheduler outputs a subsystem function call trigger when a task is scheduled. A function call trigger is a control signal, which triggers the connected subsystem to execute when a control signal has function-call event.

While one of the standard scheduling algorithms in OSEK/AUTOSAR is priority ceiling scheduling, we would like to minimize the changes in priority of tasks due to shared resources. Thus we use FPS so that we can identify race conditions that occur in the model. In FPS, each task has a fixed priority preassigned by users, and they are stored in a ready queue in an order determined by their priorities. The highest priority task are selected from the ready queue to execute. The oldest task will be selected if there are more than one of same priority tasks exist. In a preemptive system, if a higher priority task is scheduled during the execution of a lower priority one, then the higher priority task is executed immediately and the lower priority task is moved to the ready queue. RM scheduling algorithm is one of the widely used FPS algorithm and it is used in Simulink Coder for code generation. In RM scheduling, the priority of a task is associated with its period, a task has a smaller period then it has a higher priority.

The S-Function provides a mechanism to extend the capabilities of ML/SL by customizing blocks and S-Functions can be accessed from a block diagram so-called *S-Function block*. Customized algorithms can be added to Simulink models via S-Function blocks written either in MATLAB or C. An S-Function block has a parameter field that users pass specify parameters to the corresponding S-Function block. S-Functions communicate with the ML/SL engine through a set of callback functions so-called S-Function API. Thus, S-Functions make it possible to control ML/SL simulation process by using a customized algorithm.

4 Tool Implementation: Model Scheduler

Model Scheduler is implemented as a custom configurable Simulink library block that includes an FPS algorithm written in C and invokes the connected subsystem in the ML/SL interactive development environment.

An S-Function contains a set of callback functions, which the simulation engine executes at different stages during simulation. We use output function *(mdlOutputs)* that computes the output values based on the input parameters. Before running a simulation, we need to provide the necessary parameters in the block parameters dialogue shown in Fig. 2. The parameters include *Task*, *Priority*, *Period*, *Runnable*, *Task Mapping*, *Execution Time*. The user-entered parameters are implemented by a block mask, which provides the parameter dialogue box.

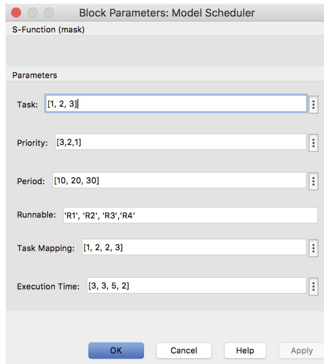


Fig. 2. Model scheduler parameters

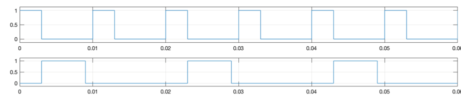


Fig. 3. Task active chart

A simple example shown in Fig. 4 illustrates the usage of our model scheduler. In this example, Model Scheduler schedules three runnables and they are mapped to two tasks. Each subsystem represents an AUTOSAR runnable. In general, the execution order of specific Simulink subsystem is determined by a Stateflow scheduler. In our case, the Stateflow scheduler is replaced by our tool Model Scheduler and each Simulink subsystem is scheduled to be executed at a specified time with a finite execution time. Model Scheduler yields three outputs. The first output port is the trigger signals that periodically output the specified time for each subsystem. The trigger signals are connected to a demux block which splits the multiple trigger signals to a single signal to trigger each subsystem. The second output port is a runnable activation chart that illustrates each runnable schedule. It shows the start and finish time of each runnable including the runnable execution time. The third output port is a task activation chart that illustrates each task schedule.

Model Scheduler is based on an FPS algorithm and implemented as a level 2 S-function written in C. Our algorithm takes as arguments the six input parameters mentioned above. The parameters can be grouped into two levels, one

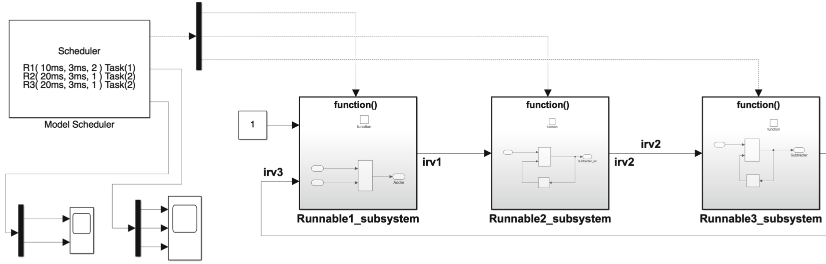


Fig. 4. A simple example of using model scheduler to schedule AUTOSAR SW-Cs.

describes the properties of a task such as *Priority* and *Period* the other one describes the properties of a runnables such as *Task Mapping* and *Execution Time*. Model Scheduler reserves this two-level information and computes the current active runnable signal. In this work, we assume execution time of each runnable is already known. The execution time could either be measured by running the code on a test platform, or by analysing the behaviour of generated code (or Simulink model) by off-the-shelf tools. Normally the FPS algorithm computes a scheduling table, then a runtime dispatching algorithm invokes each task according to the precomputed table.

Simulating a model has three phases: model compilation, link phase and simulation loop phase [10]. The Simulink engine each time goes through the loop is called as one simulation step. In each simulation step, Model Scheduler is executed and computes the running task and runnable of current sampling time and yields a signal to the output port when the current sample time is a beginning of a task period. The output signal triggers the connected subsystem, which is a runnable of the current execution task. The model scheduler determines the current active runnable along with the associated task information at each single simulation step. If a task or a runnable is expected to run at this simulation step, then model scheduler invokes a macro to trigger the subsystem connected to Model Scheduler.

Let us see an example of how Model Scheduler performs the schedule computation. The simple example (Fig. 4) has the following settings shown in Table 2. During the simulation loop phase, task T_1 and T_2 are all scheduled at the first simulation step and Model Scheduler maintains a scheduling table to store the scheduled tasks. T_1 is the only executed task at the first simulation step due to its higher priority and the execution of T_1 takes 3 ms as only runnable R_1 is mapped to T_1. In the first simulation step, Model Scheduler output a function-call signal to trigger T_1 that is connected to the first output port of demux *Runnable1_subsystem*. There is no output signals at the simulation step two because it is still during the execution of T_1. Until simulation step three, T_1 completes its execution and it is time to trigger T_2. R_2 and R_3 are mapped to T_2 so they have the same priority and period. R_2 is executed at this simulation step because its connection order is before R_3. R_3 is executed right after the

completion of R_2. After execution of R_3, that is an idle time so there is no trigger signal being output. Figure 3 illustrates the task execution process during each simulation step. T_1 is active during the first three simulation steps and T_2 is active at the following six simulation steps in the first 10 ms period.

Table 2. The simple example settings

Task	Period (ms)	Execution time (ms)	Priority	Runnable
T_1	10	3	2	R_1
T_2	20	3	1	R_2
T_2	20	3	1	R_3

5 Case Study

In this section, we use AUTOSAR compliant ML/SL models scheduled by our model scheduler to show the scheduler can capture the actual behaviour on a hardware platform during a Simulink simulation.

5.1 AUTOSAR Model Scheduling

First, we demonstrate a simple example that shows how Model in the Loop (MIL) analysis benefits from our model scheduler. Figure 5 shows a simple example using a model scheduler to schedule four runnables mapped to three tasks. The details of settings are shown in Table 3.

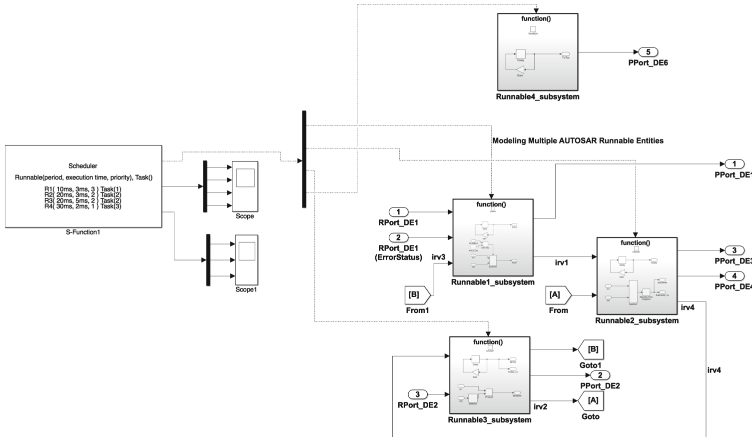


Fig. 5. Using model scheduler to schedule AUTOSAR SW-Cs.

Table 3. Using model scheduler parameters setting

Task	Period (ms)	Execution time (ms)	Priority	Runnable
T_1	10	3	3	R_1
T_2	20	3	2	R_2
T_2	20	5	2	R_3
T_3	30	2	1	R_4

In this example, we have four runnables which are mapped to three tasks: R_1 is mapped to T_1; R_2 and R_3 are mapped to T_2; R_4 is mapped to T_3. The model scheduler takes parameters of tasks as input to calculate three outputs. The first output is the runnable triggers which are connected to four subsystems accordingly. The other two outputs are time execution diagrams of tasks and runnables.

