

**Martina Seidl
Steffen Zschaler (Eds.)**

LNCS 10748

Software Technologies: Applications and Foundations

**STAF 2017 Collocated Workshops
Marburg, Germany, July 17–21, 2017
Revised Selected Papers**



Springer

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, 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>


Martina Seidl · Steffen Zschaler (Eds.)

Software Technologies: Applications and Foundations

STAF 2017 Collocated Workshops
Marburg, Germany, July 17–21, 2017
Revised Selected Papers

Editors

Martina Seidl
Johannes Kepler University of Linz
Linz
Austria

Steffen Zschaler 
Department of Informatics
King's College London
London
UK

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-74729-3 ISBN 978-3-319-74730-9 (eBook)
<https://doi.org/10.1007/978-3-319-74730-9>

Library of Congress Control Number: 2018930743

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer International Publishing AG 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 Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This volume contains revised selected technical papers presented at the six satellite events collocated with Software Technologies: Applications and Foundations (STAF 2017), a federation of leading conferences on software technologies. The events took place in Marburg, Germany, during July 17–21, 2017.

STAF 2017 brought together researchers and practitioners from both academia and industry with an interest in all aspects of software technology. The satellite events added to this by providing a collaborative environment in which to discuss emerging areas in software engineering and, in particular, model-driven engineering (MDE). This year STAF included a projects showcase, enabling research project teams to present a coherent view of their results to the community. Also, for the first time, there was a workshop on grand challenges in MDE research, which provided some interesting insights into research directions of future relevance.

The events whose papers are included in this volume are:

- BigMDE 2017: 5th International Workshop on Scalable Model-Driven Engineering
- GCM 2017: 8th International Workshop on Graph Computation Models
- GRAND 2017: First International Workshop on Grand Challenges in Modeling
- MORSE 2017: 4th International Workshop on Model-Driven Robot Software Engineering
- OCL 2017: 17th International Workshop in OCL and Textual Modeling
- STAF Projects Showcase 2017: Third event dedicated to international and national project dissemination and cooperation

Additionally, a doctoral symposium was organized as part of STAF. The corresponding proceedings have been published separately as CEUR Volume 1955.

Brief messages from the events listed above follow this preface. We are grateful to EasyChair for the support with the paper submission and reviewing process for all workshops and with the preparation of this volume. For each of the workshops at STAF 2017, we thank the organizers for the interesting topics and resulting talks. We also thank the paper contributors to these workshops and those who attended them. We would like to extend our thanks to the members of each workshop's Program Committee. Finally, we would like to thank the organizers of STAF 2017 and, in particular, the general chair, Gabriele Taentzer.

December 2017

Martina Seidl
Steffen Zschaler

STAF 2017 Organizer's Message

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 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 2017 took place in Marburg, Germany, during July 17–21, 2017, and hosted the four conferences ECMFA 2017, ICGT 2017, ICMT 2017, and TAP 2017, the transformation tool contest TTC 2017, six workshops, a doctoral symposium, and a projects showcase event. STAF 2017 featured four internationally renowned keynote speakers, and welcomed participants from around the world.

The STAF 2017 Organizing Committee thanks (a) all the 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) Philipps-Universität, the city of Marburg, and all sponsors for their support. A special thank you goes to Christoph Bockisch (local chair), Barbara Dinklage and further members of the Department of Mathematics and Computer Science of Philipps-Universität, for coping with all the foreseen and unforeseen work to prepare a memorable event.

December 2017

Gabriele Taentzer

BigMDE 2017 Organizers' Message

As model-driven engineering (MDE) is increasingly applied to larger and more complex systems, the current generation of modeling and model management technologies are being pushed to their limits in terms of capacity and efficiency. As such, additional research and development is imperative in order to enable MDE to remain relevant to industrial practice and to continue delivering its widely recognized productivity, quality, and maintainability benefits.

The 5th BigMDE Workshop (<http://www.big-mde.eu/>) was co-located with the Software Technologies: Applications and Foundations (STAF 2017) conference. BigMDE 2017 provided a forum for developers and users of modeling and model management languages and tools to present and discuss problems and solutions related to scalability aspects of MDE, including:

- Working with large models
- Collaborative modeling (version control, collaborative editing)
- Transformation and validation of large models
- Model fragmentation and modularity mechanisms
- Efficient model persistence and retrieval
- Models and model transformations on the cloud
- Visualization techniques for large models
- High-performance MDE
- Identification of scalability and performance issues in MDE

Contributions from the community were essential for the success of BigMDE 2017. In particular, we would like to acknowledge the hard work of all Program Committee members for the timely delivery of reviews given the tight review schedule, and to thank the authors for submitting and presenting their work at the workshop.

July 2017

Dimitris Kolovos
Davide Di Ruscio
Nicholas Matragkas
Jesús Sánchez Cuadrado
István Ráth
Massimo Tisi

BigMDE 2017 Program Committee

Konstantinos Barpis	University of York, UK
Marko Boger	University of Konstanz, Germany
Goetz Botterweck	LERO, Ireland
Marco Brambilla	Politecnico di Milano, Italy
Loli Burgueño	Universidad de Malaga, Spain
Rubby Casallas	Universidad de los Andes, Colombia
Tony Clark	University of Middlesex, UK
Marcos Didonet Del Fabro	Universidade Federal do Parana, Brazil
Antonio García-Domínguez	Aston University, UK
Esther Guerra	Universidad Autonoma de Madrid, Spain
Jesus J. García Molina	Universidad de Murcia, Spain
Alfonso Pierantonio	University of L'Aquila, Italy
Markus Scheidgen	Humboldt-Universität zu Berlin, Germany
Seyyed Shah	University of Oxford, UK
Harald Störrle	Technical University of Denmark, Denmark
Daniel Strüber	Philipps-Universität Marburg, Germany
Gerson Sunyé	University of Nantes, France
Dániel Varró	Budapest University of Technology and Economics, Hungary

GCM 2017 Organizers' Message

The 8th International Workshop on Graph Computation Models (GCM 2017) was held in Marburg, Germany, on July 17, 2017.

Graphs are common mathematical structures which are visual and intuitive. They constitute a natural and seamless way for system modeling in science, engineering and beyond, including computer science, life sciences, business processes, etc. Graph computation models constitute a class of very high level models where graphs are first-class citizens. They generalize classic computation models based on strings or trees, such as Chomsky grammars or term rewrite systems. Their mathematical foundation, in addition to their visual nature, facilitates specification, validation, and analysis of complex systems. A variety of computation models have been developed using graphs and rule-based graph transformation. These models include features of programming languages and systems, paradigms for software development, concurrent calculi, local computations and distributed algorithms, and biological and chemical computations.

The aim of GCM 2017 is to bring together researchers interested in all aspects of computation models based on graphs and graph transformation techniques. The workshop promotes the cross-fertilizing exchange of ideas and experiences among researchers and students from the different communities interested in the foundations, applications, and implementations of graph computation models and related areas. Previous editions of the GCM series were held in Natal, Brazil (GCM 2006), in Leicester, UK (GCM 2008), in Enschede, The Netherlands (GCM 2010), in Bremen, Germany (GCM 2012), in York, UK (GCM 2014), in L'Aquila, Italy (GCM 2015), and in Vienna, Austria (GCM 2016).

After a thorough review process, the Program Committee accepted five papers for publication in the proceedings and three additional papers for presentation and inclusion in the electronic pre-conference proceedings.

Several people contributed to the success of GCM 2017. I would like to thank the organizers of STAF 2017, and in particular the general chair, Gabriele Taentzer, and the workshop chairs, Martina Seidl and Steffen Zschaler. I would also like to express my thanks to the Program Committee and to the additional reviewers (Fabio Gadducci, Christian Sandmann, Timothy Atkinson, Berthold Hoffmann, Dennis Nolte, and Christoph Peuser) for their valuable help. The EasyChair system greatly facilitated the submission and program selection process.

I would furthermore like to thank all authors, speakers, and participants of the workshop.

GCM 2017 Program Committee

Andrea Corradini (Chair)	University of Pisa, Italia
Rachid Echahed	Laboratoire d'Informatique de Grenoble, France
Stefan Gruner	University of Pretoria, South Africa
Annegret Habel	Universität Oldenburg, Germany
Dirk Janssens	Universiteit Antwerpen, Belgium
Barbara König	Universität Duisburg-Essen, Germany
Hans-Jörg Kreowski	Universität Bremen, Germany
Mohamed Mosbah	LaBRI, Université de Bordeaux, France
Detlef Plump	University of York, UK
Leila Ribeiro	Universidade Federal do Rio Grande do Sul, Brazil

GRAND 2017 Organizers' Message

The fields of modeling and model-driven engineering have made incredible contributions to leverage abstraction and automation in almost every area of software and systems development and analysis. In many domains, including automotive software engineering, embedded systems, and business process engineering, models are key to success in modern software engineering processes. However, this success has led to an even higher demand for better processes, tools, theories, and general awareness about modeling, its scope and application. Important questions have emerged about how our field can and should respond to this changing landscape in terms of identifying the main challenges in modeling and model-driven engineering. This includes not only the challenges for today, but for the next 3, 5, and 10 years; these challenges need to be cooperatively and collaboratively defined, to help produce a challenging research agenda for the field. The goal of the GrandMDE Workshop was to come up with such a list of *grand challenges* for model-driven engineering.

The workshop received 15 paper submissions from which 11 were selected for presentation at the workshop. Owing to the nature of the workshop, all papers were position papers, each one presenting a different challenge. Topics covered in the workshop included: adoption of MDE, temporal modeling, executable models, artificial intelligence for MDE, enterprise modeling and Industry 4.0, quality aspects, etc. Each paper was reviewed by at least three Program Committee members.

The organizers would like to thank the authors and presenters of submitted papers, the Program Committee members, and the audience for the contribution to the success of the workshop.

November 2017

Jordi Cabot
Richard Paige
Alfonso Pierantonio

GRAND 2017 Program Committee

Bernhard Rumpe

Manuel Wimmer

Esther Guerra

Sahar Kokaly

Rick Salay

Marco Brambilla

Dimitris Kolovos

Antonio Vallecillo

RWTH Aachen University, Germany

Vienna University of Technology, Austria

Universidad Autónoma de Madrid, Spain

McMaster University, Canada

University of Toronto, Canada

Politecnico di Milano, Italy

University of York, UK

University of Malaga, Spain

MORSE 2017 Organizers' Message

The 4th edition of the international Workshop on Model-Driven Robot Software Engineering (MORSE) was held at the International Conference on Software Technologies: Applications and Foundations (STAF). This year's edition took place in Marburg, Germany, on July 21, 2017 and focused on scenario-based development and interaction modeling. Previous editions were located at STAF 2014 (York, UK), STAF 2015 (L'Aquila, Italy), and RoboCup 2016 (Leipzig, Germany).

With the rise of standardized robotic hardware platforms and software platforms and the pace with which software ecosystems and app stores develop in application markets, the following research topics become increasingly important in software engineering for robotics: (1) model-driven development of robotic systems; (2) software and app reuse for robotics; (3) end-user robot app development; (4) compliance to legal and safety constraints; and (5) total cost of ownership.

Model-driven development facilitates designing and engineering complex systems through automatization and concentrating on different levels of abstraction. Advances in robotics research and the increasing complexity of robotic systems demand for both. Model-driven development can help to improve the quality (e.g., re-usability, reliability, maintainability) of the robotic systems. Hence, there is a need for a new paradigm of model-driven software and system engineering for robots. Besides six interesting paper presentations, the workshop program of MORSE 2017 was enriched by a keynote talk given by Davide di Ruscio, University of L'Aquila: "The Role of Models in Engineering the Software of Robotic Systems."

The need and timeliness of this topic, as well as the excellent support of its Program Committee, resulted in having a successful fourth edition of the MORSE workshop.

November 2017

Sebastian Götz
Christian Piechnick
Andreas Wortmann

MORSE 2017 Program Committee

Colin Atkinson	University of Mannheim, Germany
Herman Bruyninckx	KU Leuven, Belgium
Kerstin Eder	University of Bristol, UK
Frank J. Furrer	TU Dresden, Germany
Kurt Geihs	Universität Kassel, Germany
Diana Goehring	Ruhr-University Bochum, Germany
Nico Hochgeschwender	Bonn-Rhine-Sieg University of Applied Sciences, Germany
Bernhard Jung	TU Freiberg, Germany
Alexander Jungmann	IAV GmbH, Germany
Jens Knoop	TU Vienna, Austria
Lorenzo Natale	Instituto Italiano di Tecnologia, Italy
Arne Nordmann	Robert Bosch GmbH, Germany
Ralf Reussner	Karlsruhe Institute of Technology, Germany
Bernhard Rumpe	RWTH Aachen University, Germany
Davide Di Ruscio	University of L'Aquila, Italy
Ulrik Schultz	University of Southern Denmark, Denmark
Serge Stinckwich	IRD, France
Cristina Vicente-Chicote	Universidad de Extremadura, Spain
Heike Wehrheim	University of Paderborn, Germany
Sebastian Wrede	CoR-Lab, Bielefeld University, Germany

OCL 2017 Organizers' Message

The goal of the OCL 2017 workshop was to create a forum where researchers and practitioners interested in building models using OCL or other kinds of textual languages could directly interact, report advances, share results, identify tools for language development, and discuss appropriate standards. In particular, the workshop encouraged discussions for achieving synergy from different modeling language concepts and modeling language use. The close interaction enabled researchers and practitioners to identify common interests and options for potential cooperation.

The workshop received seven submissions from which five were selected as full papers. Each paper was reviewed by at least three Program Committee members. The workshop hosted an open session with Lightning Talks (5 min.) at the end of the day where speakers were given the opportunity to talk about whatever they wanted, as long as it was related to the topics of the workshop. Three presentations were given.

The organizers would like to thank the authors of submitted papers, the Program Committee members, the workshop speakers, and the workshop audience for the contribution to the success of the workshop.

November 2017

Robert Bill
Achim D. Brucker
Jordi Cabot
Martin Gogolla

OCL 2017 Program Committee

Thomas Baar	HTW Berlin, Germany
Mira Balaban	Ben-Gurion University of the Negev, Israel
Tricia Balfé	Nomos Software, Ireland
Domenico Bianculli	University of Luxembourg, Luxembourg
Dan Chiorean	Babes-Bolyai University, Romania
Robert Clariso	Universitat Oberta de Catalunya, Spain
Tony Clark	Sheffield Hallam University, UK
Manuel Clavel	Universidad Complutense de Madrid, Spain
Birgit Demuth	TU Dresden, Germany
Geri Georg	Colorado State University, USA
Istvan Rath	Budapest University of Technology, Hungary
Jan Oliver Ringert	Tel Aviv University, Israel
Bernhard Rumpe	RWTH Aachen University, Germany
Adolfo Sanchez-Barbudo	University of York, UK
Massimo Tisi	Inria, France
Frederic Tuong	University Paris-Sud, France
Edward Willink	Willink Transformations Ltd., UK
Burkhart Wolff	University of Paris-Sud, France
Steffen Zschaler	King's College London, UK

Subreviewer

Olga Haubrich	RWTH Aachen University, Germany
---------------	---------------------------------

Projects Showcase 2017 Organizers' Message

The aim of the Projects Showcase event at STAF 2017 was to provide an opportunity for researchers and practitioners involved in ongoing or completed research projects related to foundations and applications of software technologies to share results, experiences, ideas, on-going work, and knowledge that could lead to fruitful inter-project collaboration.