If we simulate this example using a standard scheduler, the execution order of this example is $T_1T_2T_3$ or $R_1R_2R_3R_4$ respectively and they are completed within the first 10 ms period based on tasks settings. In reality the above order is not possible, since each task requires a certain amount of execution time on a hardware platform. Our scheduler tries to be realistic by taking execution time into account.

Figure 6 shows runnables execution diagram of our scheduler. There are four output signals and each output signal represents each runnable. The first signal shows the execution of R_1 in T_1. It has the highest priority so it is trigged at the beginning and takes 3 ms execution time. After the execution of T_1, the next highest priority is T_2 with 2 runnables. R_2 is triggered at time of 3 ms and takes another 3 ms execution time. R_3 is supposed to be triggered right after the completion of R_2. However, R_3 is triggered right after the completion of the second R_1 instance because the period of T_1 is 10 ms and the execution time for both R_1 and R_2 are 3 ms. There is only 4 ms left before R_1 is triggered at next period and it is less than the execution time of R_3 of 5 ms. Because a runnable is the smallest atomic component within a SW-C, there is no preemption between runnables. R_3 cannot be preempted by R_1. Thus, R_3 is scheduled to be executed after the completion of second R_1 instance. The execution order of our scheduler is $T_1T_2T_1T_2T_3$ or $R_1R_2R_1R_3R_4$. Figure 7 shows the tasks execution diagram. There are three output signals represent tasks execution. During the first period of T_2, T_1 is triggered twice and T_2 is preempted by T_1.

5.2 Scheduler Example

In this section, we use an example model, which is scheduled by two different schedulers the Stateflow Scheduler and Model Scheduler, to show our scheduler is able to simulate actual behaviours during the simulation phase. The Stateflow scheduler takes zero execution time during the Simulink simulation. On the contrary the model scheduler triggers each subsystem which takes a specified

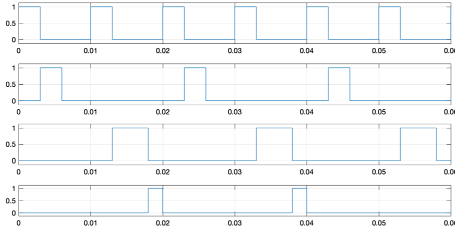


Fig. 6. Runnables execution time diagram

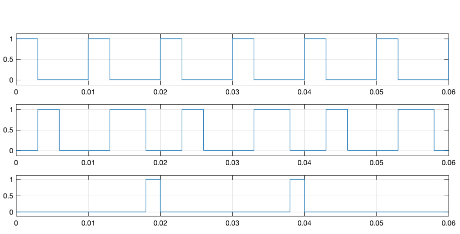


Fig. 7. Tasks execution time diagram

execution time during simulation. By comparing simulation results between these two schedulers, the potential unexpected behaviours of Simulink models are exposed.

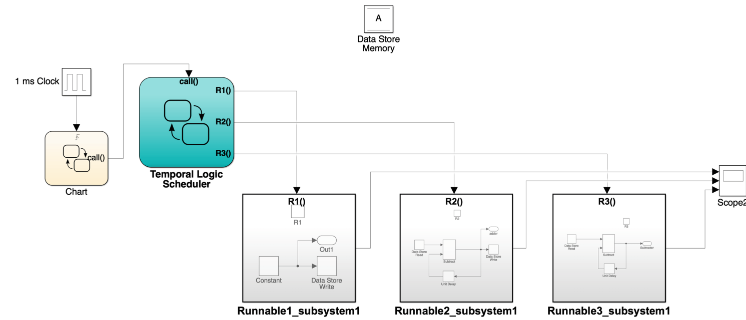


Fig. 8. An example using stateflow scheduler

Figure 8 shows the normal example which uses a Stateflow temporal logic scheduler to trigger each task. The parameter settings are the same as Table 2 except the execution time of R_3 is 5 ms. There are three runnables (R_1, R_2, R_3) mapping to two tasks (T_1, T_2) in this example. R_1 writes a constant value to a global variable A. R_2 reads A first then writes the summation of A and its delay value to A. R_3 reads A then subtracts its delay value from A, and outputs the result. Figure 9 shows the simulation output of this normal example. The three signals are the outputs of R_1, R_2, R_3 from top to bottom. From this simulation result, the output of R_3 is an increasing number. In the normal simulation, the execution order are T_1T_2 or $R_1R_2R_3$.

We replaced the Stateflow scheduler with our model scheduler and run simulation again, we can get a different result shown in Fig. 10. In the second simulation, the output of R_3 is a pattern of zero, constant value, which is different from the previous example. The execution order of this example are $T_1T_2T_1T_2$ or $R_1R_2R_1R_3$. In the previous example, R_3 always reads A which is written by R_2. Using our scheduler, R_3 reads the global variable A from the output of the second R_1 instance because T_2 is preempted by T_1 during the execution of T_2.

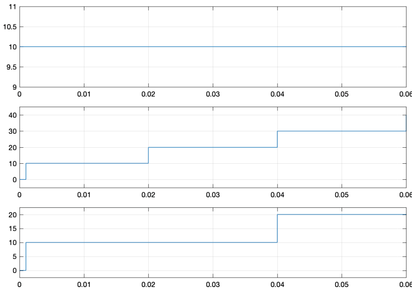


Fig. 9. Output of stateflow scheduler

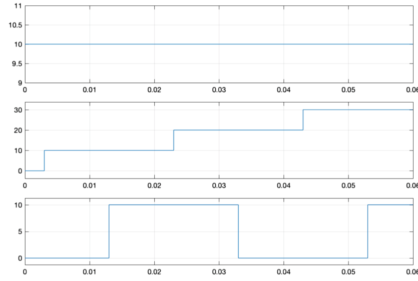


Fig. 10. Output of model scheduler

So far, we have demonstrated model scheduler is capable of simulating task interference while runnables are atomic execution. However, runnables run within the context of a task and tasks can be preempted. Hence, runnables can be preempted by runnables in other tasks. In order to show model scheduler is able to simulate the preemption at runnable level, we manually split a single runnable into several function-call subsystems so that our model scheduler treat these subsystems as "runnnables" to simulate runnables preemptions. This splitting example is depicted in Fig. 11. When we set the parameters accordingly, we can simulate the preemptions at the runnable level.

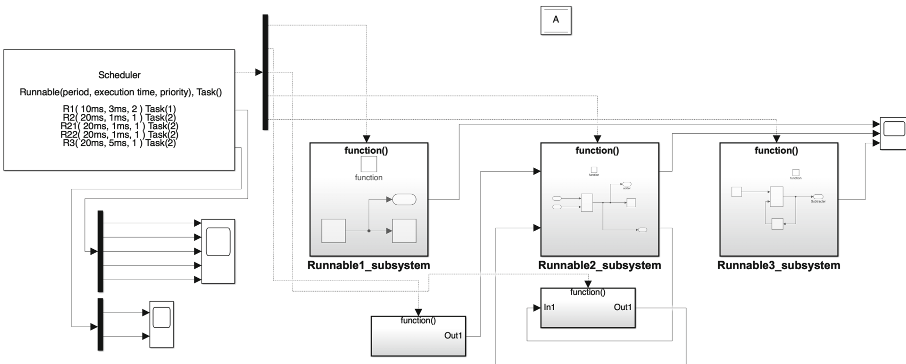


Fig. 11. Splitting a single runnable into three subsystems

6 Limitations

One of the shortcomings of model scheduler is that our model scheduler does not yet support runnable preemption automatically. In reality, runnables can be preempted by another task during its execution. If we can simulate this scenario during simulation phase automatically, it can increase the confidence of design and reduce human effort. Since the time for each block is an estimation, it may

not accurately represent the real time of the system. Multiple simulations with different time parameters may be needed to cover the possible behaviours of the system.

7 Future Work

We plan to automatically transform Simulink models to subdivide a runnable into subsystems automatically. Additionally, our current model scheduler only support periodic events. Both periodic and aperiodic tasks exist in real-time system and aperiodic events are necessary in automotive software. One possible area of expansion is to support aperiodic events in our model scheduler. Further, we could add more real-time scheduling algorithms such as Earliest-deadline-first (EDF) scheduling to model scheduler so that engineers can verify the design under different scheduling algorithms to meet the requirements of diverse target platforms.

We also plan to use the modified models to identify interference between tasks. Currently, our model scheduler requires execution time as parameters to perform simulation so that we can find potential issues during the simulation phase. In the future, we perform a model scheduler simulation based on the input parameters, we can model the execution times as variables inside the scheduler and change the value of execution until we find a potential interference.