The call for papers of the event solicited contributions disseminating the objectives and results of national and international research projects, including outcomes of specific deliverables, advances beyond the state of the art, overall innovation potential, exploitation approach and (expected) impact, marketing value, barriers, and obstacles.

Six papers were accepted for presentation and publication in the proceedings of the event, which reported on different types of national, international, and in-house research and development projects. We would like to acknowledge the hard work of the members of the Program Committee and the contribution of the authors of submitted papers to the success of this event.

November 2017

Massimo Tisi
Thanos Zolotas

Projects Showcase 2017 Program Committee

Alessandra Bagnato	Softeam, France
Antonio García-Domínguez	Aston University, UK
Tom Ritter	Fraunhofer, Germany
Elisabetta Di Nitto	Politecnico di Milano, Italy
Antonio Cicchetti	Malardalen University, Sweden
Nicos Malevris	Athens University of Economics and Business, Greece
Goetz Botterweck	Lero, Ireland
István Ráth	Budapest University of Technology and Economics/IncQuery Labs, Hungary
Nicholas Matragkas	University of Hull, UK

Contents

Scalable Model Driven Engineering (BigMDE)

Injecting Execution Traces into a Model-Driven Framework for Program Analysis.	3
<i>Thibault Bézières la Fosse, Massimo Tisi, and Jean-Marie Mottu</i>	
Introduction of an OpenCL-Based Model Transformation Engine	14
<i>Tamás Fekete and Gergely Mezei</i>	
Collaborative Modelling with Version Control	20
<i>Steven Kelly</i>	
A Java Bytecode Metamodel for Composable Program Analyses	30
<i>Bugra M. Yildiz, Christoph Bockisch, Arend Rensink, and Mehmet Aksit</i>	

Graph Computation Models (GCM)

Graph Rewriting Based Search for Molecular Structures: Definitions, Algorithms, Hardness	43
<i>Ernst Althaus, Andreas Hildebrandt, and Domenico Mosca</i>	
Towards Automatic Generation of Evolution Rules for Model-Driven Optimisation	60
<i>Alexandru Burdusel and Steffen Zschaler</i>	
Generating Efficient Predictive Shift-Reduce Parsers for Hyperedge Replacement Grammars	76
<i>Berthold Hoffmann and Mark Minas</i>	
Checking Graph Programs for Confluence	92
<i>Ivaylo Hristakiev and Detlef Plump</i>	
Loose Graph Simulations	109
<i>Alessio Mansutti, Marino Miculan, and Marco Peressotti</i>	

Grand Challenges in Modeling (GRAND)

Models, More Models, and Then a Lot More	129
<i>Önder Babur, Loek Cleophas, Mark van den Brand, Bedir Tekinerdogan, and Mehmet Aksit</i>	

On the Need for Temporal Model Repositories	136
<i>Robert Bill, Alexandra Mazak, Manuel Wimmer, and Birgit Vogel-Heuser</i>	
On the Need for Artifact Models in Model-Driven Systems Engineering Projects	146
<i>Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann</i>	
Cognifying Model-Driven Software Engineering	154
<i>Jordi Cabot, Robert Clarisó, Marco Brambilla, and Sébastien Gérard</i>	
Non-human Modelers: Challenges and Roadmap for Reusable Self-explanation	161
<i>Antonio Garcia-Dominguez and Nelly Bencomo</i>	
Some Narrow and Broad Challenges in MDD.	172
<i>Martin Gogolla, Frank Hilken, and Andreas Kästner</i>	
Modelling by the People, for the People	178
<i>Steven Kelly</i>	
From Building Systems Right to Building Right Systems: A Generic Architecture and Its Model Based Realization	184
<i>Vinay Kulkarni and Sreedhar Reddy</i>	
The Tool Generation Challenge for Executable Domain-Specific Modeling Languages	193
<i>Tanja Mayerhofer and Benoit Combemale</i>	
Toward Product Lines of Mathematical Models for Software Model Management.	200
<i>Zinovy Diskin, Harald König, Mark Lawford, and Tom Maibaum</i>	
Model-Driven Robot Software Engineering (MORSE)	
Model-Driven Interaction Design for Social Robots.	219
<i>Gary Cornelius, Nico Hochgeschwender, and Holger Voos</i>	
Towards Integration of Context-Based and Scenario-Based Development	225
<i>Achiya Elyasaf, David Harel, Assaf Marron, and Gera Weiss</i>	
(An Example for) Formally Modeling Robot Behavior with UML and OCL	232
<i>Martin Gogolla and Antonio Vallecillo</i>	

Synthesizing Executable PLC Code for Robots from
 Scenario-Based GR(1) Specifications 247
Daniel Gritzner and Joel Greenyer

Evaluating a Graph Query Language for Human-Robot Interaction
 Data in Smart Environments 263
Norman Köster, Sebastian Wrede, and Philipp Cimiano

A Simulation Framework to Analyze Knowledge Exchange Strategies
 in Distributed Self-adaptive Systems 280
Christopher Werner, Sebastian Götz, and Uwe Aßmann

OCL and Textual Modeling (OCL)

Workshop in OCL and Textual Modelling: Report on Recent Trends
 and Panel Discussions 297
*Robert Bill, Achim D. Brucker, Jordi Cabot, Martin Gogolla,
 Antonio Vallecillo, and Edward D. Willink*

Improving Incremental and Bidirectional Evaluation with an Explicit
 Propagation Graph 302
*Frédéric Jouault, Olivier Beaudoux, Matthias Brun, Fabien Chhel,
 and Mickaël Clavreul*

Translating UML-RSDS OCL to ANSI C 317
*Kevin Lano, Sobhan Yassipour-Tehrani, Hessa Alfraihi,
 and Shekoufeh Kolahdouz-Rahimi*

Mapping USE Specifications into Spec# 331
Jagadeeswaran Thangaraj and SenthilKumaran Ulaganathan

Deterministic Lazy Mutable OCL Collections 340
Edward D. Willink

Step 0: An Idea for Automatic OCL Benchmark Generation. 356
Hao Wu

Projects Showcase

SICOMORo-CM: Development of Trustworthy Systems via Models
 and Advanced Tools 367
*Elvira Albert, Pablo C. Cañizares, Esther Guerra, Juan de Lara,
 Esperanza Marcos, Manuel Núñez, Guillermo Román-Díez,
 Juan Manuel Vara, and Damiano Zanardini*

Developer-Centric Knowledge Mining from Large Open-Source Software Repositories (CROSSMINER)	375
<i>Alessandra Bagnato, Konstantinos Barmpis, Nik Bessis, Luis Adrián Cabrera-Diego, Juri Di Rocco, Davide Di Ruscio, Tamás Gergely, Scott Hansen, Dimitris Kolovos, Philippe Krief, Ioannis Korkontzelos, Stéphane Laurière, Jose Manrique Lopez de la Fuente, Pedro Maló, Richard F. Paige, Diomidis Spinellis, Cedric Thomas, and Jurgen Vinju</i>	
Technical Obsolescence Management Strategies for Safety-Related Software for Airborne Systems	385
<i>Simos Gerasimou, Dimitris Kolovos, Richard Paige, and Michael Standish</i>	
Mobile Health ID Card: Demonstrating the Realization of an mHealth Application in Austria	394
<i>Malgorzata Zofia Goraczek, Michael Sachs, Oliver Terbu, Lei Zhu, Birgit Scholz, Georg Egger-Sidlo, Sebastian Zehetbauer, and Stefan Vogl</i>	
SECT-AIR: Software Engineering Costs and Timescales – Aerospace Initiative for Reduction	403
<i>Richard F. Paige, Athanasios Zolotas, Dimitrios S. Kolovos, John A. McDermid, Mike Bennett, Stuart Hutchesson, and Andrew Hawthorn</i>	
DEIS: Dependability Engineering Innovation for Cyber-Physical Systems . . .	409
<i>Ran Wei, Tim P. Kelly, Richard Hawkins, and Eric Armengaud</i>	
Author Index	417

Scalable Model Driven Engineering (BigMDE)

Injecting Execution Traces into a Model-Driven Framework for Program Analysis

Thibault Béziers la Fosse^(✉), Massimo Tisi, and Jean-Marie Mottu

AtlanMod Team (Inria, IMT Atlantique, LS2N), Nantes, France
{thibault.beziers-la-fosse,massimo.tisi,jean-marie.mottu}@inria.fr

Abstract. Model-Driven Engineering (MDE) has been successfully used in static program analysis. Frameworks like MoDisco inject the program structure into a model, available for further processing by query and transformation tools, e.g., for program understanding, reverse-engineering, modernization. In this paper we present our first steps towards extending MoDisco with capabilities for dynamic program analysis.

We build an injector for program execution traces, one of the basic blocks of dynamic analysis. Our injector automatically instruments the code, executes it and captures a model of the execution behavior of the program, coupled with the model of the program structure. We use the trace injection mechanism for model-driven impact analysis on test sets. We identify some scalability issues that remain to be solved, providing a case study for future efforts in improving performance of model-management tools.

Keywords: Model-Driven Engineering · Dynamic program analysis
Execution traces

1 Introduction

Many properties of a program have to be analyzed during its lifecycle, such as correctness, robustness or safety. Those behaviors can be analyzed either dynamically by executing the program, or statically, usually by examining the source code. Several program analysis frameworks exist and are heavily used by the engineering community. Code coverage frameworks are a very popular example for dynamic analysis since they can significantly improve testing quality. Dynamic information can be used to add more precision to static analysis [5, 6].

The MoDisco framework [4] is designed to enable program analysis in Model-Driven Engineering (MDE) by creating a model of the source code, and using it for program understanding and modernization. The code model makes the program structure easily accessible to external modeling tools for any kind of processing, e.g. for static analysis. This uniform treatment of models and programs in the MDE technical space has proven to greatly help the understanding, development and maintenance of complex software [3].

MDE tools analyzing source code through MoDisco do not usually execute the original program. However, if MoDisco provided models of dynamic aspects of the code, this would enable other useful analysis. For instance, a model of test traces for a *software under test* (SUT) and its test suite could foster a better understanding of the test suite, and of the impact that a statement modification could have on the test cases executing it. Moreover, it could be used to get an overview of the testing code coverage. Indeed, if the modification of a statement has no impact on any test method, the statement is considered as uncovered. Naturally this analysis model could provide a significant improvement to the quality of the test sets [11].

In this paper we show a method to build a model of the program execution, using both static and dynamic analysis. Thus, an initial structural model is statically built, containing the basic blocks of a program: packages, classes, methods and statements. Thereafter, the code is instrumented in order to add execution traces to the model during program execution. Consequently, each test method is associated to the ordered sequence of statements it executes.

The remainder of this paper is organized as follows. Section 2 presents our approach for model creation. Section 3 proposes an evaluation of the process on a set of projects having different sizes. Related work is discussed in Sect. 4, and Sect. 5 concludes this paper.

2 Approach

This section illustrates our approach for dynamically building a model of the program execution. We propose an automatic process made of a sequence of four steps. Figure 1 gives an overview of this process. On the left-hand side of Fig. 1, the input is the source code of the considered system (e.g. a program under test and its tests). First, a static model is generated thanks to a reverse engineering step (Sect. 2.1). Second, on the right-hand side of Fig. 1, the static model is refactored into an analysis model by a model transformation (Sect. 2.2). Third, on the left-hand side of Fig. 1, a source code instrumentation step (Sect. 2.3) prepares the code before execution. Finally, the instrumented code is executed and its instrumentation allows us to complete the analysis model into the dynamic model of the source code (Sect. 2.4). This model is the output of the process.

The dynamic model should contain the structure of the source code, especially describing the targeted system (e.g. the system under test). Furthermore it should reify which statements are executed when the system is run under the action of a launcher (e.g. a set of tests). In addition, the order of the calls should be stored to be used when analyzing the behavior of the system based on the dynamic model.

In this work, we consider Java source code and we exemplify the approach by the two classes in Fig. 2. The considered system is the class under test *Factorial* (left of Fig. 2) associated to a JUnit test class *TestFactorial* (right of Fig. 2). Those classes are the source code entry of the process in Fig. 1.

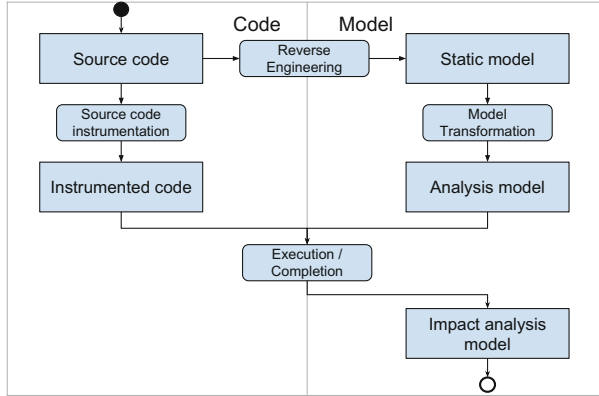


Fig. 1. 4-step process generating an test impact analysis model from source code

```

package main;

public class Factorial {

    public int fact(int n) {
        if (n == 1) {
            return 1;
        }
        return n * fact(n - 1);
    }
}

package main;

import static org.junit.Assert.*;
import org.junit.Test;

public class TestFactorial {
    @Test
    public void checkFact() {
        Factorial f= new Factorial();
        assertEquals(1, f.fact(1));
    }
}

```

Fig. 2. Factorial class, and its incomplete test suite.

2.1 Model Driven Reverse Engineering

The first step of our approach consists of generating the model of the code structure using MoDisco¹. MoDisco has a visitor-based system which navigates through the full abstract syntax tree (AST), and then builds a model from it, according to its Java meta-model [4]. We use a specific option of MoDisco, which annotates each element of the output model with its location in the source code. This information will be necessary for the dynamic analysis, as we show later.

Figure 3 shows a simplified version of the model generated by MoDisco from the code in Fig. 2 (*Static model* in Fig. 1). *Node* elements contain the position, and a reference to the statements. Since the full static model generated by MoDisco is rather large, we extract the excerpt in Fig. 3, focusing on statement-level information, in order to improve performance. For instance we filter out information about expressions, binary files, or import declarations. Since in this work we focus on the execution trace of the statements, only those ones are needed, within their respective classes, methods, and packages containers. Specifically this filtering is required to minimize the final model in-memory size

¹ <http://www.eclipse.org/MoDisco/>.

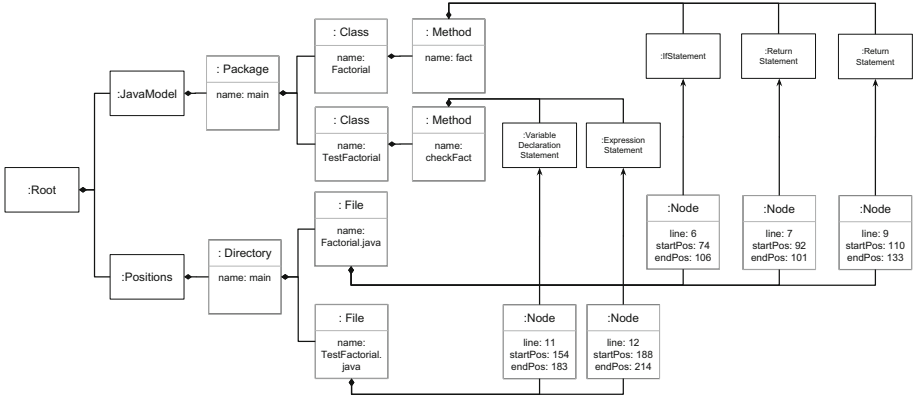


Fig. 3. Excerpt of the static model generated by MoDisco from the code in Fig. 2

afterwards. The filtering is performed during the model transformation described in the next section.