8 Conclusion

Model scheduler is able to schedule Simlulink models in a more realistic way so that ML/SL simulation can reflect the real-time execution on the target platform. This was implemented in an S-Function block based on FSP algorithm written in C. Model scheduler can manage the hierarchy of tasks and runnables, moreover runnables are scheduled according to the tasks parameters. We have demonstrated a few simple examples to show the abilities of model scheduler. The approach discussed in this paper enables that ML/SL simulation takes software execution time into account without any modification to the current models. It can fill in the gap between the semantics of model simulation and its real-time execution.

References

1. AUTOSAR: AUTOSAR development partnership (2018). http://www.autosar.org
2. Cremona, F., Morelli, M., Di Natale, M.: TRES: a modular representation of schedulers, tasks, and messages to control simulations in simulink. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 1940–1947 (2015)
3. Derler, P., Naderlinger, A., Pree, W., Resmerita, S., Templ, J.: Simulation of LET models in simulink and ptolemy. In: Choppy, C., Sokolsky, O. (eds.) Monterey Workshop 2008. LNCS, vol. 6028, pp. 83–92. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12566-9_5

4. Ferrari, A., Di Natale, M., Gentile, G., Reggiani, G., Gai, P.: Time and memory tradeoffs in the implementation of AUTOSAR components. In: Proceedings of 2009 Design, Automation & Test in Europe Conference & Exhibition, pp. 864–869. IEEE, April 2009
5. Henriksson, D., Cervin, A., Årzén, K.E.: TrueTime: real-time control system simulation with MATLAB/Simulink. In: Proceedings of the Nordic MATLAB Conference (2003)
6. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: a time-triggered language for embedded programming. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 166–184. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45449-7_12
7. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. **30**(9), 1175–1185 (1990)
8. Lehoczky, J., Sha, L., Ding, Y.: The rate monotonic scheduling algorithm: exact characterization and average case behavior. In: Proceedings of Real-Time Systems Symposium, pp. 0–5 (1989)
9. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment scheduling algorithms for multiprogramming. J. Assoc. Comput. Mach. **20**(1), 46–61 (1973)
10. MathWorks: Developing S-Functions, R2017b (2017). http://www.mathworks.com
11. MathWorks: Simulink User's Guide, R2017b (2017). http://www.mathworks.com
12. MathWorks: Stateflow User's Guide, R2017b (2017). http://www.mathworks.com
13. Naderlinger, A.: Simulating preemptive scheduling with timing-aware blocks in Simulink. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 758–763. IEEE, March 2017
14. Naderlinger, A., Templ, J., Pree, W.: Simulating real-time software components based on logical execution time. In: Proceedings of the 2009 Summer Computer Simulation Conference, SCSC 2009, Vista, CA, Society for Modeling & Simulation International, pp. 148–155 (2009)
15. The AUTOSAR Consortium: Applying Simulink to AUTOSAR, R3.1 (2006)
16. The AUTOSAR Consortium: AUTOSAR Methodology, R4.3 (2018)
17. The AUTOSAR Consortium: The AUTOSAR Standard, R4.3 (2018)
18. Zeng, H., Di Natale, M.: Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms. In: SIES 2011–6th IEEE International Symposium on Industrial Embedded Systems, Conference Proceedings, pp. 140–149. IEEE, June 2011

Trace Comprehension Operators for Executable DSLs

Dorian Leroy[1(✉)], Erwan Bousse[2], Anaël Megna[3,4], Benoit Combemale[3], and Manuel Wimmer[2]

[1] JKU Linz, Linz, Austria
dorian.leroy@jku.at
[2] TU Wien, Vienna, Austria
[3] University of Toulouse, CNRS IRIT, Toulouse, France
[4] Safran SA, Paris, France

Abstract. Recent approaches contribute facilities to breathe life into metamodels, thus making behavioral models directly *executable*. Such facilities are particularly helpful to better utilize a model over the time dimension, *e.g.,* for early validation and verification. However, when even a small change is made to the model, to the language definition (*e.g.,* semantic variation points), or to the external stimuli of an execution scenario, it remains difficult for a designer to grasp the impact of such a change on the resulting execution trace. This prevents accessible trade-off analysis and design-space exploration on behavioral models. In this paper, we propose a set of formally defined operators for analyzing execution traces. The operators include dynamic trace filtering, trace comparison with *diff* computation and visualization, and graph-based view extraction to analyze cycles. The operators are applied and validated on a demonstrative example that highlight their usefulness for the comprehension specific aspects of the underlying traces.

Keywords: Model execution · Domain-specific languages
Executable DSL · Execution trace · Trace analysis

1 Introduction

A large amount of DSLs are used to represent behavioral aspects of systems in the form of *behavioral models* (*e.g.,* [1–5]). To better appreciate how such models unfold over the time dimension, a lot of efforts have been made to facilitate the design of so-called *executable DSLs* (*e.g.,* [6–12]), which enable the execution of conforming models using *execution semantics*. Two approaches are commonly used to define the execution semantics of an executable DSL, namely operational semantics (*i.e.,* interpretation) and translational semantics (*i.e.,* compilation). We focus in this paper on executable DSLs defined with operational semantics, and more precisely with discrete-event operational semantics.

Executing a model gives the possibility to observe the evolution of its *state* over time, *i.e.,* the *trace* of the execution [13]. Once an execution trace has been

© Springer International Publishing AG, part of Springer Nature 2018
A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 293–310, 2018.
https://doi.org/10.1007/978-3-319-92997-2_19

captured (*e.g.,* by instrumenting the model interpreter), it can be exploited in several development contexts, such as providing prompt feedback to the modeler, understanding the fault revealed by a failed test case, or performing complex automated dynamic analyses of the considered traces. In particular, in the context of design-space exploration, *trade-off analyses* of different design choices can be achieved by comparing the traces resulting from executing different variants of the model. This is especially important as even the smallest change in the model (*e.g.,* changing a guard on a transition), in the considered execution scenario (*e.g.,* exchanging the order of two signals sent to an UML state machine), or in the language definition (*e.g.,* semantic variation points) can lead to a completely different execution trace.

However, it remains difficult for a modeler to grasp the impact of a design change on the resulting execution trace. Comparing execution traces is often hampered by noisy or redundant data captured in execution traces, which leads to finding irrelevant differences between the traces. In particular, it is often compulsory to filter out extraneous dynamic information from an execution trace, in order to focus only on the changes occurring in a relevant subset of the model state. In addition, due to the sequential nature of a trace, it is laborious to visualize which model states were explored multiple times during the execution of a behavioral model, and between which states the model may have oscillated, *e.g.,* in order to discover potential cycles or bottlenecks.

To address these problems, we propose in this paper a set of formally defined *trace comprehension operators*. These operators can be used for dynamic information filtering, trace comparison with *diff* computation and visualization, and graph-based views extraction to analyze cycles. Some operators can be combined for better results, *e.g.,* to extract a graph-based view out of filtered traces. We provide a formalization of each operator, and we implemented them as part of the GEMOC Studio[1], an Eclipse-based language and modeling workbench. We validate the approach by demonstrating the relevance of the operators for model variants conforming to a State Machines DSL inspired by UML State Machines.

The remainder of the paper is structured as follows. Section 2 presents the scope of considered DSLs and execution traces, and a motivating example. Section 3 shows an overview of the proposed solution. Section 4 describes the formalization of the trace comprehension operators. Section 5 presents the implementation and the validation of our approach with a use case based on a State Machine DSL. Section 6 presents related work. Finally, Sect. 7 concludes.

2 Background and Motivating Example

In this section, we first precisely scope the executable DSLs considered in our approach, *i.e.,* metamodel-based DSLs with discrete-event operational semantics. We then present the considered execution traces, and finally we present a motivating example based on a State Machines DSL.

[1] https://eclipse.org/gemoc.

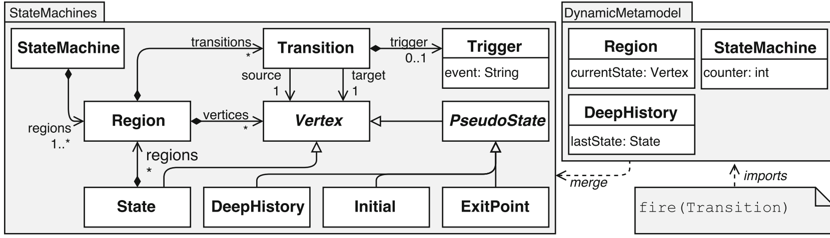


Fig. 1. The State Machines executable DSL.

2.1 Considered Executable DSLs

An executable DSL is composed of both an abstract syntax defining the concepts of the domain, and an execution semantics defining how these concepts are executed. In this paper, we focus on executable DSLs whose abstract syntax is a *metamodel*, and whose execution semantics is an *operational semantics*. A metamodel is a class-based object-oriented model defined using a metamodeling language (*e.g.,* MOF [14] or Ecore [15]) composed of a set of metaclasses. A metaclass is composed of a set of *properties*, each either an attribute (typed by a datatype, *e.g.,* , integer) or a reference to another metaclass.

The left part of Fig. 1 shows the abstract syntax of an example of State Machines DSL, which is directly inspired from UML State Machines. A StateMachine contains at least one Region. A Region contains Vertex elements, which can be State elements or PseudoState elements. PseudoState elements are further refined into Initial, ExitPoint and DeepHistory elements. Finally, a Region also contains Transition elements which point to a `source` and a `target` Vertex. Transition elements contain Trigger elements, which may each possess an `event`.

Next, we decompose the *operational semantics* of an executable DSL in two parts: a data structure representing the state of the executed model, and a model transformation altering the model state. We define this data structure as a metamodel which extends the abstract syntax with new dynamic metaclasses and properties using *package merge*. Executing the model consists in applying the model transformation of the semantics, which performs an endogenous, possibly in-place, transformation on the model state. We do not make assumptions on the language used to define the model transformation, nor on the content of the transformation. Instead, we only consider that it produces a sequence of changes in the state of the model, each change caused by a given *observable event* (*e.g.,* a rule call for rule-based languages, or a method call for imperative languages).

The upper right part of Fig. 1 shows the metamodel of the operational semantics extending the abstract syntax with new dynamic properties: the `currentState` property in Region is used to track the current state of the Region during the execution, the `lastState` property in DeepHistory stores the last visited state in the owning Region element, and finally the `counter` counts the number of fired transitions during the execution. Finally, we consider that the

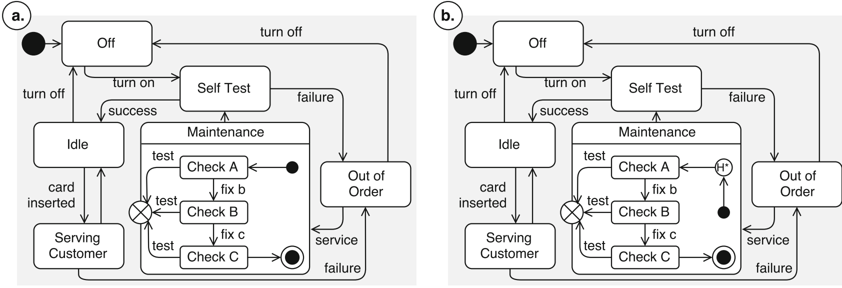


Fig. 2. Two ATM state machine variants: *a* without history and *b* with history.

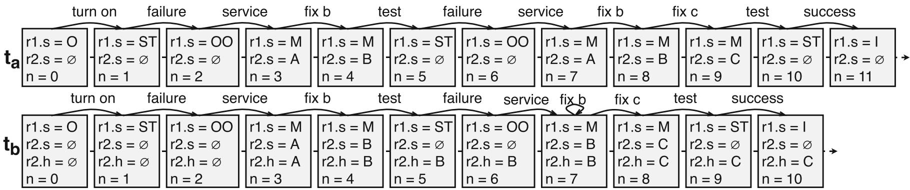


Fig. 3. Traces from the example models: *a* is without history, *b* is with history.

only observable event of the DSL is the firing of Transition elements. Thus, the execution semantics only defines a `fire` event.

2.2 Considered Execution Traces

At runtime, an executed model is composed of a set of objects, each object being an instance of a metaclass of the DSL. An object assigns one value per property of the corresponding metaclass. A *model state* is a recording of all values assigned to dynamic properties—*i.e.*, the properties added by the operational semantics—of the executed model at a certain point of the execution.

We call *execution trace* a sequence containing all model states reached during an execution and all observed execution events. A trace is obtained by recording the execution of a behavioral model. In a trace, we call *dimension* the sequence of all values assigned to a specific dynamic property by an object over the execution. As our previous work [13,16], considering a trace as a set of dimensions is central to our approach, as it gives the possibility to efficiently manipulate the parts of a trace related to specific dynamic properties. We present examples of traces in the following subsection.

2.3 Motivating Example

Figure 2 depicts two models conforming to the State Machines DSL shown in Fig. 1. Both models represent the behavior of a *cash dispenser*, also called *ATM*, with states such as *Idle* or *Serving Customer*. Transitions represent how the ATM

switches mostly between idling, maintenance and service states. The difference between the two models lies in the added deep history pseudostate in the region of the *Maintenance* state. The semantics of the deep history pseudostate is that it stores the last visited state of its containing region and, when targeted by a transition, restores this state as the current state of the region. Adding such a pseudostate can thus affect greatly how the execution unfolds, and predicting the impact of such a change can be difficult.

Figure 3 shows two execution traces resulting from the execution of the models in Fig. 2 with the following sequence of stimuli: *turn on*, *failure*, *service*, *fix b*, *test*, *failure*, *service*, *fix b*, *fix c*, *test*, *success*. In these traces, `r1` refers to the Region element owned by the state machine and `r2` to the Region element owned by the *Maintenance* state. The `s` property refers to the current state of a Region element, and the `h` refers to the last state of a DeepHistory element. Finally, the `n` property refers to the counter of fired transitions of the state machine. The name of the states are abbreviated for space reasons.

By looking at the execution traces shown in Fig. 3, we can glimpse that if we were ignoring the dimensions `counter` and `lastState`, then many similarities between the two traces could be found. For instance, the states ⟨*Maintenance*, *Check C*, 9⟩ and ⟨*Maintenance*, *Check C*, *Check C*, 8⟩ would then be equivalent. Even in a single trace, the value of the `counter` is different in each model state, which make it hard to identify possible cycles encountered in the states of the State Machine. For instance, t_a features a cycle that can only be detected if the `counter` dimension is ignored: ⟨*Maintenance*, *Check B*, 4⟩ and ⟨*Maintenance*, *Check B*, 8⟩ are two states part of this cycle.

In summary, even with small models with little dynamic information, and with only small changes between model variants, it is already challenging to understand and to compare the resulting execution traces. A similar observation could be made if small changes were made to the semantics of the considered DSL, or to the stimuli of the considered execution scenario. In this context, to ease the task of understanding execution traces, our contribution is a set of four *trace comprehension operators*. We give a brief presentation of these operators in Sect. 3, before defining them formally in Sect. 4.

3 Approach Overview

In this section, we present an overview of the contribution of this paper, *i.e.,* four different and complementary trace comprehension operators. Figure 4 summarizes the application context including the inputs and outputs of the different operators. On the left, two behavioral models named A and B are shown, and a trace is obtained for each of their executions. Then, four different operators can be used to manipulate the obtained traces:

- The *Filter* operator takes an execution trace as input and produces a refined version of the input execution trace as output. It removes a selected set of dimensions (see Sect. 2.2 for the definition of dimension) from the input trace,

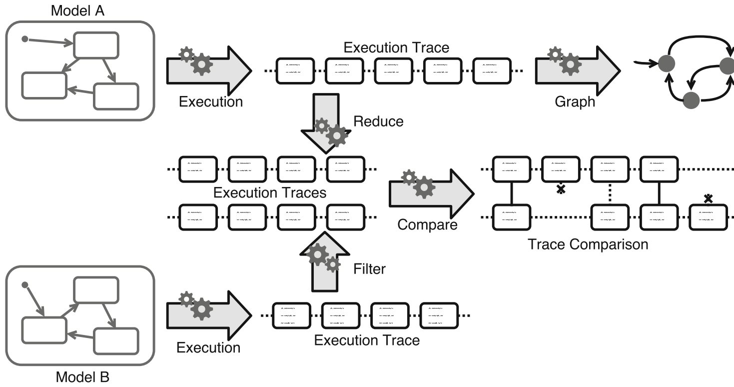


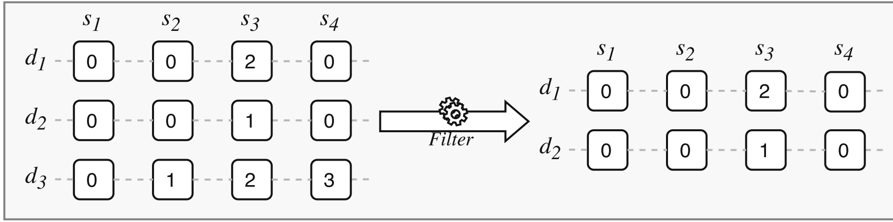
Fig. 4. Overview of a possible workflow using all four proposed operators.

which results in a simplified trace that only reflects the evolution of a subset of the model state. Note that *Filter* does not change the amount of model states in the trace, and only changes the content of each model state.
– The *Reduce* operator also takes an execution trace as input and produces a refined version of the input execution trace as output, where each subsequence of successive identical model states is merged into a single model state. *Reduce* is particularly useful when applied after the *Filter* operator when the only differences between the states of a sequence of successive states were found in the dimensions that were filtered out.
– The *Compare* operator takes two execution traces as input and produces a trace difference model as output. This difference model highlights all the changes that occurred between the first trace and the second one: which states were added, or removed, or substituted by other states. Such comparison can be used to better understand the impact of a design change on the trace resulting from the execution.
– The *Graph* operator takes an execution trace as input and produces a state graph as output. This state graph is a representation of all different model states reached during the execution. Among other benefits, such higher-level view provides a better global understanding of the execution, and can highlight cycles and bottleneck states.