2.2 Model Transformation

The goal of this model transformation is to produce a simpler model to use, containing only the data needed for the program analysis. Moreover, this model must be able to associate methods to statements in order to highlight all the methods which execute a specific statement. This link is created during the execution, i.e. dynamically.

The input of this model transformation is the model statically generated by MoDisco, while the target is a model conforming to a meta-model we introduce. This meta-model needs to differentiate the test classes from the SUT classes (targets), since the execution trace is only computed for the System Under Test. This meta-model is illustrated in Fig. 4.

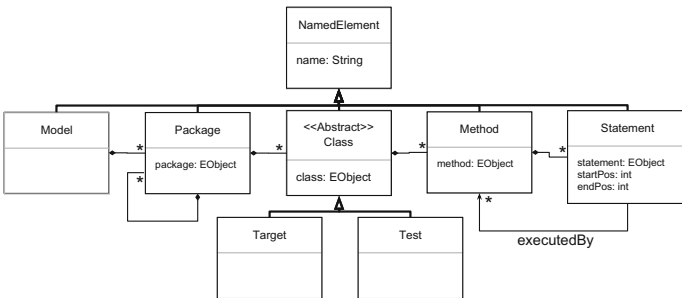


Fig. 4. The impact analysis meta-model.

The `model` element corresponds to the root of the model. It contains packages, which can contain either other packages or classes. Each class can have several methods, which contain a list of statements. Finally, each statement owns a list of references to methods, called `executedBy`. When the model is created by the transformation, this list is empty, nonetheless the completion step described in Fig. 1 consists into filling this list in order to get an execution trace. Since this trace only concerns the SUT classes, the statements of the test suites are not necessary in the analysis model. Thus, target and test classes are differentiated, and only the SUT's statements are kept.

To separate the target classes from the test classes, we assumed that test classes either use the `@Test` annotations, or inherit the `TestSuite` class, from the well known Java testing framework JUnit². That information can easily be accessed inside the static model. Though, several other conditions can be added to the transformation, in order to differentiate the tests from other classes, using the TestNG³ annotations for example.

The model transformation is made using ATL⁴, with the EMF Transformation Virtual Machine (EMFTVM). Indeed, besides including all the standard ATL VM features, EMFTVM offers better model loading and saving, smaller file results, and an overall performance improvement during the ATL transformation [12].

To focus a bit more about the transformation itself, 7 ATL rules are applying a top-down approach on the static model, starting by transforming the packages, classes, and going down to the statements. Apart from the `namedElement`, each element of the meta-model corresponds to an ATL rule.

This transformation drastically reduces the size of the model manipulated in the further sections. In fact this transformation is used to generate the analysis model out of the JUnit4 source code static model. The size of one model is lowered from 50 MB to 1 MB for instance, by keeping only the relevant objects for future processing.

Once this model is built, we need to run the tests in order to produce the dynamic analysis. Before, an instrumentation of the code is necessary, to be able to observe and reify its behavior into the dynamic model during the last step (Sect. 2.4).

2.3 Code Instrumentation

Code instrumentation is conducted by inserting additional statements in the code, at specific places, so that when the SUT is executed, those new statements are executed too [6].

Several instrumentation approaches exist:

- Source code instrumentation consists in adding code into the source file, before compiling it.

² <http://junit.org/junit4/>.

³ <http://testng.org/>.

⁴ <https://eclipse.org/atl/>.

- Binary instrumentation modifies or re-writes the compiled code.
- Bytecode instrumentation performs tracing within the compiled code.

During the development of our process, we considered those strategies, and found that the source code instrumentation is the most accurate approach for producing execution traces. Indeed, required information about the source cannot be obtained when analyzing the binaries or the bytecode, especially the source code position of the statements. Besides having the line number for each statement like the other approaches, source code instrumentation can also provide column position numbers. This data will be used to match the executed statements of the code with the analysis model's statements. It is mandatory for the next step (Sect. 2.4) to be able to add the behavior into the static model.

The instrumentation has been performed using the Spoon Framework [9]. This step is done first by iterating over the analysis model, in order to get both Test and Target classes. When instrumenting Test classes, a `setMethod()` statement is inserted at the beginning of each test method. Its parameters are the test class name, and the method name. When instrumenting SUT classes, a `match()` call is added before each statement. Its parameters are the SUT class qualified name where it belongs, the method, and finally the source code position of this statement. The source-code instrumentation generates the code of the Fig. 5, which is compiled and loaded before the execution.

```

package main;
public class Factorial {
    public int fact(int n) {
        match("Factorial", "fact", 74, 106);
        if (n == 1) {
            match("main.Factorial", "fact", 92,
                101);
            return 1;
        }
        match("main.Factorial", "fact", 110,
            133);
        return n * (fact((n - 1)));
    }
}

package main;
public class TestFactorial {
    @Test
    public void testFactorial() {
        setMethod("main.TestFactorial",
            "testFactorial");
        Factorial f = new Factorial();
        Assert.assertEquals(1, f.fact(1));
    }
}

```

Fig. 5. Instrumented code of Factorial class, and its incomplete test suite.

2.4 Execution and Completion

The execution part consists of running the test cases. When the new instrumented statements are executed, the model statically built is completed with dynamic information. Before the test execution, a singleton class is instantiated. The method `setMethod()` sets the current test method being executed into the singleton class.

The other method, `match()`, iterates over the analysis model to find the SUT statement being executed, using its qualified class name, method name, and finally the source code position of the statement. Finally, the `executedBy`

list of the newfound statement is completed dynamically, by adding the singleton’s current test method. This model dynamically completed reifies the link between statements and test methods, as showed in Fig. 6. Besides showing the test methods that would be impacted when modifying a SUT statement, an empty `executedBy` list shows an uncovered statement, which possibly implies an incomplete test suite. Thus, this model answers to the problematic announced, as it is dynamically created, and analyze the program’s behavior i.e. with impact analysis.

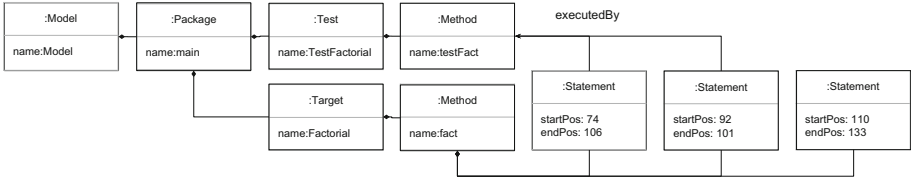


Fig. 6. The analysis model obtained after reifying the link between statements and test methods.

However this model highly depends on the source code’s size. Indeed, each statement in the SUT corresponds to one element in the analysis model, thereby an important source code might lead to scalability issues. This scalability is evaluated in the next section.

3 Evaluation

3.1 Execution Environment

In this section we evaluate the overall performances of our dynamic model generation framework, using the XMI persistence layer for our models.

This evaluation is conducted on a simple Java project containing the classes introduced in the Sect. 2. We programmatically increase the size of this project by duplicating the number of classes and test classes. This way, every test class is testing a single target class. Using this setup, we can manage the size of the output model, and observe the behavior of our prototype with either small and big models.

The experiments are executed on a laptop running Windows 7 Professional 64 Bits, using an Intel Core i7-4600 (2.70 GHz) CPU and, a Samsung SSD 840 EVO 250 GB. The Java Virtual Machine version is JDK 1.8.0_121, and runs with a maximum Java heap size of 2,048 MB.

This experimentation starts by running the program analysis on Java Projects containing a few hundred classes, which can be considered as small here. Subsequently, this number of classes is increased, up to thousands. We measured the execution time for each step of our prototype, and reported it in

the Fig. 7. As described in the previous parts, those steps are: Reverse Engineering (RE), Model Transformation (ATL), Source code Instrumentation (Instr), Test Execution (Exec).

When the project under analysis reaches approximately 12,000 test classes, the model created by MoDisco using reverse engineering is too big to be stored in memory, thus preventing any other analysis on bigger projects.

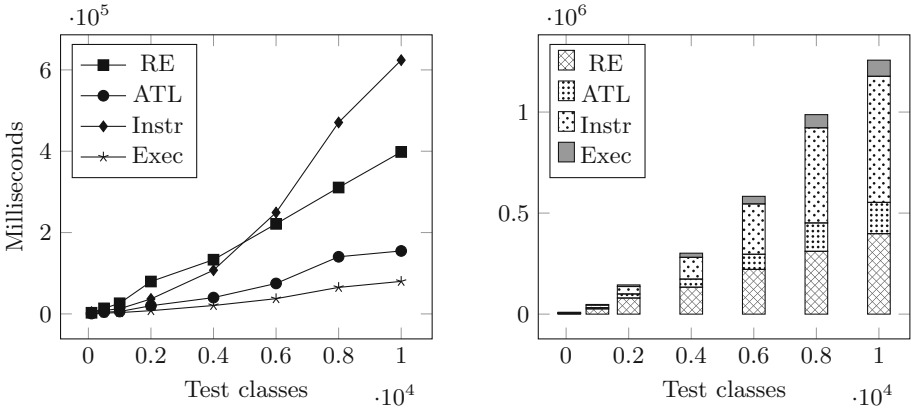


Fig. 7. Dynamic program analysis execution times using the XMI persistence layer.

3.2 Discussion

The curves from the left diagram in the Fig. 7 are showing the growth of execution times when the number of classes increases. The other diagram shows the same data, but its representation gives a better understanding of each step's duration in the whole process and its total duration. This diagram shows that the MoDisco static model generation, and the source code instrumentation are by far the longest steps of the program analysis, up to 32% for the reverse engineering step, and 50% for the instrumentation step, approximately.

This can be explained by the fact that both are parsing the whole source code. Yet it is interesting to notice that the instrumentation tends to be longer than the reverse engineering once the model contains more than 5,000 test classes. The reason is the Spoon's instrumentation's complexity being more than linear.

Also, as written in the previous subsection, the MoDisco static model creation will not be achieved when the program under analysis gets very big (approximately 12,000 test classes) due to a lack of memory and the well-known XMI scalability problem.

Pagán et al. explained in their paper [8] that the XMI persistence layer scales badly with large models, due to the fact that XMI files cannot be partially loaded. Indeed the XMI resource needs to keep the complete object in the memory to use it.

This scalability problem can be partially resolved using a more scalable persistence layer for the EMF Models, such as NeoEMF [2] or CDO⁵. Nonetheless, MoDisco has its own meta-models, and uses EMF generated code. Using this code with NeoEMF and CDO resources cannot currently improve the scalability, since those layers need to generate their own code from an Ecore meta-model. It is one of our future challenges.

4 Related Work

In order to reduce the maintenance costs of large applications, knowing the impacts a code modification can have on other parts of the program can really improve the developers life. Several approaches already exist within the impact analysis domain.

PathImpact [7] is a dynamic impact analysis technique that does not require any static analysis during its process. PathImpact instruments the binaries or a program, and generates traces during its execution. This execution trace is then used to create call graphs, which are analysed in order to study the impact analysis. The impacts are identified at the method level, a coarser level than our approach based on source code instrumentation.

Chianti [10] is a change impact analysis tool based on call graphs. Using two different versions of the source code, Chianti creates a set of *atomic changes*, and analyses their dependencies in order to determine the tests affected by code changes. Basically, considering two atomic changes A_1 and A_2 , if adding A_1 in the first version of the source code leads to a program syntactically invalid, then it has a dependency to A_2 . For each tests, a call graph is generated (statically or dynamically), and from those dependencies, operations affecting the tests can be identified. This approach is really different from ours since it needs two version of the source code. This impact analysis are more coarse-grained than the approach presented in this paper, i.e. classes, methods and fields, but Chianti supports more operations, such as insertion and deletion.

Imp [1] is a static impact analysis tool for the C++ language based on program slicing [13]. Program slicing consists into computing a set of points, such as statements, that can have an effect on other point of the program. To compute the impact analysis, Imp only considers “forward” slicing, which computes a set of statements that are affected by a previous point. This approach is fine-grained, since the analysis can be done at the statement level. Nonetheless it suffers from the disadvantages of the static approach, a loss of precision, in order to limit the execution time when the program’s source code is being too big to analyse.

5 Conclusion and Future Works

In this paper we present our prototype for dynamic program analysis purposes, using Model Driven Engineering. The multiple steps of the process dynamically

⁵ <https://eclipse.org/cdo/>.

generate a model, highlighting the links between program's statements and test suites. Nevertheless, the experimentations led on this prototype showed that it suffers from the XMI scalability issues induced by the MoDisco static analysis.

Our future work will focus on the scalability of our prototype, more specifically by integrating a more efficient persistence layer in term of scalability, such as CDO or NeoEMF. Furthermore, the source code instrumentation remains a very cumbersome technique, and evolving towards a more common solution, such as "on the fly" byte-code instrumentation would be interesting, especially if the granularity is maintained.

Acknowledgment. This work is supported by DIZOLO project - Aurora mobility programme 2017.

References

1. Acharya, M., Robinson, B.: Practical change impact analysis based on static program slicing for industrial software systems. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 746–755. ACM (2011)
2. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, a scalable persistence layer for EMF models. In: Cabot, J., Rubin, J. (eds.) ECMFA 2014. LNCS, vol. 8569, pp. 230–241. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09195-2_15
3. Bézivin, J.: On the unification power of models. *Softw. Syst. Model.* **4**(2), 171–188 (2005)
4. Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: a model driven reverse engineering framework. *Inf. Softw. Technol.* **56**(8), 1012–1032 (2014)
5. Ernst, M.D.: Static and dynamic analysis: synergy and duality. In: ICSE Workshop on Dynamic Analysis, WODA 2003, pp. 24–27 (2003)
6. Gosain, A., Sharma, G.: A survey of dynamic program analysis techniques and tools. In: Satapathy, S.C., Biswal, B.N., Udgata, S.K., Mandal, J.K. (eds.) Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014. AISC, vol. 327, pp. 113–122. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-11933-5_13
7. Law, J., Rothermel, G.: Whole program path-based dynamic impact analysis. In: 2003 Proceedings of the 25th International Conference on Software Engineering, pp. 308–318 (2003)
8. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: 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
9. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: SPOON: a library for implementing analyses and transformations of Java source code. *Softw. Pract. Exp.* **46**(9), 1155–1179 (2016)
10. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of Java programs, vol. 39, no. 10, pp. 432–448 (2004)
11. Sherwood, K.D., Murphy, G.C.: Reducing code navigation effort with differential code coverage. Department of Computer Science, University of British Columbia, Technical report (2008)

12. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 623–637. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_46
13. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449. IEEE Press (1981)

Introduction of an OpenCL-Based Model Transformation Engine

Tamás Fekete^(✉) and Gergely Mezei

Budapest University of Technology and Economics, Budapest, Hungary
{fekete,gmezei}@aut.bme.hu

Abstract. As model-driven engineering (MDE) became a popular software development methodology, several tools are built to support working with MDE. Nowadays, the importance of performance is getting higher as the size of the systems grow. New solutions are needed that can take advantage of modern hardware components and architectures. One step towards this goal is to use the unique processing power of GPUs in model-driven environments. Our overall goal is to create a graph transformation framework that fits into the parallel execution environment provided by GPUs. Our approach is based on the OpenCL framework and it is referred to as PaMMTE (Parallel Multiplatform Model-transformation Engine). This paper presents an overview of our tool and the description of the implementation. We believe that this new approach will be an attractive way to accelerate MDE tools efficiently.