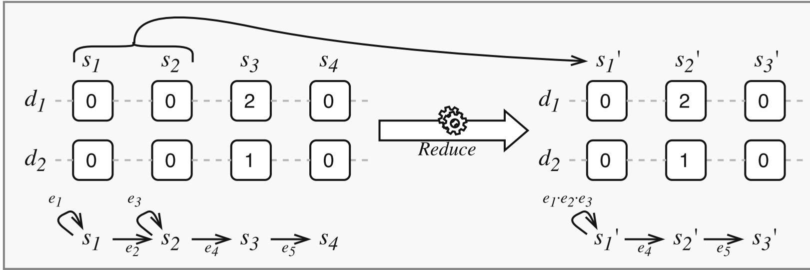
The middle and right part Fig. 4 show a typical workflow where the traces obtained from the models are simplified through the use of the *Filter* and *Reduce* operators, before being used as input for the *Graph* and *Compare* operators. In the following section, we provide a formal specification of these four operators.

4 Operators for Execution Trace Comprehension

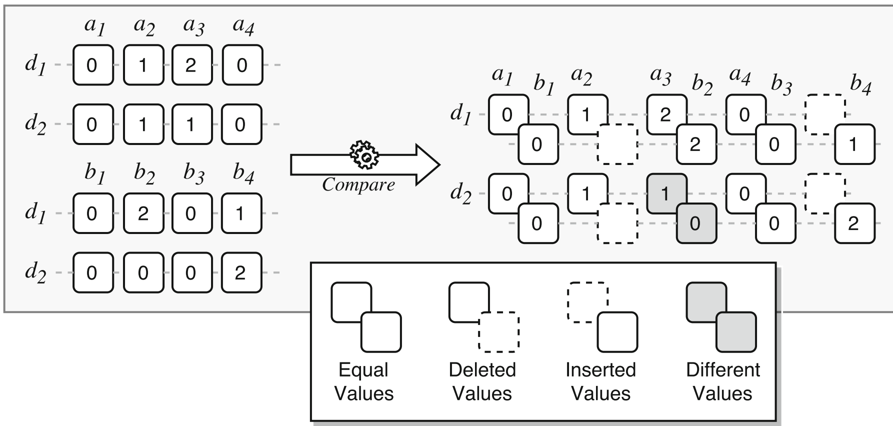
In this section we present our contribution, *i.e.,* a set of four trace comprehension operators. Figure 5 summarizes graphically all proposed operators using abstract



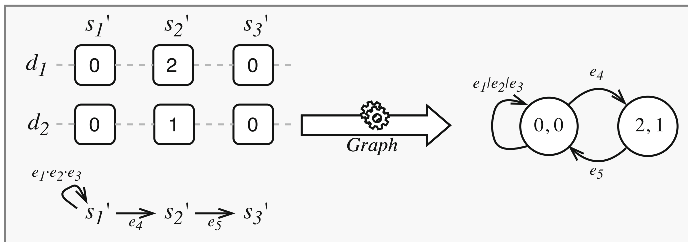
(a) *Filter* operator, which removes dimensions from a trace.



(b) *Reduct* operator, which factorizes redundant states.



(c) *Compare* operator, which produces a trace difference.



(d) *Graph* operator, which extracts a state graph from a trace.

Fig. 5. Graphical summary of all four trace comprehension operators.

examples, and will be used throughout the section to illustrate the operators. In what follows, we first formally define what is a execution trace, then we provide a formal definition of each trace comprehension operator.

4.1 Execution Trace Formalization

In order to give a formal definition of operators that manipulate execution traces, we must first formally define the concept of trace. In the remainder of the paper, we denote T the set of all execution traces.

Definition 1 (Trace). A trace is a tuple $\langle S, D, E_<, val, step\rangle$ where:

- S is the set of model states of the execution trace.
- D is the set of dimensions of the execution trace.
- $E_< = (E, <_E)$ is the totally ordered set of events that occurred during the execution where, $\forall e_1, e_2 \in E, e_1 <_E e_2$ if e_1 happens before e_2.
- $val : (S \times D) \rightarrow V$ is the function mapping a model state and a dimension to a value. Using val, we define a state equivalence relation $Eq \subseteq S \times S$ as $(a, b) \in Eq \Leftrightarrow \forall d \in D, val(a, d) = val(b, d)$, denoted $a \equiv b$.
- $step : E_< \rightarrow (S \times S)$ is the function mapping an event to a starting an ending state. Note that an event *can* have the same starting and ending state, which means that the model state did not change due to the event occurrence. We denote:
  - $a \xrightarrow{e} b$ the fact that $step(e) = (a, b)$,
  - $a \xrightarrow{*} b$ the fact that $step$ can lead from a to b with a sequence of events, *i.e.*:

$$\exists e \in E_<, a \xrightarrow{e} b \vee \exists n \in \mathbb{N}, \exists e_1, ..., e_n \in E_<, \exists s_1, ..., s_{n-1} \in S,$$
$$a \xrightarrow{e_1} s_1 \xrightarrow{e_2} ... \xrightarrow{e_{n-1}} s_{n-1} \xrightarrow{e_n} b \wedge \forall i \in]1; n], e_{i-1} <_E e_i,$$

  - $a \rightarrow b$ the fact that $a \neq b \wedge \exists e \in E_<, a \xrightarrow{e} b$, *i.e.,* a directly precedes b,
- for any two states a and b, there is an ordered sequence of events that lead from a to b or from b to a, *i.e.,* $\forall a, b \in S, a \xrightarrow{*} b \vee b \xrightarrow{*} a$.
- the total order on events $<_E$ combined with the $step$ function create a total order $<_S$ over the states defined as: $\forall a, b \in S, a <_S b \Leftrightarrow a \xrightarrow{*} b$ We denote $s = S_i$ the fact that $|\{s' \in S : s' <_S s\}| = i$

Example 1. Using only natural integer values (*i.e.,* $V = \mathbb{N}$), and events e_i ordered by their index i, let t_{ex} be an execution trace conforming to Definition 1:

$$\begin{aligned} t_{ex} &= \langle \{s_1, s_2, s_3, s_4\}, \{d_1, d_2, d_3\}, \{e_1, e_2, e_3, e_4, e_5\}, val, step\rangle \\ &\text{where } val(s_1, d_1) = 0 \quad val(s_2, d_1) = 0 \quad val(s_3, d_1) = 2 \quad val(s_4, d_1) = 0 \\ &\qquad val(s_1, d_2) = 0 \quad val(s_2, d_2) = 0 \quad val(s_3, d_2) = 1 \quad val(s_4, d_2) = 0 \\ &\qquad val(s_1, d_3) = 0 \quad val(s_2, d_3) = 1 \quad val(s_3, d_3) = 2 \quad val(s_4, d_3) = 3 \\ &\text{and } s_1 \xrightarrow{e_1} s_1 \quad s_1 \xrightarrow{e_2} s_2 \quad s_2 \xrightarrow{e_3} s_2 \quad s_2 \xrightarrow{e_4} s_3 \quad s_3 \xrightarrow{e_5} s_4 \end{aligned}$$

4.2 Dimension Filtering

When an operational semantics introduces a large amount of dynamic properties, or when the executed model is very large, an execution trace may contain a large amount of dimensions to grasp. Yet, understanding specific aspects of the behavior might only require looking of a specific subset of dimensions of interest. For this purpose, our first operator is called *Filter* (see Fig. 5a), and aims at removing dimensions out of a trace in order to simplify it. This operator is in fact an abstraction operator on the model states contained in the trace.

Definition 2 (*Filter*). Given an input trace $\langle S, D, E_<, val, step\rangle$ and an input set of dimensions I, the *Filter* operator is defined as:

$$\begin{array}{rrcl} \mathit{Filter}: & (T \times \mathcal{P}(D)) & \rightarrow & T \\ & (\langle S, D, E_<, val, step\rangle, I) & \mapsto & \langle S, D', E_<, val', step\rangle \end{array}$$

where $D' = D \setminus I$ and $val' : S \times D' \rightarrow V$ is defined as $val'(s, d') = val(s, d')$.