Keywords: MDE · GPU · Graph transformation
High-performance computation · Parallel computation · OpenCL

1 Introduction

Model-driven engineering (MDE) can simplify the software development processes that caused the sudden spreading of its usage in various domains. MDE works with models that are no longer created only for presentation purposes but transformed, processed and often used directly or indirectly as the basis of code generation. Hence, it is an important and challenging part of MDE to find and apply suitable model transformation techniques. The graph rewriting-based model transformation (or graph transformation for the sake of simplicity) is one of the most popular among them [1]. Besides CPU, there are other hardware components to accelerate the execution of these algorithms. The advantages of platform independence are obvious here, since the hardware available to the users is quite heterogeneous. In order to handle this, the OpenCL framework¹ is used in our approach. OpenCL is platform independent and can be used to handle the most widely used hardware components uniformly (CPU, GPU, FPGA, DSP). In this paper, we show that by using an OpenCL-based solution, a promising way

¹ <https://www.khronos.org/opencl>.

to accelerate model transformations efficiently can be found. Although OpenCL is rarely used in MDE tools, we provide reasons why OpenCL is moving forward in a promising direction. We also introduce the base architecture and model transformation logic of our tool.

2 Related Work

Paper [2] studies the most widely used MDE tools: GREAT, IncQuery, Fujaba, Groove, Henshin, MOLA, Viatra2 in order to understand them. Although the performance is not the most important property, these tools manage model transformations efficiently. For example, in [3] IncQuery uses the so-called incremental evaluation of queries to accelerate. Moreover, there are tools (like GMTE²) that use a C++ implementation to achieve better performance. However, none of the existing tools can efficiently use the benefit of the parallel execution architecture offered by the GPUs. OpenCL is a popular way to use the computation power of GPUs, FPGAs and many other devices. The key aspect to achieve high-performance computation is to apply appropriate scalability techniques. Improving the scalability in different contexts is an actively researched area as seen in [4]. Papers [5, 6] showed that OpenCL can be efficiently used with graphs. However, mapping graph algorithms from CPU version to OpenCL is a significant challenge. In [7], the k-Nearest Neighbor is implemented using the multi-GPU OpenCL. We should mention at this point that the paper also provides a CUDA-based implementation. CUDA is another major GPU programming platform, however, it is strongly hardware-dependent while OpenCL is not. The measurements in this paper show that the efficiency of the two platforms varies. Taking everything into account, we choose to continue working with OpenCL mainly because of its platform independence. Paper [8] shows the usage of the OpenCL with some C++ and STL related features as part of the official Boost.

3 Parallel Multiplatform Model-Transformation Engine

In this section, we introduce our solution: the Parallel Multiplatform Model-Transformation Engine (PaMMTE)³. PaMMTE is implemented in C++ 14 to maximize performance. We should note that the approach is currently limited to execute the graph transformation rules separately; no control flow support is given. However, specifying a pivot node where the match should be started can help matching.

3.1 The Representation of the Domain Model

At the beginning of the model transformation, the input domain model is read and converted using a domain specific adapter. Using the adapter, we split the

² <http://homepages.laas.fr/khalil/GMTE/>.

³ <https://www.aut.bme.hu/Pages/Research/VMTS/PaMMTE>.

input model into two sets of data. The first set is a graph representing the topology of the original model, while the second set represents the attributes attached to the model entities. During the transformation, these representations are used and the changes are evaluated on the original domain model as the last step of the transformation. Although we need to create an adapter for each domain, we provide a template to simplify the task. In the topology graph, all elements are represented by an `elementID` and a `typeID`. The `elementID` is a unique identifier of the node generated by the adapter creating the graph representation. Type information on domain elements is expressed by the `typeID` that contains the unique identifier of the type (metaelement) of the given element. Both `elementID` and `typeID` are integer values in order to accelerate their use on the GPU. In case of attributes, we create an array of data referring to the container entity by using its `elementID`. From a technical point of view, we use a hash table to build the graph and create the inner topology/attribute representation from the input domain model. The main benefit of using a hash table is its ability to find entities quickly (in $O(1)$ time). Practically speaking, matching requires several orders of magnitude more time than rewriting. Therefore, the costly operation of modifying the hash table has no serious affect on performance. The graphs are further processed by the host (the CPU) just before working with it on the GPU. This transformation is not complex, however, it is advantageous in order to simplify and accelerate the algorithms running on the GPU. The original graph is mapped into two one-dimensional structures using the `elementIDs` of the nodes: (i) The first structure contains the list of the neighbors one-by-one from the first to the last node. (ii) The second structure contains the starting positions of the neighbor list and is a helper structure to process the first. Using these two arrays and the size of the second array, all graphs can be passed to the OpenCL device. The structure of attributes is much simpler in that arrays refer to their container entity by using `elementIDs`.

3.2 Steps of the Approach

Unlike most of the tools in our approach, the execution of graph transformation rules is divided into three major logical steps (Fig. 1): (i) pattern matching, (ii) attribute processing and (iii) graph rewriting. The three logical steps are connected to each other and are executed sequentially. (i) Pattern-matching is responsible for searching for topological matches according to the user defined rewriting rules. In this step, only the aforementioned topology graph is used. (ii) Attribute processing works on the result of the first step and it filters the matching candidates by evaluating attribute constraints on them, which are evaluated separately and sequentially. If a certain constraint fails, the candidate is dropped. We have created several dedicated kernels for the most typical constraint types (e.g., regular expressions in strings, simple numeric operations, etc.). In addition to these dedicated kernels, we support using custom atypical constraints, however, they must be specified in OpenCL. To simplify this task, we are continuously working on extending the range of built-in constraints. At this point, domain attributes are also needed; thus, attribute arrays are copied

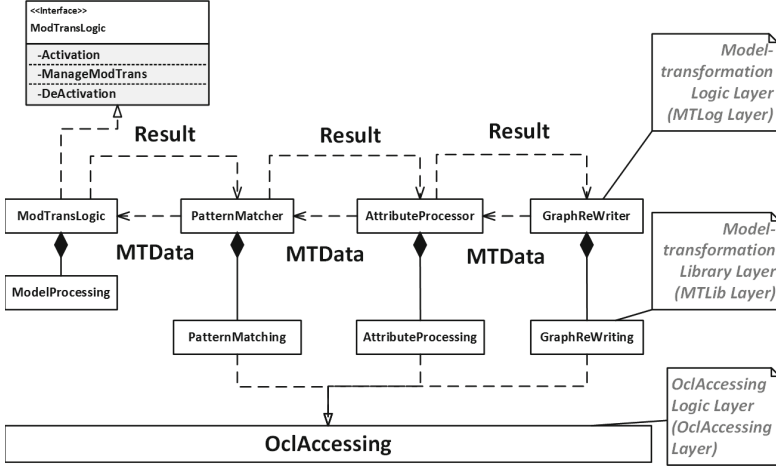


Fig. 1. The main designing concept and the three logical steps

to the GPU. It is important that kernels working on attribute constraint evaluation must receive only the necessary attribute data in the concatenated format. Otherwise, the cost of transferring the data from the CPU to the GPU would seriously degrade the performance of the approach. (iii) Rewriting applies the modifications defined in the rewriting rules by modifying the data sets representing the domain model. To avoid inconsistencies caused by parallel execution, the result is verified just before rewriting. After rewriting, we also have to decide whether the graph transformation is finished, or another rewriting is required (for “as long as possible” rules). Finally, if applying the rewriting rule is finished, the input domain model is changed based on the data sets of the transformation using the domain adapters. All three steps have input and output data, which is not stored but rather is temporally used by steps. Each step obtains an input data and then processes it and generates the output. The data is composed of three parts: (i) the model (accessed via `modelProcessing` package), (ii) the transformation rules (iii) and the temporal results. By rigidly separating the steps, a highly modular and easily extendable design is achieved. The logical steps have several kinds of responsibilities like supporting the scalability issues of the actual step, and preparing and configuring of the core algorithms, which belong to the Model-transformation Library Layer (Fig. 1). Library components can be easily exchanged to vary the dynamic behavior of the engine by using template programming. The common interface of the steps and the modularity also support the testability.

3.3 Illustrating the Topological Match

In order to illustrate the truly parallel behavior of the engine (same running time results are received in several case studies), the pattern matching logical

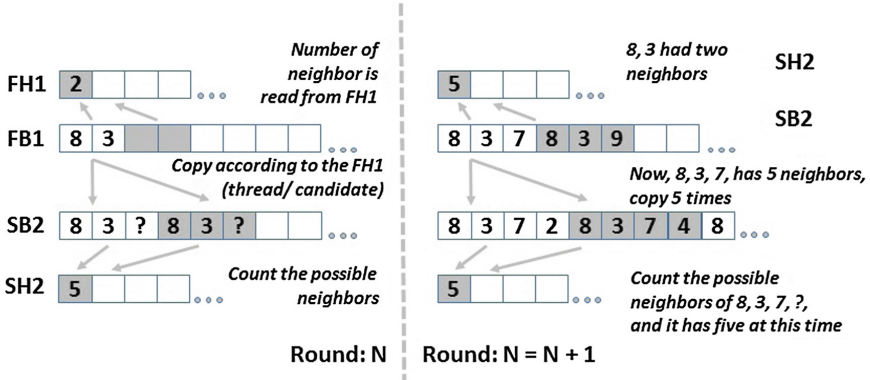


Fig. 2. Handling the buffers during pattern matching

step is detailed. The main concept is that we start a kernel from each potential matching point. Initially we try to find the first entity in the pattern, then the submatch is extended with the second entity, etc. Each kernel obtains a submatch already found and returns with its possible completion. Thus, pattern matching is applied in several steps. Four temporary buffers are used (Fig. 2) during the process: (i) FH1 - first helper, (ii) FB1 - first result candidate, (iii) SH2 - second helper, (iv) SB2 - second candidate. The kernel binary reads FH1 and FB1 and writes SH2 and SB2. The host applies two important steps before calling the kernel. First, it cumulates the numbers in the first helper buffer to provide information about the index of the candidates, then it swaps the first and second buffers. The kernels always work from the first buffers and save their result to the second: (i) The kernel copies the candidates from the first buffer to the second buffer and also takes the new neighbor using the helper buffer and the kernel worker thread ID. The number of threads started is equal to the number of new candidates. Each new thread knows its base candidate and copies the candidate from the first buffer to the second buffer. (ii) The thread knows which neighbor is to be taken to the new empty position. (iii) The thread validates whether the new candidate is matching. In the case of a mismatch, the thread sets the number of possible new neighbors to zero. If the new candidate is matching, the thread adds the number of potential new neighbors that must be checked in the next loop. Finally, the new candidate buffer is created. We have built our tool by following the principles of Test-driven development. Many test cases were created and applied from the beginning. This method helped us to find implementation issues and avoid degeneration of the code. Later on, we have searched for a domain that can be used to apply transformations. The Internet Movie Database⁴ (IMDb) was chosen. Because of its size, IMDb data is perfectly suited for scalability measurements and for performance tests. We applied several tests on the database in our earlier researches [9].

⁴ <http://www.imdb.com/interfaces>.

4 Conclusion and Future Work

The continuous growth of modeled systems is driving the focus on high performance model transformation solutions. We believe that using the remarkable potency in computing power of GPUs provides a solution to this issue. We are currently working on an OpenCL-based model transformation engine. In this paper, we introduced our framework PaMMTE by showing the basics of our approach and the most important parts of our engine, as well as illustrating the mechanisms by elaborating the steps of the pattern matching in more detail. Although our results are already promising, there are further acceleration and optimization points to discover and apply. The tool supports only the application of a single rewriting rule, not a complete sequence of rules. Our current research involves implementing a control flow that allows defining the sequence of rules and data transfer between them. The usage of further real-life case domains and studies can bring new challenges to solve. In the meantime, the achieved results can be used in MDE tools to accelerate their performance. Processing data, like Ecore, is a task for the future and it will give us a chance to create practical comparisons to other MDE tools.

References

1. Ehrig, H., Rozenberg, G., Kreowski, H.J.: Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific, Singapore (1999)
2. Jakumeit, E., Buchwald, S., Wagelaar, D., Dan, L., Hegedüs, A., Herrmannsdorfer, M., Horn, T., Kalnina, E., Krause, C., Lano, K., Lepper, M.: A survey and comparison of transformation tools based on the transformation tool contest. *Sci. Comput. Program.* **1**(85), 41–99 (2014)
3. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 76–90. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_6
4. Strüber, D., Kehrer, T., Arendt, T., Pietsch, C., Reuling, D.: Scalability of model transformations: position paper and benchmark set. In: Workshop on Scalable Model Driven Engineering, pp. 21–30 (2016)
5. Yan, X., Shi, X., Wang, L., Yang, H.: An OpenCL micro-benchmark suite for GPUs and CPUs. *J. Supercomput.* **69**(2), 693–713 (2014)
6. Xu, Q., Jeon, H., Annavaram, M.: Graph processing on GPUs: where are the bottlenecks? In: 2014 IEEE International Symposium on Workload Characterization (IISWC), pp. 140–149. IEEE, 26 October 2014
7. Masek, J., Burget, R., Povoda, L., Dutta, M.K.: Multi-GPU implementation of machine learning algorithm using CUDA and OpenCL. *Int. J. Adv. Telecommun. Electrotech. Sig. Syst.* **5**(2), 101–107 (2016)
8. Szuppe, J.: Boost. Compute: a parallel computing library for C++ based on OpenCL. In: Proceedings of the 4th International Workshop on OpenCL, p. 15. ACM, 19 April 2016
9. Fekete, T., Mezei, G.: Generic approach for pattern matching with OpenCL. In: Proceedings of the 24th High Performance Computing Symposium. Society for Computer Simulation International, p. 15. ACM, April 2016

Collaborative Modelling with Version Control

Steven Kelly  

MetaCase, Jyväskylä, Finland
stevek@metacase.com

Abstract. Modelling and version control both play key roles in industrial-scale software development, yet their integration has proved difficult. Significant effort has been expended on improving file-based merging of modellers' work, but empirical research shows the results of that approach still leave much to be desired in practice. Approaches based on multi-user modelling databases have often foundered by locking too broadly, thus preventing work on different elements in the same model, or by handling versioning themselves, leading to a silo. This article presents an approach to combining multi-user modelling with any external version control system, with no merging and no lock-outs.

Keywords: Multi-user · Modelling · Locking · Merge
Version control systems

1 Introduction

Around 15 years ago, model-driven development faced a crossroads. Where models became primary assets in software development, should collaborative work on them follow the multi-user repository approach of leading modelling tools, or the clone and merge approach that had worked so well on the earlier primary assets, source code files? In academic research, the balance swung towards XMI model files under source code version control systems. In industrial use, both approaches have continued to this day, with multi-user functionality in tools like MagicDraw, Enterprise Architect and MetaEdit+¹. Both approaches have been fruitful, yet combining them seamlessly to get the best of both worlds has proved elusive, particularly for the increasingly important case of distributed version control systems.

This article presents an approach to combining multi-user modelling in any languages with any external version control system, with no merging and no lock-outs.

2 Background and Related Research

The issue of collaboration and versioning is central to the scalability of model-driven development teams, as noted in two key items in the roadmap laid out in BigMDE 2013 [1]: “support for collaboration” and “locking and conflict management”.

¹ nomagic.com/products/magicdraw, sparxsystems.com/products/ea, metacase.com/mep.

2.1 Two Ways to Collaborate

With multiple developers interacting on a large project, it is generally not feasible to divide the system into isolated modules, one for each developer. We thus have to cope with integrating the work of several simultaneous developers on a single module.

There are two different ways to approach the problem of multiple users. The first way gives each user their own copy of the files to be edited and allows any changes, with the first user to version getting a “free ride”, and others having to merge their changes into that version after the fact. The second way allows users to work on the same set of models, integrated continuously as they save, with locks to prevent conflicting changes (Fig. 1).

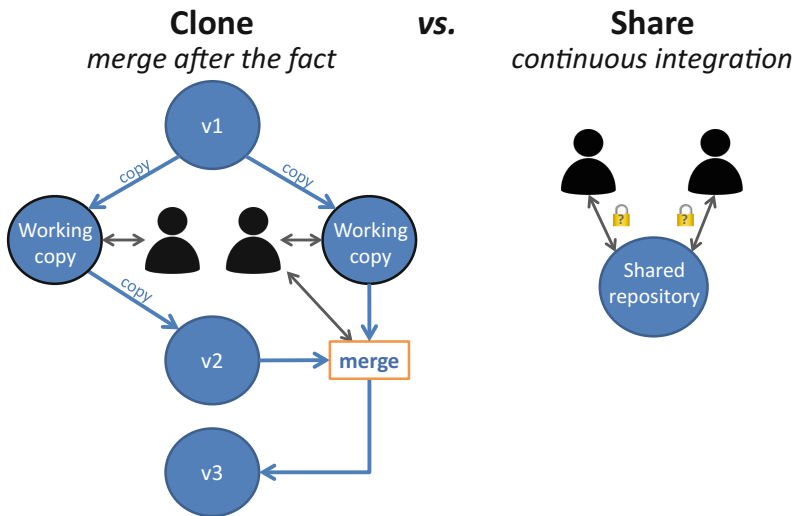


Fig. 1. Two ways to collaborate: clone vs. share

Although it is common in the research literature to refer to these two approaches as “optimistic locking” and “pessimistic locking”, these are not particularly accurate terms. In the former there is actually no locking, and in the latter there is no pessimism: a lock is taken only when necessary or requested, not as a worst-case precaution. We will thus refer to them here as “clone and merge” and “share and lock”, describing what actually happens.

2.2 Research on the “Clone and Merge” Approach

The vast majority of research has concentrated on the problem that VCS algorithms for merge are designed for simple text files, and applying them to the more highly structured and interlinked data in model files leads to results that vary between inconvenient and unacceptable. Out of 470 articles in the University of Siegen’s

excellent Bibliography on Comparison and Versioning of Software Models², almost all focus on this approach. In Europe, the MODELWARE (2005–2006) and MODELPLEX (2006–2009) projects, each costing around 20 million Euros, included significant effort on this topic. With that scale of research and effort, it seems fair to assume that theory and implementation are already close to the best that is possible on this path, and indeed we make no effort here to offer incremental improvements to them. There is a good overview of the issues in [2] and the current state in [3].

Recent empirical evaluations of user experiences with the “clone and merge” approach show significant problems remain: “Exchanging information among tools using XMI creates many problems due to the format misalignment in the various tools”, “all companies try to avoid merging as much as they can”, differencing “is error-prone and tedious” [4]. Experiences with a major vendor’s UML tool revealed “difficulties in merging... became impossible”, leading to “trying not to branch” [5].

2.3 Research on the “Share and Lock” Approach

Ohst and Kelter found that improving locking in ADAMS to use a finer granularity reduced the need to branch and merge from 30% of cases to zero [6].

Gómez et al. have tried to bring a locking approach to EMF with HBase [7], but the lack of explicit transaction support meant ACID properties were lost above the level of single changes: unacceptable for modelling, where a coherent unit of work requires several changes.

Odyssey-VCS 2 is a version control system tailored to fine-grained UML model elements [8]. It moved away from explicit branching to auto-branching, provided better information of a single user’s changes and intentions, and allowed locking at a fine granularity. However, locks must be taken explicitly, and if unused must be freed explicitly too.

Systemite’s SystemWeaver offers explicit versioning within a database, although their data is more focused on components than graphical models, and they offer no external VCS integration. They criticise file-based approaches, citing three studies showing that 14–30% of engineers’ time is spent on finding the correct data and version [9].

3 Current State of the Art

In this section, we examine a state of the art example of each approach.

3.1 Clone and Merge: EMF Compare + EGit

The file-based approach is easy to explain: model files are treated like source code files, so all the normal operations of Git or similar are available – but also required. The main benefit of EMF Compare [10] is to offer diff and merge algorithms better-suited to

² <http://pi.informatik.uni-siegen.de/CVSM/>.

EMF XMI models, and to display results in a partially graphical format. The simplest EMF Compare + EGit integration seen to date is in an EclipseSource video³ by Philip Langer. Even there, versioning with a single simple conflict requires a sequence of 11 user operations: “branch + checkout, save, add + commit, checkout, pull, merge, resolve conflicts, save, add, commit, push”. Even without a conflict, the sequence is “branch + checkout, save, add + commit, checkout, pull, merge + commit, push”: seven manual operations which the user has to remember and find within the various menus of Eclipse. (Watching the video is recommended to understand the current state of the art, and compare the improvements offered in this paper.)

Is this complexity essential, or could another way show it to be accidental complexity, introduced by applying tools and methods designed originally for simple text files to the rather different world of models? Indeed, could the more structured nature of models actually help us avoid the need for this complexity? There are certainly substantial gains to be found: empirical research finds that modellers spend over one hour a day on interacting with version control systems [11].

3.2 Share and Lock: MetaEdit+

The second way has its origins in the richer structures of databases. In today’s multi-user modelling tools, the second way is seen in MetaEdit+ [12] or the collaboration mode of the commercial version of Obeo Designer⁴. The basics of the approach are similar between the tools, but here we will discuss the approach taken in MetaEdit+, as it will also serve as a starting point for our later discussions.

Figure 2 shows the architecture of the MetaEdit+ multi-user version [13], which we describe in this section as it was from release in 1996 to before the work in this paper. The heart of the multi-user environment is the Object Repository, a database running on a central server. All information in MetaEdit+ is stored in the Object Repository,

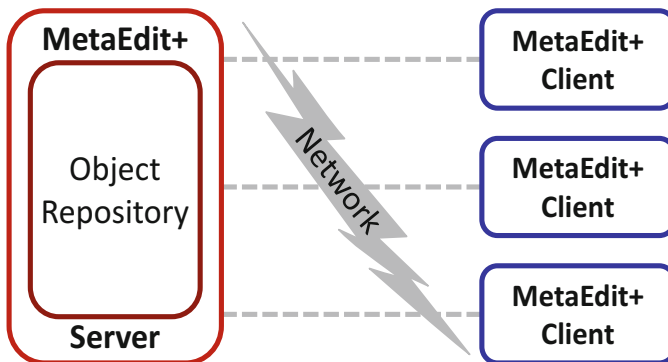


Fig. 2. MetaEdit+ multi-user architecture

³ <https://youtu.be/NScFYAukYgk>.

⁴ <https://www.obeodesigner.com/>.

including modelling languages, diagrams, objects, properties, and even font selections. Clients access the repository over the network via the lightweight MetaEdit+ Server; the user interface and execution of all modelling operations is in the clients.

The Object Repository itself is designed to be mostly invisible to users. Objects are loaded as needed and cached by clients when read, and thus are only read once per session over the network: performance after that initial read is identical to normal objects. Because of the JIT loading of objects, and no need to parse XML, performance of opening a model compares favourably with XML-based tools, particularly for large models [14]. This corresponds to the “Efficient model storage” and “Model indexing” items of the scalability roadmap [1]. Objects are also automatically flushed from the cache if models become larger than the configured desired memory usage, allowing seamless work and global queries on massive models (a similar approach has been used later by Daniel et al. on Neo4EMF [15]).

An extra layer of consistency is familiar from ACID rules for database transactions: during a transaction, users see a consistent view of the repository as it was at the start of their transaction. This gives the ease of Google Docs multi-user editing, without the distraction of others’ half-finished changes happening before your eyes – a problem that users of other tools have described as feeling “like shooting at a moving target” [16].

Fine granularity locks ensure no conflicts, while letting users work closely together in parallel. Making a change automatically takes a minimal lock, down to the level of granularity of a single property, preventing conflicts without preventing users working closely together. When a user has finished making a coherent set of changes, he can commit and his changes are then automatically available to other clients, with no need for manual integration.

Whereas the “clone” approach to collaboration gives rise to versions as a welcome side-effect, the “share” approach does not itself create versions. Although VCS functionality is not needed here to enable collaboration, we still need versioning in order to be able to record what we have shipped. To save the state of the repository to a version control system, all the users first had to log out and the server process had to be stopped: an unwelcome interruption and a harsh return to the file-based world.

Similarly, whereas the “clone” approach must calculate differences, and by convention requests the developer to enter a human-readable description of the changes, the “share” approach does not do this. We still need to see what has changed as an aid to documentation, bug hunting and impact analysis, so extra functionality is needed to help developers capture that information.

4 Comparison Functionality in a Modelling Tool

We want a way to bring together the best of both approaches above. We will start from MetaEdit+ as described above, and show in this section how we have now added comparison functionality directly in the modelling tool, avoiding the complication of reconstituting it by comparing lower-level XMI files. The section after this covers the other novel addition of this paper, integration with external version control systems.

4.1 View Changes as a Tree

The Changes & Versions Tool shows what changes users have made, helping them document the version history of their models. A Version is made by a user on demand, and consists of the one or more transactions by that user since his previous Version. The history is shown as a tree with Versions at the top level, containing that user's Transactions, containing Graphs and Objects that changed in that transaction.

By default the tree shows only the current Working Version of the current user, but 'Show all versions' broadens this to previous versions and all users. Users can choose 'Ignore representations', so simple graphical rearrangements of diagram elements do not clutter the display.

Colour and symbols are used to highlight specific types of change: green for new, red for deleted, black for changed, and grey for elements that are not themselves changed, but which are parents for changed elements (Fig. 3).

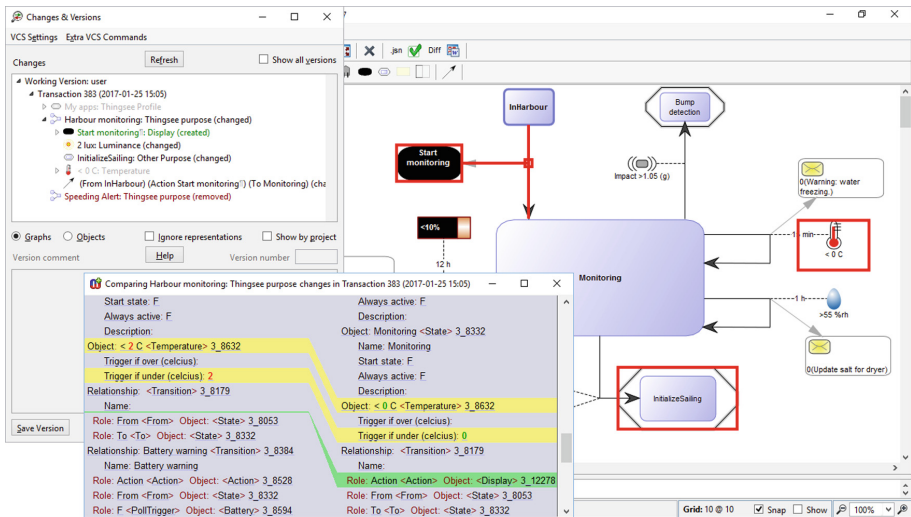


Fig. 3. Changes & Versions Tool with tree view, graphical view and text view (Color figure online)

4.2 Graphical Comparison to Highlight Changes

For a selected graph version in the Changes tree, 'Open' will open the current diagram and highlight the elements changed in that version (Fig. 3). For many cases, this highlighting gives a quick, intuitive, graphical overview of what has changed, shown in the context of the current state of the model. For the full details, e.g. of sub-objects not directly represented in a diagram, users can use the tree or textual Compare.

4.3 Textual Comparison with Model Links

From the Changes tree, users can choose 'Compare' to compare the selected Version, Transaction or Graph with its predecessor. This will open a text comparison of the

selected state with its previous state (Fig. 3). The text elements have Live Code hyperlinks, so double-clicking a link will open that element in its current context in the model. The text lists the graph, its objects and relationships, together with all their details and links to any subgraphs. This approach is intended as a good compromise between coverage and readability. It is produced by a normal MERL generator, and so can be customized where necessary: domain-specific diff.

When comparing a Version, the comparison is to the previous Version, hence multiple graphs across multiple transactions. Rather than show each graph many times, each time with just the changes made in a single transaction, the changes for each graph will be combined and shown from the initial state to the final state. Only if other users have committed transactions interleaved with this user's, those changes will be filtered out of this user's view by splitting at that point into more than one textual comparison. Each change is thus shown once, in the Version of the user who made it, and can thus be documented once, without the confusion of interleaved changes made by others being highlighted too.

5 Version Control System Integration

We add integration with external Version Control Systems using MetaEdit+'s API command-line operations and MERL (the MetaEdit+ Reporting Language for writing generators and scripts). The VCS operations are invoked from the Changes & Versions Tool, and use generators to call the necessary commands. Full implementations with generators for Git and TortoiseSVN integration are included for Windows, and users can build on these to add support for other VCSs or platforms. For instance, the TortoiseSVN implementation for versioning consists of one line for "svn update" and another for TortoiseSVN's commit command, plus some batch file housekeeping.

5.1 What to Version and How

A user can make changes and commit in one or more transactions, and when ready to version, can press 'Save Version' in the Changes & Versions Tool. The VCS integration puts the current state into the VCS working directory, and makes a VCS version from there.

The VCS working directory is kept separate from the MetaEdit+ directory, because VCSs cannot work sensibly with a live set of database files. MetaEdit+ copies the repository into a `versionedDB` subdirectory of the VCS working directory. (The repository's binary files' history compresses twice as well with standard Git or SVN deltas than if 7-Zipped first.) MetaEdit+ also writes the current textual snapshots of each graph, and in a `metamodel` subdirectory textual snapshots of the metamodels and generators. These textual files add about 10% to the initial size, but as only small parts change in subsequent versions, the ongoing increase is significantly less. The improvement in the ability to compare versions within the VCS itself is well worth it.

The standard way of using command-line VCSs – manually change a file, and in a separate command manually tell the VCS that the file has changed – is not necessary in a situation where a tool can guarantee that what is in the working directory is the exact

state of the next version. Only a few VCS commands are thus needed, and MetaEdit+ will handle calling them. The commands are specified in a few MERL generators called by the various phases of the Save Version action. The bulk of the generators are for any VCS, with any variations isolated into generators with a suffix for the VCS name – e.g. `_vcsCheckIn_git()` checks in the current working directory contents, adding and committing them locally, and syncing and pushing them to a remote Git repository shared by all users of this MetaEdit+ repository.

Having the commands in MERL generators allows users to add integration for new VCSs, and to tweak existing integration if necessary. For instance, the default integration follows most customers' wishes in not including source code generated from the models, but if a particular customer wanted that, adding it for all VCSs would be a single line in the generic `_vcsCheckIn()` generator. Another customer might choose to generate source code into a separate VCS repository.