Example 2. We apply *Filter* to the trace t_{Ex} and dimension d_3 from Example 1:

$$\mathit{Filter}(t_{Ex}, \{d_3\}) = \langle\{s_1, s_2, s_3, s_4\}, \{d_1, d_2\}, \{e_1, e_2, e_3, e_4, e_5\}, val', step\rangle$$

$$\begin{array}{llll} \text{where } val'(s_1, d_1) = 0 & val'(s_2, d_1) = 0 & val'(s_3, d_1) = 2 & val'(s_4, d_1) = 0 \\ \quad val'(s_1, d_2) = 0 & val'(s_2, d_2) = 0 & val'(s_3, d_2) = 1 & val'(s_4, d_2) = 0 \end{array}$$
$$\text{and } s_1 \xrightarrow{e_1} s_1 \quad s_1 \xrightarrow{e_2} s_2 \quad s_2 \xrightarrow{e_3} s_2 \quad s_2 \xrightarrow{e_4} s_3 \quad s_3 \xrightarrow{e_5} s_4$$

Note that $s_1 \equiv s_2$ and $s_1 \rightarrow s_2$, *i.e.,* two successive model states are identical. The next operator will enable the merging of these states to obtain a more compact trace, *i.e.,* where a state is always different from the preceding state.

4.3 Trace Reduction

When using a trace recorder that always records the model state at each occurring observable event without checking if the state has changed, or when using the *Filter* operator introduced above, a trace may contain successive equivalent states which are redundant and can be considered as superfluous data. This phenomenon is also known as stuttering [17]. To simplify such traces, we propose an operator *Reduce* (see Fig. 5b) which merges such successive equivalent states while preserving the behavior depicted by the trace.

Definition 3 (*Reduce*). The *Reduce* operator is defined as:

$$\begin{array}{rrcl} \mathit{Reduce}: & T & \rightarrow & T \\ & \langle S, D, E_<, val, step\rangle & \mapsto & \langle S', D, E_<, val', step'\rangle \end{array}$$

where:

– S' is the set of sets of successive equivalent states of S, *i.e*:

$$S' = \{s \in \mathcal{P}(S) : \forall a \in s, \forall b \in S, a \equiv b \land a \rightarrow b \Rightarrow b \in s\}$$

– $step' : E_< \rightarrow (S' \times S')$ is defined as: $step'(e) = \langle A, B\rangle \Leftrightarrow step(e) \in (A \times B)$
– $val' : (S' \times D) \rightarrow V$ is defined as $val'(B, d) = val(a, d)$ for any $a \in B$

Hence, each output state of S' is composed of (and thus replaces) a set of equivalent successive states of S, and both $step'$ and val' are adjusted accordingly.

Example 3. Resulting trace from $Reduce(Filter(T_{Ex}, \{d_3\}))$.

$$\begin{aligned} &Reduce(Filter(T_{Ex})) = \\ &\quad \langle \{s'_1 = \{s_1, s_2\}, s'_2 = \{s_3\}, s'_3 = \{s_4\}\}, \{d_1, d_2\}, \{e_1, e_2, e_3, e_4, e_5\}, val', step'\rangle \\ &\quad \text{where } val'(s'_1, d_1) = 0 \quad val'(s'_2, d_1) = 2 \quad val'(s'_3, d_1) = 0 \\ &\quad val'(s'_1, d_2) = 0 \quad val'(s'_2, d_2) = 1 \quad val'(s'_3, d_2) = 0 \\ &\quad \text{and } s'_1 \xrightarrow{e_1} s'_1 \quad s'_1 \xrightarrow{e_2} s'_1 \quad s'_1 \xrightarrow{e_3} s'_1 \quad s'_1 \xrightarrow{e_4} s'_2 \quad s'_2 \xrightarrow{e_5} s'_3 \end{aligned}$$

The soundness of *Reduce* can easily be proven, *i.e.,* the fact that two successive states of S' cannot be equivalent, and that one state of S is only mapped to a single state of S'. These properties can be rephrased as two theorems:

Theorem 1. $Reduce(T) = \langle S', _, _, _, _\rangle \Rightarrow \forall s_1, s_2 \in S', s_1 \rightarrow s_2 \Rightarrow s_1 \not\equiv s_2.$

Theorem 2. $Reduce(T) = \langle S', _, _, _, _\rangle \Rightarrow \bigcap_{s \in S'} s = \emptyset.$

4.4 Trace Comparison

As understanding a single execution trace is already a difficult task, grasping the differences between two execution traces is even more challenging and error-prone. To address this problem, we propose a *Compare* operator that produces a *trace difference* showing the similarities and dissimilarities between two traces. Note that since traces may come from different models (*e.g.,* an original and a revised one), each trace may possess its own set of dimensions, hence *Compare* requires an explicit mapping between the dimensions of the first and second traces.

Our comparison procedure relies on the notorious *Levenshtein distance* [18], which is an operator counting the minimal number of insertion, deletion or substitution operations required to transform one string into another. For instance, the Levenshtein distance between "*S*TR**ING**" and "TR**ACE**" is four, which is computed by summing the number of insertions in *italics* and of substitutions in **bold**. While the output of the *Levenshtein distance* is an integer, computing this distance requires computing all the distances between all the possible prefixes of the input strings (*i.e.,* substrings starting with the first character). It is then possible to infer from all these distances the exact set of insertions, deletions or substitutions required to transform the first string into the second string, which

is the kind of information we require to construct a trace difference. For our work, we adapted the Levenstein distance to compare traces instead of strings, where model states play the role of characters, which can be compared using the equivalence relation.

Definition 4 (Levenshtein distance on traces). The Levenshtein distance between two traces $T_1 = \langle A, _, _, _, _ \rangle$ and $T_2 = \langle B, _, _, _, _ \rangle$ is given by $lev_{T_1,T_2}(|A|, |B|)$ where:

$$lev_{T_1,T_2}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0, \\ min \begin{cases} lev_{T_1,T_2}(i-1,j)+1 \\ lev_{T_1,T_2}(i,j-1)+1 \\ lev_{T_1,T_2}(i-1,j-1)+1_{A_i \not\equiv B_j} \end{cases} & \text{otherwise} \end{cases}$$

Where $1_{A_i \not\equiv B_j}$ equals 0 when $A_i \equiv B_j$, and equals 1 otherwise.

As we can see, to obtain the Levenstein distance $lev_{T_1,T_2}(|A|, |B|)$, we rely on a recursive operator $lev_{T_1,T_2}(i,j)$ which computes the distance between the subsequence of states $[0, i]$ of T_1 and the subsequence of states $[0, j]$ of T_2. These distances can be used to infer the insertions, deletions and substitutions required to go from the first trace to the second. In that goal, we define the following notations on top of *lev*:

- $in_{T_1,T_2}(i,j)$ denotes $lev_{T_1,T_2}(i,j) = lev_{T_1,T_2}(i,j-1)+1$,
- $del_{T_1,T_2}(i,j)$ denotes $lev_{T_1,T_2}(i,j) = lev_{T_1,T_2}(i-1,j)+1$,
- $subst_{T_1,T_2}(i,j)$ denotes $lev_{T_1,T_2}(i,j) = lev_{T_1,T_2}(i-1,j-1)+1$.

Using this $lev_{T_1,T_2}(i,j)$ through these notations, we can define the *Diff* operator which produces a unique set containing states of T_1 that were deleted, states of T_2 that were inserted, and pairs of states from T_1 and T_2 that were substituted.

Definition 5 (*Diff*). We define the union set of inserted, deleted and pairs of substituted states identified as part of a Levenshtein distance computation as $Diff_{T_1,T_2} = DiffRec_{T_1,T_2}(|A|, |B|)$, where:

$$DiffRec_{T_1,T_2}(i,j) = \begin{cases} DiffRec_{T_1,T_2}(0,0) = \emptyset \\ DiffRec_{T_1,T_2}(i,j-1) \cup B_j & \text{if } in_{T_1,T_2}(i,j) \\ DiffRec_{T_1,T_2}(i-1,j) \cup A_i & \text{if } del_{T_1,T_2}(i,j) \\ DiffRec_{T_1,T_2}(i-1,j-1) \cup \{A_i, B_j\} & \text{if } subst_{T_1,T_2}(i,j) \\ DiffRec_{T_1,T_2}(i-1,j-1) & \text{otherwise} \end{cases}$$

Finally, we define *Compare* as a trivial projection of the output of the *Diff* operator into a tuple that separates insertions, deletions and substitutions in three different sets.

Definition 6 (*Compare*). Given two traces $T_1 = \langle A, _, D_1, _, _\rangle$ and $T_2 = \langle B, _, D_2, _, _\rangle$ and a mapping $M \subseteq \mathcal{P}(D_1 \times D_2)$, the *Compare* operator is defined as:

$$\begin{aligned} \mathit{Compare} : T \times T \times (D_1 \times D_2) &\rightarrow \mathcal{P}(B) \times \mathcal{P}(A) \times \mathcal{P}(A \times B) \\ (T_1, T_2, M) &\mapsto \langle \mathit{In}, \mathit{Del}, \mathit{Subst}\rangle \end{aligned}$$

where $\mathit{In} = \mathit{Diff}_{T_1,T_2} \cap B$, $\mathit{Del} = \mathit{Diff}_{T_1,T_2} \cap A$ and $\mathit{Subst} = \mathit{Diff}_{T_1,T_2} \cap (A \times B)$.

Note that while the comparison results are unordered, this does not prevent the presentation of the comparison result in a human-readable way. This can be done by iterating over the states of both traces in parallel, and looking for them in the trace difference. For instance, Fig. 5c was obtained from the trace difference obtained with *Compare* containing $\langle\{b_4\}, \{a_2\}, \{\langle a_3, b_2\rangle\}\rangle$ using the following reasoning:

- a_1 and b_1 are absent from the result: hence all their values are equal (first column).
- a_2 is not contained in a pair: hence it has been deleted from t_1 (second column).
- a_3 and b_2 are contained in a pair: hence some values are different from a_3 to b_2. These values can be identified by iterating over the dimension pairs that are part of the provided matching (third column).
- a_4 and b_3 are absent from the result: hence all their values are equal (fourth column).
- b_4 is not contained in a pair: hence it has been inserted in t_2 (fifth column).

4.5 State Graph Extraction

For each model state in an execution trace, there may be other *equivalent* model states scattered over the trace, which means that the execution is going back to this state several times during the execution. However, the sequential nature of a trace makes it difficult to grasp such information, and to understand the possible *cycles* in the execution trace. To provide a better understanding of the encountered model states, we propose the last operator called *Graph* (see Fig. 5d), which creates a directed graph from a trace, where each vertex is mapped to a set of equivalent states of the trace, and each event adds an edge between the vertexes containing its source and target states, if such an edge does not already exists. These edges also carry the set of events that caused their existence.

Definition 7 (*Graph*). Let G be the set of all directed graphs. The *Graph* operator is defined as:

$$\begin{aligned} \mathit{Graph} : \quad T &\rightarrow G \\ \langle S, D, E_<, \mathit{val}, \mathit{step}\rangle &\mapsto \langle V, A\rangle \end{aligned}$$

where:

- V is the set of vertices, with $V = \{s \in \mathcal{P}(S) : \forall a, b \in s, a \equiv b\}$

– A is the set of directed edges, with $A = \{\langle v_1, \mathit{Events}, v_2 \rangle \in V \times \mathcal{P}(E) \times V : \forall e \in \mathit{Events}, \exists a \in v_1, \exists b \in v_2, a \xrightarrow{e} b\}$

Example 4. Resulting graph from $\mathit{Graph}(\mathit{Filter}(T_{Ex}, \{d_3\}))$

$$\mathit{Graph}(\mathit{Filter}(T_{Ex})) = \langle \{v_1 = \{s_1, s_2, s_4\}, v_2 = \{s_3\}\}, \\ \{(v_1, \{e_1, e_2, e_3\}, v_1), (v_1, \{e_4\}, v_2), (v_2, \{e_5\}, v_1)\} \rangle$$

5 Implementation and Evaluation

In this section, we first explain how we implemented the operators as part of the GEMOC Studio. We then present how we validate the approach using the motivating example conforming to the State Machine DSL from Sect. 2.3.

5.1 Implementation Within the GEMOC Studio

We implemented the four operators within the GEMOC Studio, an open source (EPL 1.0) Eclipse package atop the Eclipse Modeling Framework. The GEMOC Studio includes a language workbench to implement executable DSLs, and a modeling workbench to create, execute and debug conforming models. We implemented a set of graphical views to display both execution traces and operators outputs (*i.e.,* traces, *diff* models and graphs) in a human-readable way.

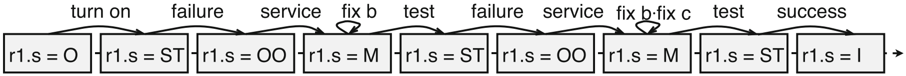
At runtime, traces commonly reach a large amount of states that must be stored in memory. Therefore, while this is out the scope of this paper, our implementation aims at limiting the amount of memory required by the trace comprehension operators. Most notably, when both the input and the output of an operator are traces (*i.e.,* with *Filter* and *Reduce*), and when the output is significantly similar to the input, we produce a *virtual trace* that contains links to the concrete input trace instead of containing values, along with information on filtered dimensions (for *Filter*) or regrouped states (for *Reduce*).

The source code of the operators can be found in the Github repository of the GEMOC execution framework[2], and more information can be found about the implementation on our companion web page[3].

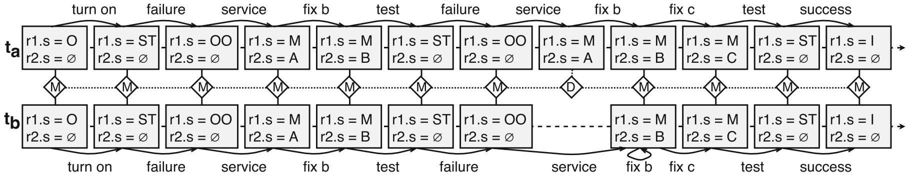
5.2 Evaluation

To evaluate the contribution of this paper, we demonstrate the usefulness of the proposed operators to understand the execution traces of the State Machine models previously shown as a motivation in Sect. 2.3. We recall that the models were depicted in Fig. 2, and the considered execution traces in Fig. 3. Figure 6 shows different applications of the four operators to the two execution traces, most of them by combining the use of multiple operators. We explain below how the results help better understand the traces.

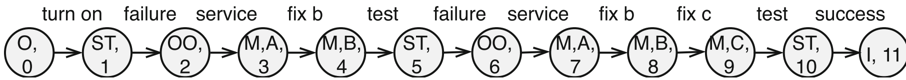
[2] https://github.com/eclipse/gemoc-studio-modeldebugging.
[3] http://gemoc.org/ecmfa18.



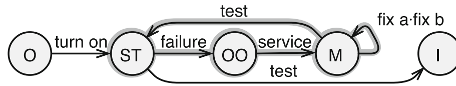
(a) $Reduce(Filter(t_a, \{\texttt{counter}, \text{Maintenance}.\texttt{currentState}\}))$



(b) $Compare(Filter(t_a, \{\texttt{counter}\}), Filter(t_b, \{\texttt{counter}, \text{Maintenance}.\texttt{lastState}\}))$



(c) $Graph(t_a)$



(d) $Graph(Reduce(Filter(t_a, \{\texttt{counter}, \text{Maintenance}.\texttt{currentState}\})))$

Fig. 6. Various applications of the operators on the traces from in Fig. 3.

***Filter* and *Reduce*.** To obtain the trace shown in Fig. 6a from t_a, we first apply *Filter* on the `counter` and Maintenance.`currentState` dimensions, then we apply *Reduce*. We choose to filter the `counter` dimension because it changes at each state and thus hampers further cycle analysis or trace comparison, and the Maintenance.`currentState` in order to hide the internal working of the *Maintenance* hierarchical state. The result is a more high-level trace which only focuses on the information of interest, *i.e.,* which states of the main state machine were visited. This demonstrates that the *Filter* and *Reduce* can be used both to get rid of noisy data (*e.g.,* the `counter`), and to modulate the level of detail featured in a trace my removing undesired dimensions (*e.g.,* Maintenance.`currentState`).

***Compare*.** To obtain the trace difference shown in Fig. 6b, we first apply *Filter* on the `counter` dimension on t_a and t_b and on the Maintenance.`lastState` dimension on t_b, then we apply *Compare* on the resulting traces. The mapping of dimensions provided to *Compare* is not shown, as it is trivial except for the deep history dimension which has no match. Figure 6b shows us that both traces align almost perfectly—except for a deleted state from one trace to the other—which was difficult to notice simply by looking at the original traces from Fig. 3. Note that this result is only possible because the traces were filtered before the

comparison, since comparing unfiltered traces would not find much similarities because of the `counter` property. This demonstrates that the *Compare* operator, especially when combined with *Filter* and *Reduce*, can effectively help to understand subtle behavioral differences induced by design choices.

Graph. To obtain the graph shown in Fig. 6c, we directly applied *Graph* on the original t_a trace. We can observe that the resulting graph is of little interest as it takes the form of a sequence identical to t_a, which is mostly due to the incremented `counter`. However, in Fig. 6d, we first applied the *Filter* operator on t_a to filter out the `counter` and Maintenance.`currentState` dimensions, followed by the *Reduce* operator. Applying *Graph* on the resulting trace shows us a better overview of the states visited during the execution. In particular, we can observe a *cycle* in the visited states, highlighted in gray. This demonstrates that the *Graph* operator, especially when combined with *Filter* and *Reduce*, can effectively help understanding which model states were visited in the execution, and which cycles can be observed between model states.

Additional Material. Our companion web page (See footnote 3) extends this evaluation with more complex models conforming to a real world DSL called ThingML.

6 Related Work

Several approaches rely on execution trace comparison to better understand the *semantic differences* between executable models [19–23]. Among the approaches closest to our work, the work done by Langer et al. [22] relies on dedicated *matching rules* to align pairs of traces in order to compute semantic differences, where a set of matching rules define how traces should be meaningfully compared in the context of a given executable DSL. In contrast, our generic approach does not require any matching rules as input, and instead relies on simplifying first the traces using *Filter* and *Reduce* in order to abstract away details that would prevent from aligning equivalent states.