We want this new version to succeed the latest version in the VCS (regardless of who made that), and not the previous version made by this user. We thus perform an update or reset operation on the local VCS working directory before versioning, to bring the local VCS's view up to date. Since the MetaEdit+ multi-user server has already integrated the work of multiple users, the state of the repository seen by this user is exactly what we want in the next version, and we can simply write it to the working directory as detailed above. Before writing we empty the working directory, allowing us to avoid old files being left when graphs etc. have been deleted. There are thus no merges or conflicts, and we can simply commit the state of the working directory as the next version in the VCS. Repository, local and remote VCS stay in sync.

In this way, versioning in a multi-user environment is as easy as with a single user (see video⁵). The multi-user repository already makes sure we have all the other users' changes, and there is no need for the user to manually fetch others' changes, deal with diff, merge and conflicts, or think about any of these details when versioning: one click is enough to publish as the next version in a single, consistent trunk. This compares favourably with the 7–11 operations and choices the user needs to make correctly in the best Eclipse implementations (Sect. 3.1 above).

6 Conclusions

Over the last 15 years, much effort has been expended on improving collaboration and versioning with models. Two starting points were on offer: “clone and merge” with file-based models under a VCS, and “share and lock” with multi-user modelling tools. The vast majority of research effort has concentrated on the “clone and merge” branch, and the results obtained can be considered to approach the local maximum of functionality and local minimum of complexity on that branch. This paper presents work on the less-investigated “share and lock” branch, starting with a multi-user modelling tool and adding comprehensive model comparison functionality and integration with any

⁵ <https://youtu.be/nvGQlt8dqjI>.

VCS. The results seem promising in offering similar functionality to the best examples on the “clone and merge” branch, but at a significantly lower level of complexity to the user. The number of explicit manual user operations needed to version is an order of magnitude lower than with EMF Compare and EGit, helping modellers to stay focused on their primary task: modelling.

In today’s world of near-ubiquitous internet access, the requirement of a network connection to a multi-user repository is not a large one. For the rare cases where that is not possible, future work will look at adding the 3- or 4-way merges from tools like EMF Compare or CDO⁶, allowing offline work to be more easily integrated.

Acknowledgments. We would like to thank MetaCase’s Risto Pohjonen and Juha-Pekka Tolvanen for implementation work and discussion, and Jordi Cabot for making an early version of this material available for comment on his blog (<http://modeling-languages.com/smart-model-versioning/>).

References

1. Kolovos, D., Rose, L., Matragkas, N., Paige, R., Guerra, E., Cuadrado, J., De Lara, J., Ráth, I., Varró, D., Tisi, M., et al.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, p. 2 (2013)
2. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. *Int. J. Web Inf. Syst.* **5**, 271–304 (2009)
3. Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: An introduction to model versioning. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) *SFM 2012*. LNCS, vol. 7320, pp. 336–398. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30982-3_10
4. Cicchetti, A., Ciccozzi, F., Carlson, J.: Software evolution management: industrial practices. In: Proceedings of the 10th Workshop on Models and Evolution Co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, 2 October 2016, pp. 8–13 (2016)
5. Burden, H., Heldal, R., Whittle, J.: Comparing and contrasting model-driven engineering at three large companies. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, p. 14 (2014)
6. Ohst, D., Kelter, U.: A fine-grained version and configuration model in analysis and design. In: Proceedings of the 18th International Conference on Software Maintenance, pp. 521–527 (2002)
7. Gómez, A., Benellallam, A., Tisi, M.: Decentralized model persistence for distributed computing. In: Proceedings of the 3rd Workshop on Scalable Model Driven Engineering Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, L’Aquila, Italy, 23 July 2015, pp. 42–51 (2015)
8. Murta, L., Corrêa, C., Prudêncio, J., Werner, C.: Towards Odyssey-VCS 2: improvements over a UML-based version control system. In: Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models, New York, NY, USA, pp. 25–30 (2008)

⁶ <https://eclipse.org/cdo/>.

9. Shahrokni, A., Söderberg, J.: Beyond information silos challenges in integrating industrial model-based data. In: Proceedings of the 3rd Workshop on Scalable Model Driven Engineering Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, L'Aquila, Italy, 23 July 2015, pp. 63–72 (2015)
10. Brun, C., Pierantonio, A.: Model differences in the Eclipse modeling framework. *UPGRADE Eur. J. Inf. Prof.* **9**, 29–34 (2008)
11. Kalliamvakou, E., Palyart, M., Murphy, G., Damian, D.: A field study of modellers at work. In: Proceedings of the Seventh International Workshop on Modeling in Software Engineering, pp. 25–29 (2015)
12. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+ a fully configurable multi-user and multi-tool CASE and CAME environment. In: Constantopoulos, P., Mylopoulos, J., Vassiliou, Y. (eds.) *CAISE 1996. LNCS*, vol. 1080, pp. 1–21. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61292-0_1
13. Kelly, S.: CASE tool support for co-operative work in information system design. In: Information Systems in the WWW Environment, IFIP TC8/WG8.1 Working Conference, Beijing, China, 15–17 July 1998, pp. 49–69 (1998)
14. Boersma, M.: Language Workbench Challenge 2014. <http://www.languageworkbenches.net/2014/08/lwc2014-the-participants/>
15. Daniel, G., Sunyé, G., Benelallam, A., Tisi, M.: Improving memory efficiency for processing large-scale models. In: Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering Co-located with the Software Technologies: Applications and Foundations Conference, BigMDE@STAF2014, York, UK, 24 July 2014, pp. 31–39 (2014)
16. Bendix, L., Emanuelsson, P.: Collaborative work with software models-industrial experience and requirements. In: International Conference on Model-Based Systems Engineering, MBSE 2009, pp. 60–68 (2009)

A Java Bytecode Metamodel for Composable Program Analyses

Bugra M. Yildiz¹, Christoph Bockisch²(✉),
Arend Rensink¹, and Mehmet Aksit¹

¹ University of Twente, Enschede, The Netherlands
{b.m.yildiz,arend.rensink,m.aksit}@utwente.nl

² Philipps-Universität Marburg, Marburg, Germany
bockisch@mathematik.uni-marburg.de

Abstract. Program analyses are an important tool to check if a system fulfills its specification. A typical implementation strategy for program analyses is to use an imperative, general-purpose language like Java; and access the program to be analyzed through libraries for manipulating intermediate code, such as ASM for Java bytecode. We show that this hampers composability, interoperability and reuse of analysis implementations.

We propose a complete Ecore-metamodel for Java bytecode as a common basis for program analysis implementations, as well as an Eclipse plug-in to create bytecode metamodel instances from Java bytecode and vice versa. Code analyses can be defined as model transformations in a declarative, domain-specific language. As a consequence, the implementations of program analyses become more composable and more modular in general. We demonstrate the effectiveness of this approach with a case study.

Keywords: Java bytecode · Metamodel · Model transformation
Model-driven software engineering · Program analyses · Composition

1 Introduction

Program analyses are developed for, e.g., checking correctness, performance or real-time requirements. In general, such analyses either determine statically accessible properties, or they modify the code such that information is collected at execution time. Thus, analyses are either static or dynamic, or hybrid analyses combining both [14, 18, 19]. Much of the research in program analysis targets the Java language; more precisely, it should be said that the bytecode format of the Java Virtual Machine is targeted, as analyses usually inspect and instrument this intermediate representation rather than the source code. This has several advantages; for example, many different source languages compile to the Java bytecode format, thus, multiple languages can be supported at once. Furthermore, typically, the bytecode is available for the whole program, including third-party

libraries. A typical implementation strategy for bytecode-level program analyses is to use an imperative, general-purpose language like Java, and to access the program to be analyzed through libraries that offer an API for inspecting or instrumenting intermediate code, such as BCEL or ASM [5, 9, 11, 15, 22].

New analyses often *conceptually* extend or combine existing ones by optimizing the information collection or collecting additional information. Considering that many of these analyses involve similar concepts and work on the same intermediate representation, being able to extend and compose existing analysis implementations would save time and effort, and improve interoperability and maintenance. Though Java (and other general-purpose languages) offers language-level modularity mechanisms, such as class libraries and inheritance, they are not sufficient to implement program analyses in a composable fashion. To compose two or more analyses (or parts thereof), the only possibility is to apply one analysis to the output of the previous one. However, since an analysis in general alters the bytecode, the subsequent ones do not see the original code, which may invalidate their results. In this paper, we promote model-based definitions of program analyses as a more flexible mechanism [6].

The contribution of this paper, therefore, is a complete Ecore-metamodel for Java bytecode, which can be used as a common basis for arbitrary program analyses. Instances of our metamodel can be created from compiled Java code in the class file format, and vice versa. Code analyses can now be defined as model transformations, in one of the well-researched domain-specific languages available for this purpose. Furthermore, analysis results can be represented directly as extensions of the bytecode model of the analyzed program, making them easily accessible to subsequent manipulation and to other tools. We claim that, as a consequence, the implementation of a program analysis becomes more composable and modular [21]. We have implemented an Eclipse plug-in, called *JBCPP* for the bytecode-to-model and model-to-bytecode transformations.¹

We demonstrate the effectiveness of this approach with a motivating example comparing the composability of two program analyses in the traditional (using a general-purpose programming language) and in our implementation approach (using a model transformation language utilizing our metamodel).

2 Motivation

While there are metamodels available for high-level programming languages (e.g., JaMoPP [17], MoDisco [7] and domain-specific languages developed with tools like xText [13] or EMFText [16]), we are not aware of any program analyses developed in a model-driven way using these source-level metamodels. There are several reasons why program analyses are typically implemented based on bytecode rather than source code:

- The bytecode is always available for the whole program, also for the third-party components and libraries that are not available in source code.

¹ The plug-in and the metamodel are available on the JBCPP homepage: <https://bitbucket.org/bmyildiz/java-bytecode-metamodel-repository>.

- Many implicit features in the source code are resolved and represented explicitly in bytecode. Some examples are simple type names, which can only be fully qualified by interpreting `import` statements, or the implicit default constructors.
- Java bytecode is the compilation target for various languages. Therefore, implementing an analysis for Java bytecode generally makes it applicable to programs written in these different source languages.
- A single statement in the source language is typically represented by multiple finer-grained bytecode instructions, which makes Java bytecode more flexible. For example, control flow is not limited to properly nested blocks. Therefore code instrumentations often are easier to be defined at the bytecode level.
- Besides, at least for Java programs, almost no information from the source level gets lost when compiling to bytecode. A notable exception are scopes confined to blocks in the source code, e.g., for a `for` loop, the sections initialization, condition, increment and body are not explicitly represented but can typically be recognized by simple and local analyses [23].

2.1 Motivating Example

We will express our problem statement by employing two explanatory program analyses. The first analysis is to count how often each method call in the program is executed. To do so, this analysis instruments each invocation instruction in the bytecode by inserting a call to the method `InvocationCount.increase()`. To identify the instruction whose executions are counted, the analysis numbers all invocation instructions in a method and generates an identifier based on the fully qualified method name, that contains the invocation instruction and the instruction's number. This identifier is passed to the `increase()` method as an argument. After the program execution, the results are written to a file. We call this analysis *invocation count*.

The second analysis measures the time that elapses for each method invocation. This analysis prepends each method invocation with code to store the result of `System.currentTimeMillis()` in a local variable. Then, it appends code to calculate the difference of the current time and the stored start time and to pass the result to the method `Time.increase()`. This method receives a unique identifier of the invocation instruction, which is computed in the same way as for the invocation count analysis. The method `Time.increase()` stores the accumulated elapsed time per invocation instruction, which again is dumped at the termination of the execution. We call this analysis *time* in short.

Neither analysis instruments invocations that occur in its respective `increase()` method or invocations of methods from the system class library, i.e., classes in a subpackage of `java`, to avoid endless recursions.

2.2 Implementing Program Analyses with a Bytecode Toolkit

Several toolkits for reading and manipulating Java bytecode are available [3, 8, 10]. These toolkits basically support two styles for implementing program

analyses: First, the bytecode is transformed to an object-oriented representation and analyses subsequently process this representation, possibly altering it; this facilitates random access to elements in the bytecode. Second, the bytecode is traversed in one pass during which a visitor, implementing the analysis, reacts to encountering relevant elements; in this style the bytecode is naturally traversed sequentially. If a such-implemented analysis employs code instrumentation, each encountered element is, by default, copied to the output, unless the analysis decides to suppress or modify a visited element or insert something at its location.

Both styles have the following points in common: (1) By default, there is no way to combine multiple analyses other than to perform them sequentially; and (2) it is not possible identify which elements in the resulting bytecode stem from the original input or are inserted by an analysis.

In our examples, we employ the ASM bytecode toolkit and make use of the visitor style since this is currently the most common implementation approach for program analyses based on Java bytecode.

Listing 1.1 shows the visitor methods handling the encounter of a method invocation of *invocation count*. What is not shown in the listing is that the visitor does not descend into the methods `InvocationCount.increase()` but copies them verbatim, such that the method invocation instructions within this methods are not visited. The visitor method of *time* is implemented analogously.

```

1  @Override
2  public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean
   itf) {
3      if (!owner.startsWith("java/")) {
4          String id = getInstructionID();
5          super.visitLdcInsn(id);
6          super.visitMethodInsn(Opcodes.INVOKESTATIC, "ic_analysis/InvocationCount",
   "increase", "(Ljava/lang/String;V", false);
7      }
8      super.visitMethodInsn(opcode, owner, name, desc, itf);
9  }

```

Listing 1.1. Instrumenting method invocations for the invocation count analysis with the ASM toolkit.

As the listing shows, the instrumentation code is inserted into the output by calling the respective `super.visitXXX()` methods. This is the case for original instructions occurring in the input (e.g., line 8 in Listing 1.1) as well as the additional code. The invocations to `getInstructionID()` (e.g., line 4 in Listing 1.1) return the unique identifier of the method call instruction, as described in the previous subsection.

2.3 Composing the Toolkit-Based Analyses

The two described analyses measure the total execution time of each method call as well as an execution count for each method call. Thus, composing both analyses would allow to compute the average execution time for each method call.

The typical approach of combining two analyses implemented in an approach such as outlined above is to apply them sequentially. This is, analysis 1 is applied to the original bytecode yielding intermediate bytecode as result. Second, analysis 2 is applied to this intermediate code yielding the final bytecode.

In the case of our example, both analysis instrument all method invocations: This causes the intermediate code produced by the first instrumentation to contain additional method invocations, which are then unintentionally instrumented by the second analysis. As a result, depending on the application order of the two instrumentations, either method invocations are counted or timed, which were not part of the original program.

Furthermore, the identifier computed for each method call (`getInstructionID`) is based on the position of the instruction in the bytecode, which changes because of the instrumentations. Therefore, the identifiers of both analyses do not match. Only the identifiers of the first analysis can be mapped to the original bytecode.

One might think that analyses implemented in the outlined approach could also be composed by inheritance. However, this does not solve the problem, as the visitor method (`visitMethodInsn`), which is implemented to react to the encounter of a method call, is also called to insert additional method calls.

2.4 Problem Statement

The current state of the art bytecode manipulation toolkits follow approaches that do not support composition of independently developed program analyses. While it could be possible to employ specific patterns for extensibility and composability when implementing a program analysis, we are not aware of such approaches, let alone bytecode manipulation toolkits supporting this.

For this reason, we suggest a new approach, implementing program analyses in a model-driven way. Model transformation approaches from the model-driven engineering (MDE) world have a strong focus of declarative definitions, composability and extensibility, which is why we think that the ability to implement program analyses as model transformations is a significant added value.

3 Java Bytecode Metamodel

To facilitate implementing program analyses in a model-driven way, we have developed a metamodel (using the Ecore format provided by the Eclipse Modeling Framework (EMF) [1]) of Java bytecode, called *JBCMM*². Bytecode models, instances of the metamodel, act as a basis for analyses. Furthermore, the metamodel can be extended to meet various concerns such as the representation of analysis results. In our metamodel, all relevant elements are uniquely identifiable. For example, classes have a unique name (made up of the package name plus the simple class name), and the name and descriptor of a method is unique within

² Published on the JBCPP homepage: <https://bitbucket.org/bmyildiz/java-bytecode-metamodel-repository>.

a class. In addition to the names used in the bytecode specification, additional elements like instructions have names in our metamodel to make them identifiable. These names have to be unique within their scope, e.g. the fully qualified identifier of an instruction is composed of the instruction name, the containing method's name and descriptor and the declaring class's name.

Therefore, information generated by an analysis can be uniquely associated with model elements by using these fully qualified identifiers, staying valid independently of which modifications are applied to the model. Java bytecode (thus also our metamodel) accommodates for storing a mapping between bytecode and the source line from which it was compiled. Therefore, it is also possible to trace each object in a JBCMM model to the corresponding source line.

3.1 Structure of Java Bytecode Metamodel

The metamodel mainly follows the organization of Java class files as defined in the Java Virtual Machine specification [2]. In general, each kind of entity from the class file format (like method declarations, attributes or instructions) is represented as one Ecore class in the metamodel. Lexical nesting (e.g., a method is nested inside its declaring class) is represented as a containment relationship in the metamodel (in terms of the previous example: a method is contained in the class that declares it). To simplify implementation of analyses, all containment relationships are navigable bidirectionally.

The most relevant elements of the metamodel are shown in graphical form in Fig. 1 and described below. Entities not relevant for our case studies (e.g., fields) are omitted here, but are treated analogously.

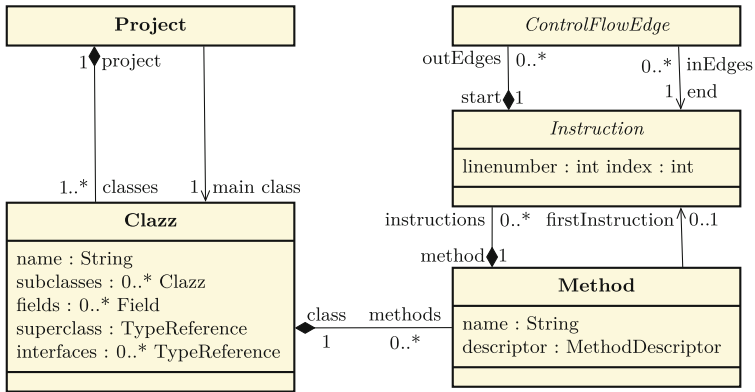


Fig. 1. A view from the bytecode metamodel

Project, the root of the Ecore model, contains all classes and refers to the designated main class. **Clazz** represents a class or an interface, storing type hierarchy information, type-level declarations such as annotations or modifiers, and

all the members declared inside the class. `Method` stores method-level declarations and (for non-abstract methods) contains the method's instructions and a reference to the first instruction.

`Instruction` is an abstract entity that build the root of a hierarchy of bytecode instruction entities, containing the properties shared all instruction types. Also, an instruction contains zero or more `ControlFlowEdges`. The subclasses of `Instruction` form a hierarchy organized according to shared semantics and, thus, also shared structure of instructions. For example, all instructions for invoking methods are represented as the subtype `MethodInstruction`, which is further extended by types for the specific kinds of invocation such as `InvokeStaticInstruction`.

The control flow information, which is implicitly available in the class file through the ordering of instructions or targets of jump instructions, is explicitly stored as a property of an instruction and is presented in our metamodel via a hierarchy of `ControlFlowEdges` between instructions. Concrete types of edges are unconditional, different types of conditional or exceptional control flow edges.

3.2 JBCPP Plug-in

To conveniently create instances of our Java bytecode metamodel from existing code, we have developed an Eclipse plug-in, called *Java Bytecode++ (JBCPP)*.

In [24] we have evaluated the performance of JBCPP when processing projects at different scales. The largest model derived from a Java program with over 1,000 classes consisted of over 750,000 objects. The generation of the model took almost 90 min. This shows that our approach is feasible at least for, e.g., nightly analysis runs, but performance improvements are needed.

```

1 pattern InstrumentInvocationCountIncrease
2   thisMethodInstruction:InOutJBCModel!MethodInstruction {
3   match: thisMethodInstruction.isPatternApplicable()
4
5   do {
6     var newLcdInstruction = thisMethodInstruction.getNewLcdStringInstruction(
7       thisMethodInstruction.uuid );
8     var newInvokeStaticInstruction =
9       thisMethodInstruction.getNewInvokeStaticInstruction();
10    var parameterList = new OrderedSet();
11    parameterList.add("Ljava/lang/String;");
12    newInvokeStaticInstruction.methodReference = getMethodReference( "increase",
13      "Lic_analysis/InvocationCount;", "V" , parameterList);
14    createNewUnconditionalEdge(newLcdInstruction, newInvokeStaticInstruction);
15    insertBefore( newInvokeStaticInstruction, thisMethodInstruction);
16  }
17 }
```

Listing 1.2. Implementation of the invocation count analysis as a model transformation.

We have used the Epsilon Pattern Language (EPL) to implement these transformations, which is one of the domain-specific languages for model management tasks provided by the Epsilon language family [20].

In EPL, the transformation actions are defined in terms of *patterns*, which in the first place filter by the type of model objects to which they are applicable. Second, the `match` part can filter based on the properties and attributes of the selected model object. When both the type-based and the property-based filtering selects a model object, the transformation specified in the `do` part is executed.

For our example, the pattern is defined on `MethodInstruction` instances, represented by `thisMethodInstruction`. The guard in line 4 checks if `thisMethodInstruction` is not a call to a method of classes in the system library and that the call does not appear inside the analysis' `increase()` method. After the matching, the `do` part starts. From line 6 to line 10, the two bytecode instructions that will be inserted are generated. In line 11, these bytecode instructions are connected with a control flow edge. Finally, in line 12, the newly created instructions are inserted before `thisMethodInstruction` via the `insertBefore` operation. This operation redirects any incoming edges of `thisMethodInstruction` to the first instruction of instrumentation, and creates a new control flow edge between the last instruction of instrumentation and `thisMethodInstruction`.

Patterns implemented independently in different EPL modules can be easily composed: A new transformation can be implemented that imports both the module for the invocation count and the module for the time analysis. Then, both modules will be applied to the same input model at once, yielding one output model that has the extra (instrumented) instructions added by both analyses for all method call instructions present in the input model – and only for these instructions. Since both analyses use the unique identifiers of call-instruction objects in the input model, the data produced by both analyses can be easily mapped back to the original method calls.

4 Related Work

There are not many attempts in the field of metamodeling of bytecode. Eichberg et al. [12] provide an XML Schema-based metamodel of bytecode supporting multiple instruction set architectures, such as Java bytecode. They report the benefits of using an explicit metamodel: ease of changing and extending a metamodel in case of new requirements, and facilitating the development of generic analyses with the help of a well-defined data structure. A similar approach, however working at the level of source code is MoDisco [7]. Their approach is to derive a language-independent model from source code, which then acts as store for analysis results. Like the work of Eichberg et al., they do not facilitate code instrumentation using the model.

Heidenreich et al. [17] propose an Ecore metamodel for the Java source code language, including a parser to create instances of this metamodel from Java code and Eclipse plug-ins to create Java source code from the instances of this metamodel. We can use JaMoPP to investigate our claim that implementing (hybrid) analyses as transformations of a bytecode metamodel is more suitable than using a source code metamodel.

5 Conclusion

In this paper, we have presented our complete Java bytecode metamodel and the JBCPP plug-in to be used for the bytecode-to-model and model-to-bytecode transformations. The metamodel allows code analyses to be written as model transformations in well-studied domain-specific languages. In this way, program analyses become more composable and the results of these analyses can be associated with the entities in bytecode via unique identifiers, which have been demonstrated with the motivating example.

The scalability of this approach has not yet been evaluated systematically, but we have made an initial assessment. For four realistically sized programs we derived bytecode models, transformed them and converted them back to bytecode (also cf. [24]). The relevant sizes and times are shown in Table below. The data already shows that our approach is feasible even of realistic programs. We expect significant performance gains through better engineering of the derivation and bytecode generation. By incrementalizing our approach we believe that the performance can reach a sufficient level for use in practice.

	Model size					Duration [seconds]			
	Classes	Methods	Instr.	Flow edges	Total	Deriv.	Trans.	To bc	Total
LiveGraph	131	350	11,795	11,740	24,016	18	51	35	104
Groove Gen.	930	5,392	99,738	98,634	204,694	1,414	86	364	1,864
Groove Sim.	1,482	9,232	203,030	203,071	416,815	1,480	300	977	2,757
Weka	1,041	8,322	367,774	374,854	751,991	764	803	2,402	3,969

As future work, we will re-implement several published static and dynamic analyses in our approach and compare them to their original implementations. One result of this exercise will be the provision of a library of reusable and composable fine-grained building blocks of program analyses.

In our previous work, we have proposed a framework to derive timed-automata models for model checking purposes from instances of an earlier version of the bytecode metamodel [24]. The framework transforms the bytecode models to extended models in order to handle recursion and to enrich them with loop and timing information. All these steps are implemented via model transformations. At the end of the process, the framework produces timed-automata models compatible with the UPPAAL [4] model checker. We will update this framework to the most recent version of our Java bytecode metamodel.

References

1. Eclipse Modeling Framework (2016). <https://www.eclipse.org/modeling/emf/>
2. Java Class File Format, May 2016. <https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html>
3. BCEL, June 2016. <https://commons.apache.org/proper/commons-bcel/>

4. Bengtsson, J., Larsen, K., Larsson, F., Petterson, P., Yi, W.: UPPAAL — a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0020949>
5. Binder, W., Hulaas, J., Moret, P.: Reengineering standard Java runtime systems through dynamic bytecode instrumentation. In: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 91–100, September 2007
6. Bockisch, C., Sewe, A., Yin, H., Mezini, M., Aksit, M.: An in-depth look at ALIA4J. *J. Object Technol.* **11**(1), 7:1–7:28 (2012)
7. Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. *IST* **56**(8), 1012–1032 (2014)
8. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems (2002). <http://asm.ow2.org/current/asm-eng.pdf>
9. Chander, A., Mitchell, J.C., Shin, I.: Mobile code security by Java bytecode instrumentation. In: Proceedings of DARPA Information Survivability Conference and Exposition II, DISCEX 2001, vol. 2, pp. 27–40 (2001)
10. Chiba, S.: Load-time structural reflection in Java. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 313–336. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45102-1_16
11. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE TSE* **30**(12), 859–872 (2004)
12. Eichberg, M., Monperrus, M., Kloppenburg, S., Mezini, M.: Model-driven engineering of machine executable code. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 104–115. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13595-8_10
13. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of OOPSLA, pp. 307–309. ACM (2010)
14. Fairley, R.E.: Tutorial: static analysis and dynamic testing of computer software. *Computer* **11**(4), 14–23 (1978)
15. Gates, A.Q., Mondragon, O., Payne, M., Roach, S.: Instrumentation of intermediate code for runtime verification. In: Proceedings of 28th Annual NASA Goddard Software Engineering Workshop, pp. 66–71, December 2003
16. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Model-based language engineering with EMFText. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2011. LNCS, vol. 7680, pp. 322–345. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35992-7_9
17. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the gap between modelling and Java. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 374–383. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12107-4_25
18. Lindlan, K.A., Cuny, J., Malony, A.D., Shende, S., Mohr, B., Rivenburgh, R., Rasmussen, C.: A tool framework for static and dynamic analysis of object-oriented software with templates. In: ACM/IEEE SC, p. 49, November 2000
19. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200 (2005)
20. 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 International Conference on Engineering of

Complex Computer Systems, pp. 162–171. IEEE Computer Society (2009). <http://dx.doi.org/10.1109/ICECCS.2009.14>

21. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. *Sigplan Not.* **35**(6), 26–36 (2000)
22. Wang, C., Mao, X., Dai, Z., Lei, Y.: Research on automatic instrumentation for bytecode testing and debugging. In: *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, vol. 1, pp. 268–274 (2012)
23. Yildiz, B.M., Rensink, A., Bockisch, C., Akşit, M.: A model-derivation framework for timing analysis of Java software systems. Technical report TR-CTIT-15-08, Centre for Telematics and Information Technology, University of Twente (2015)
24. Yildiz, B.M., Rensink, A., Bockisch, C., Aksit, M.: A model-derivation framework for software analysis. In: Hermanns, H., Höfner, P. (eds.) *Proceedings MARS. EPTCS*, vol. 244, pp. 217–229. Open Publishing Association (2017)

Graph Computation Models (GCM)

Graph Rewriting Based Search for Molecular Structures: Definitions, Algorithms, Hardness

Ernst Althaus, Andreas Hildebrandt, and Domenico Mosca^(✉)

Johannes Gutenberg-Universität Mainz, Staudingerweg 9, 55128 Mainz, Germany
{ernst.althaus, andreas.hildebrandt, mosca}@uni-mainz.de

Abstract. We define a graph rewriting system that is easily understandable by humans, but rich enough to allow very general queries to molecule databases. It is based on the substitution of a single node in a node- and edge-labeled graph by an arbitrary graph, explicitly assigning new endpoints to the edges incident to the replaced node. For these graph rewriting systems, we are interested in the subgraph-matching problem. We show that the problem is NP-complete, even on graphs that are stars. As a positive result, we give an algorithm which is polynomial if both rules and query graph have bounded degree and bounded cut size. We demonstrate that molecular graphs of practically relevant molecules in drug discovery conform with this property. The algorithm is not a fixed-parameter algorithm. Indeed, we show that the problem is W[1]-hard on trees with the degree as the parameter.

1 Introduction

Small molecules are of crucial importance in molecular biology. They serve various functions, e.g., as inhibitors or activators of proteins, as carriers of information, or energetic storage. Consequently, small molecules are a focus of numerous research fields (e.g., [2–4]). A prominent example is drug design, where small molecules are used to inhibit or activate proteins to achieve a desired biological function, or to prevent undesirable ones. Often, certain substructures of these small molecules are crucial for different aspects of their biological role: for example, a given molecular substructure might be favorable for an interaction with a given target, while another substructure may be responsible for toxic activity. Working with molecular substructures is hence an important task in computational chemistry, biology, and pharmacy. In fact, substructure representation is a core component of most applications in computational chemistry, including structure drawing, database search, and virtual screening [5].

Traditionally, substructures are modeled in chemical description languages such as Daylight’s SMARTS¹. These languages tend to be very complex and require significant training efforts before they can be routinely used. Furthermore, these languages are very restricted in their ability to describe patterns of typologies of the underlying graphs. This is relevant e.g. for searching certain

¹ <http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html>.

cyclic peptides. To make matters worse, parsing and matching such patterns against databases of molecules is NP-complete. Furthermore, these description languages often suffer from the lack of a formal definition of their semantics.

To circumvent these problems, we propose a simple graph rewriting system (or graph transformation) to describe substructures; experience shows that it is more intuitive to use methods to describe graphs directly compared to a string based description. Even graph rewriting systems of very simple form allow a high expressive power which almost reaches that of SMARTS and allow specification of some interesting substructures beyond the expressive power of SMARTS. Although the related search problem remains NP-complete, it becomes polynomial if each minimal cut of the query graph has bounded size, which we empirically find to be true for most molecules contained in the standard databases.