Alimadadi et al. [24] propose a high-level abstraction operator that detects recurring patterns and hierarchies of patterns in sequences of events. Their algorithm is inspired from sequence alignment algorithms used in bioinformatics, similarly to our *Compare* operator. Overall, while the motivation for their work is the same as ours, our approach relies on a set of more low-level operators that manipulate sequences of both states and events. In other words, our operators could be used as basic building blocks for providing higher-level operators.

Process mining is a process-centric management technique bridging the gap between data mining and traditional Business Process Management (BPM) [25, 26]. The main objective of process mining is to extract process-related information from event logs for providing information about actual processes [26]. Events are defined as process steps and event logs as sequential events recorded by an information system [27]. In [25], discovery is mentioned as one of the main goals of process mining, *i.e.,* taking an event log as input and to produce a process model as output. Event log comparisons techniques are also discussed in

the realm of process mining [28]. Compared to our presented approach, process mining starts with logs produced by information systems and not directly by the interpretation of the process models. Furthermore, current process mining techniques are only applicable on business process modeling languages such as BPMN and their formal representation such as Petri nets. Our techniques are general enough to be applicable for executable modeling languages in general. Furthermore, we consider events and data while the latter is mostly neglected by process mining approaches.

7 Conclusion and Perspectives

Traces obtained from the execution of behavioral models are essential both as sources of feedback and to perform trade-off analyses. Yet, it remains difficult for a modeler to understand how a design change impacts the obtained execution traces. To address this problem, we proposed in this paper a set of formally defined *trace comprehension operators* which can be used for dynamic information filtering, trace comparison with *diff* computation and visualization, and graph-based views extraction to analyze cycles. We implemented our approach as part of the GEMOC Studio, an Eclipse-based language and modeling workbench, and we validated the approach using model variants conforming to a State Machine DSL. We showed that our operators can be used to better understand the impact of small but significant changes made to the considered model.

The direct perspectives of this work include extending the trace comparison operator to consider events along model states (*e.g.,* to compare different operational semantics with different observable events), improving the execution trace metamodel to support execution traces of concurrent behaviors (*i.e.,* a partial ordering of events), extending the state graph operator to consider multiple traces as input, providing an additional operator to compare state graphs, and specifying rigorous guidelines explaining how and when to use which operator.

Acknowledgments. This work was supported by: the Austrian Science Fund (FWF) P 28519-N31, the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development and the Safran/Inria/CNRS collaboration GLOSE.

References

1. Object Management Group: Semantics of a Foundational Subset for Executable UML Models, V 1.1, August 2013
2. Bendraou, R., Combemale, B., Crégut, X., Gervais, M.P.: Definition of an executable SPEM 2.0. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007), pp. 390–397. IEEE (2007)
3. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: a new graph rewrite language based on the unified modeling language and java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-46464-8_21

4. Harel, D., Lachover, H., Naamad, A., Pnuelli, A., Politi, M., Sherman, R., Shtull-trauring, A., Trakhtenbrot, M.: STATEMATE: a working environment for the development of complex reactive systems. IEEE Trans. Softw. Eng. **16**(4), 403–414 (1990)
5. OASIS: Web Services Business Process Execution Language Version 2.0 (2007)
6. Combemale, B., Crégut, X., Pantel, M.: A design pattern to build executable DSMLs and associated V&V tools. In: Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC 2012), pp. 282–287 (2012)
7. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: executable DSMLs based on fUML. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 56–75. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_4
8. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40011-7_23
9. Bandener, N., Soltenborn, C., Engels, G.: Extending DMM behavior specifications for visual execution and debugging. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 357–376. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_24
10. Hegedüs, Á., Bergmann, G., Ráth, I., Varró, D.: Back-annotation of simulation traces with change-driven model transformations. In: Proceedings of the 8th International Conference on Software Engineering and Formal Methods (SEFM 2010), pp. 145–155. IEEE (2010)
11. Soden, M., Eichler, H.: Towards a model execution framework for Eclipse. In: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture (BD-MDA 2009). ACM (2009)
12. Tatibouët, J., Cuccuru, A., Gérard, S., Terrier, F.: Formalizing execution semantics of UML profiles with fUML models. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) MODELS 2014. LNCS, vol. 8767, pp. 133–148. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11653-2_9
13. Bousse, E., Mayerhofer, T., Combemale, B., Baudry, B.: Advanced and efficient execution trace management for executable domain-specific modeling languages. Softw. Syst. Model., 1–37 (2017)
14. Object Management Group: Meta Object Facility (MOF) Core Specification, V 2.5, June 2016. http://www.omg.org/spec/MOF/2.5
15. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Eclipse Series, 2nd edn. Addison-Wesley Professional, Boston (2008)
16. Bousse, E., Corley, J., Combemale, B., Gray, J., Baudry, B.: Supporting efficient and advanced omniscient debugging for xDSMLs. In: Proceedings of the International Conference on Software Language Engineering (SLE 2015). ACM (2015)
17. Groote, J.F., Vaandrager, F.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: Paterson, M.S. (ed.) ICALP 1990. LNCS, vol. 443, pp. 626–638. Springer, Heidelberg (1990). https://doi.org/10.1007/BFb0032063
18. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Sov. Phys. Dokl. **10**, 707–710 (1966)
19. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 163–172 (2011)

20. Maoz, S., Ringert, J.O., Rumpe, B.: Cddiff: Semantic differencing for class diagrams. In: Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP 2011), pp. 230–254 (2011)
21. Maoz, S., Ringert, J.O., Rumpe, B.: ADDIFF: semantic differencing for activity diagrams. In: Proceedings of the SIGSOFT/FSE 2011 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC 2011: 13th European Software Engineering Conference (ESEC-13), pp. 179–189 (2011)
22. Langer, P., Mayerhofer, T., Kappel, G.: Semantic model differencing utilizing behavioral semantics specifications. In: Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2014), pp. 116–132 (2014)
23. Addazi, L., Cicchetti, A., Rocco, J.D., Ruscio, D.D., Iovino, L., Pierantonio, A.: Semantic-based model matching with emfcompare. In: Proceedings of the 10th Workshop on Models and Evolution (ME 2016), pp. 40–49 (2016)
24. Alimadadi, S., Mesbah, A., Pattabiraman, K.: Inferring hierarchical motifs from execution traces. (2018)
25. van der Aalst, W.M.P.: Process mining: making knowledge discovery process centric. SIGKDD Explor. **13**(2), 45–49 (2011)
26. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes, 1st edn. Springer Publishing Company, Heidelberg (2011). Incorporated
27. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Process-Aware Information Systems: Bridging People and Software Through Process Technology. Wiley, Hoboken (2005)
28. Bose, R.P.J.C., van der Aalst, W.M.P.: Process diagnostics using trace alignment: opportunities, issues, and challenges. Inf. Syst. **37**(2), 117–141 (2012)

Author Index

Ahmadi, Reza 147
Alalfi, Manar H. 279
Anjorin, Anthony 97
Asoudeh, Nesa 199

Baduel, Ronan 132
Barmpis, Konstantinos 251
Barquero, Gala 46
Bertoa, Manuel F. 46
Biró, Miklós 264
Bousse, Erwan 293
Bruel, Jean-Michel 132
Burger, Erik 63
Burgueño, Loli 46

Chami, Mohammad 132
Chen, Jian 279
Combemale, Benoit 293
Cuadrado, Jesús Sánchez 28

Dean, Thomas R. 279
Dingel, Juergen 147
Diskin, Zinovy 80
Dupont, Guillaume 164

Egyed, Alexander 264

Gerasimou, Simos 12
Grieger, Marvin 97

Härtel, Johannes 216
Heinz, Marcel 216
Hili, Nicolas 147
Hingorani, Justin 251
Hinkel, Georg 63
Hoyos Rodriguez, Horacio 12

Iovino, Ludovico 80

Jürjens, Jan 179

Khendek, Ferhat 164
Kolovos, Dimitrios S. 12, 115
Kolovos, Dimitrios 251
Kolovos, Dimitris 235
König, Harald 80
Kossak, Felix 264

Labiche, Yvan 199
Lämmel, Ralf 216
Leblebici, Erhan 97
Leroy, Dorian 293

Madani, Sina 115
Mashkoor, Atif 264
Megna, Anaël 293
Moreno, Nathalie 46
Mustafiz, Sadaf 164

Ober, Iulian 132

Paige, Richard F. 12, 115
Polack, Fiona 235

Ramadan, Qusai 179
Ramesh, S. 279
Riediger, Volker 179
Rodriguez, Horacio Hoyos 235
Rutle, Adrian 80

Salnitri, Mattia 179
Stevens, Perdita 1
Strüber, Daniel 179

Toeroe, Maria 164
Troya, Javier 46

Vallecillo, Antonio 46

Wei, Ran 12
Wimmer, Manuel 293

Yigitbas, Enes 97
Yohannis, Alfa 235

Zolotas, Athanasios 12