Using graph grammars instead of established molecular description languages not only allows a more intuitive representation of graph topology, but also enables queries that cannot be realized within the framework of, e.g., SMARTS and would typically have to be formulated as an additional external filter. One such example of practical relevance is the search for cyclic peptides. Many important drugs have such a structure, since they tend to resist digestion quite effectively.

Molecular function is rooted in molecular interactions. Such interactions typically do not require contributions from all atoms of the molecule, but rather involve specific groups of atoms. Hence, molecules that share certain molecular subgraphs can be expected to share some of their interactions, and hence, some of their function. We find that functional groups can be intuitively described by devising graph grammars that produce the molecular subgraphs belonging to this kind of group. Hence, matching such a graph grammar against a molecular database allows to query for instances that contain certain chemical groups and thus potentially show a desired chemical function.

A graph rewriting system is used to define a language of graphs, similarly to well-known string grammars. Therefore, graph rewriting systems are also known as ‘graph grammars’. A graph rewriting system consists of a set of rules defining how a graph can be transformed into another. The corresponding language is the set of all graphs that can be constructed from the rules starting with a graph consisting of a single node of a specific label. There are several known definitions of the transformation rules, such as node replacement, hyperedge replacement, and single- and double-pushout. We will relate our definition to those latter ones. Our definition is particularly simple and intuitive, as the graphs we are interested in are of bounded degree (atoms in molecules can form only a finite and typically very small number of covalent bonds) and thus we typically have rewriting systems of bounded degree. This allows a model where the rules explicitly reroute all edges that lose an endpoint due to the removal of a node of the graph. More precisely, we assume that the graphs are node- and edge-labeled multigraphs without self-loops. A rule replaces a single node with a specific label and specific labels of the incident edges by a subgraph. The edges originally

incident to the removed node are given new endpoints in the new subgraph. The labels of these edges may not be changed.

The paper is structured as follows. In the following sections, we give the formal definitions and review some related work. Before we state our algorithm for graphs with bounded size of minimal cuts in Sect. 5, we show how the rules of our graph grammar can be transformed in some easier form. In Sect. 6, we show that the problem is NP-complete even on stars. If we use the degree as parameter, the problem is W[1]-hard even on trees, as shown in Sect. 7. For graphs of degree 5, the problem is NP-complete, even if they have bounded pathwidth. The proof of this result is not given in this extended abstract. Finally, we give some experimental justification for the assumption that the graphs have bounded size of minimal cuts in Sect. 8 and conclude in Sect. 9.

2 Definitions and Summary of the Results

We now turn to the formal definitions. Let \mathcal{L}^V and \mathcal{L}^E be a set of node- and edge-labels. A node- and edge-labeled multi-graph over $(\mathcal{L}^V, \mathcal{L}^E)$ is a tuple $G = (V, E, N, L_V, L_E)$, where V and E are finite disjoint sets, $N : E \rightarrow V^{(2)}$ is assigning each edge its two endpoints, $L_V : V \rightarrow \mathcal{L}^V$ is assigning each node a label and $L_E : E \rightarrow \mathcal{L}^E$ assigns each edge a label. Notice that $V^{(2)}$ denotes the subsets of cardinality 2 of V . In the following, we always mean a node- and edge-labeled multi-graph when we talk about a graph.

For a graph $G = (V^G, E^G, N^G, L_V^G, L_E^G)$ and $U, U' \subseteq V^G$ with $U \cap U' = \emptyset$, let $\delta_G(U) = \{e \in E^G \mid \emptyset \neq U \cap N(e) \neq N(e)\}$, $\delta_G(U, U') = \{e \in E^G \mid U \cap N(e) \neq \emptyset \text{ and } U' \cap N(e) \neq \emptyset\}$ and let $G[U]$ be the induced subgraph of U . If the graph is clear from the context, we write $\delta(U)$ instead of $\delta_G(U)$ and furthermore, we write $\delta(v)$ for $\delta(\{v\})$. Let $\Delta(G)$ be the maximal degree of a node of G . A cut $(U, V \setminus U)$ is a partition of the nodes of a graph into two disjoint subsets, often referred to as the cut U . A cut U is minimal if there is no cut U' such that $\delta(U')$ is a proper subset of $\delta(U)$. It is well known that in a connected graph, a cut U is minimal if $G[U]$ and $G[V \setminus U]$ are connected.

A graph rewriting system in our definition is a tuple (S, P) , where $S \in \mathcal{L}^V$ is a label and P is a finite set of replacement rules of the form $(L, L_1, \dots, L_d) \rightarrow (G, n_1, \dots, n_d)$, where $L \in \mathcal{L}^V$, $L_1, \dots, L_d \in \mathcal{L}^E$, G is a graph and n_1, \dots, n_d are nodes of G . We call d the degree of the rule and G the replacement graph. For a graph rewriting system (S, P) , let $\Delta(S, P)$ be the maximal degree of any rule in P . Such a rule allows replacement of a node with label L whose incident edges have labels L_1, \dots, L_d by the graph G . The edges incident to the removed node are assigned new endpoints n_1, \dots, n_d in this order. More formally, given a graph $H = (V^H, E^H, N^H, L_V^H, L_E^H)$, a rule $(L, L_1, \dots, L_d) \rightarrow ((V^G, E^G, N^G, L_V^G, L_E^G), n_1, \dots, n_d)$ can be applied to a node $v \in V^H$, if there is a bijection $m : \{1, \dots, d\} \rightarrow \delta_H(v)$ such that $L^E(m(i)) = L_i$ for $1 \leq i \leq d$ and $L_V^H(v) = L$. If we apply this rule using m at v , the resulting graph is $H' = (V^{H'}, E^{H'}, N^{H'}, L_V^{H'}, L_E^{H'})$ with

- $V^{H'} = V^H \setminus \{v\} \cup V^G$, where the nodes of V^G are assumed to be disjoint from the nodes V^H (if they are not, we first make an isomorphic copy of G),
- $E^{H'} = E^H \setminus \{m(1), \dots, m(d)\} \cup E^G \cup \{e_1, \dots, e_d\}$, where e_1, \dots, e_d are new edges,
- $N^{H'}$ is defined by $N^{H'}(e) = N^H(e)$, if $e \in E^{H'} \cap E^H$, $N^{H'}(e) = N^G(e)$ if $e \in E^G$ and $N^{H'}(e_i) = \{n_i\} \cup (N^H(m(i)) \setminus \{v\})$ for $1 \leq i \leq d$,
- $L_V^{H'}$ is defined by $L_V^{H'}(v) = L_V^H(v)$ for $v \in V^{H'} \cap V^H$ and $L_V^{H'}(v) = L_V^G(v)$ for $v \in G$
- $L_E^{H'}$ is defined by $L_E^{H'}(e) = L_E^H(e)$ for $e \in E^{H'} \cap E^H$, $L_E^{H'}(e) = L_E^G(e)$ for $e \in E^G$, $L_E^{H'}(e_i) = L_E^H(m(i))$ for $1 \leq i \leq d$.

In the definition, we do not distinguish terminal and non-terminal node-labels. Nevertheless, in our examples there are node-labels that do not appear in the query graph and only those node-labels appear on the left hand side of the replacement rules. Hence these labels are in principal the non-terminals and drawn in boxes in the examples and all other labels are the terminals and drawn in circles. We refer to Fig. 1 for a pictorial description. In the following, we will sometimes additionally require that the graph G in a replacement rule is connected.

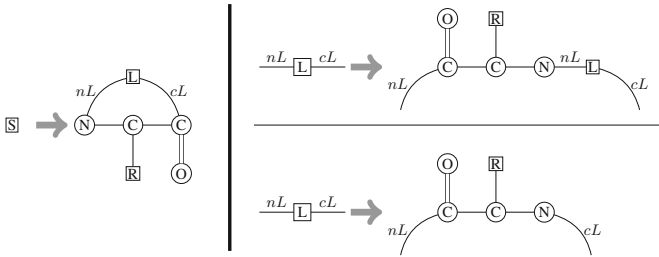


Fig. 1. A grammar for cyclic peptides: The starting label is S . The left-hand side shows the rule in which the starting graph can be replaced by an initial graph, where the label L stands for an arbitrary symbol chain of amino acids and R for a single amino acid side chain residue. With the upper rule, one can expand the chain by one further amino acid. After application of the lower rule, the chain can not be extended further. The edge labels are used to ensure that an N -node is always connected to a C -node and not to another N -node. An edge has label nL if it is between an N -node and an L -node on its creation and has label cL if it is between a C -node and an L -node. Whether there are rules that transform R into one of the amino acids or that transform R to a single node depends on whether it is necessary to describe that we have a peptide or not.

Let $\mathcal{G}(S, P)$ be the set of all graphs that can be obtained by iteratively applying rules starting from the graph consisting of a single node with label S and no edges. Given a graph G called a query graph and a graph rewriting system (S, P) , we are interested in the existence of a subgraph of G contained in $\mathcal{G}(S, P)$. Note that this query differs from the question typically addressed in the literature

on graph rewriting systems, where one is typically interested in whether G is contained in $\mathcal{G}(S, P)$. Another related question would be the existence of an induced subgraph of G in $\mathcal{G}(S, P)$. Our algorithm can be generalized to these questions easily.

We further note that we will use the edge-labels only to restrict the applicability of the rules to have a richer expressive power, while our input graphs typically have node-labels only. If we are given a graph G without edge-labels, we are interested whether there exists a labeling of the edges, such that the resulting graph has a subgraph contained in $\mathcal{G}(S, P)$. Our algorithm can easily be extended to this question.

Finally, it is important to note that the complexity of graph rewriting algorithms is usually investigated under the assumption that the graph rewriting system is fixed (and hence a polynomial algorithm can exponentially depend on the description size of the system), whereas we assume that the graph rewriting system is part of the input.

Notice that if the degree of a replacement graph of a rule is larger than the maximum of $\Delta(S, P)$ and $\Delta(G)$ for a query graph G , the corresponding rule can not be used to derive a subgraph of G , as a node of the replacement graph with degree larger than this maximum can neither be a node of the query graph nor replaced by any rule of the rewriting system. Hence, we will assume in the following that the degree of any replacement graph is at most $\max(\Delta(S, P), \Delta(G))$ if we apply our algorithm for query graph G .

We will show that the problem is NP-complete, even if the pattern graph (and hence all graphs in the rules) is a star. Let $\zeta(G)$ be the maximal size of a minimal cut of G . In other words, $\zeta(G)$ is the maximal cardinality of a cut $U \subseteq V$ such that $G[U]$ and $G[V \setminus U]$ are connected. For a graph rewriting system (S, P) let $\zeta(S, P)$ be the maximum value $\zeta(G)$ for a replacement graph G . We give a polynomial time algorithm solving the subgraph matching problem for a graph G and a graph rewriting system (S, P) for the special case where $\Delta(G)$, $\zeta(G)$, $\Delta(S, P)$ and $\zeta(S, P)$ are bounded. Notice that $\zeta(T) = 1$ for every tree T (and hence for every star). We sketch a proof for the fact that the problem is NP-complete even for graphs with bounded degree and bounded treewidth. Hence this well known parameter to describe the complexity of a graph cannot be used for our problem.

Our algorithm is not a fixed-parameter algorithm in the degree and the maximal size of a minimal cut. We show the problem is W[1]-hard even on trees with the degree as the parameter and hence, there is no such algorithm unless $W[1] = FPT$.

Notice that the algorithm of Lautemann [7] gives an algorithm for the graph matching problem if the degree of the input graph is bounded without the need to additionally bound another parameter. We show that the subgraph matching problem is NP-hard in this case and hence, it is unlikely that the algorithm can be generalized without increasing the running time. Nevertheless, our algorithm can be considered as a generalization of the algorithm of Lautemann.

Figure 2 gives a summary of our results.

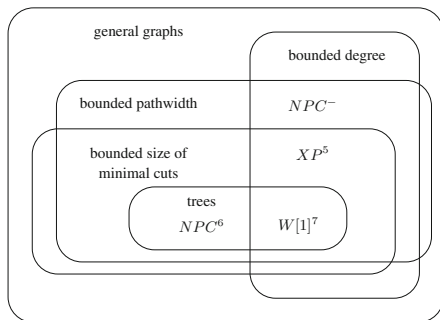


Fig. 2. Overview of our results for the subgraph matching problem. The algorithm which is polynomial-time if the size of the parameters are bounded (XP-algorithm), can clearly also be applied on more specialized classes, whereas the hardness results (NP-completeness (NPC) and $W[1]$ -hardness ($W[1]$)) generalize to larger graph classes. The superscript numbers in the results indicate the section of the proof; the hardness-proof for graphs with fixed degree and pathwidth is not given in this extended abstract. All hardness results except Sect. 6 also hold for a fixed set of rules, whereas the algorithm can be used if the rules are part of the input. For the graph matching problem, the algorithm of Lautemann [7] is an XP-algorithm for all bounded degree graphs and the respecting NP-hardness proof does not carry over. The other two hardness proofs also hold for the graph matching problem.

3 Related Work

There are many chemical description languages that are currently in use, such as the Sybyl Line Notation (SLN) [1] and the SMILES Arbitrary Target Specification (SMARTS), all of which describe molecular structures in the form of strings. Using graph rewriting systems to describe structures is more direct and hence more intuitive.

Graph rewriting systems have been studied since the 1960s. Three main approaches to formalize the basic idea to successively replace graphs by other graphs to define classes of graphs have been proposed. We relate our definition to the most important other definitions of graph rewriting systems, i.e. algebraic methods (single- and double-pushout-method), node-replacement grammars and hyperedge replacement grammars. The following definitions are sometimes abbreviated where the full details are not needed for this work. We refer to [9] for details.

In the double-pushout method, a rule of a graph rewriting system is described by two graphs $L = (V^L, E^L, N^L, L_V^L, L_E^L)$ and $R = (V^R, E^R, N^R, L_V^R, L_E^R)$ with a common set of nodes K (formally, we need a mapping of the nodes K to the nodes V^L and V^R respectively). To apply a rule, we look for an isomorphic

copy of L in G . We remove $L[V^L \setminus K]$ and all edges in $\delta_L(K)$. Then we insert $R[V^R \setminus K]$ and add the edges in $\delta_R(K)$ to the original nodes of K . Our rules can be interpreted as a restricted double-pushout method, in which L is a star, K are all nodes except the center of the star, and R is a connected graph. Furthermore each node of K has the same number of edges with the same labels in L and R . The double-pushout method has a higher expressive power than our type of graph grammar.

In a node replacement grammar, a single node u with a given label in a graph G is replaced by a graph D , as in our rule. The difference is how the embedding into the former neighborhood is done. Whereas in our case, the edges are explicitly assigned new endpoints, the embedding in node replacement grammars is done purely by the node labels. More specifically, a set of tuples of node-labels (μ, δ) is given and edges added between each node of label μ in G without u and each node of label δ in D .

The definition which is in some sense closest to ours are hyperedge replacement grammars. A hypergraph is a graph where N is not necessarily a set of cardinality two, but of arbitrary cardinality. A rule of a hyperedge replacement grammar specifies a label of a hyperedge e which is to be replaced by a hypergraph whose nodes are a superset of the nodes $N(e)$ incident to e . Notice that due to the fact that we remove an edge (and no nodes), the embedding is quite simple.

For a hypergraph H , its dual hypergraph is the graph which has a node for each hyperedge of H and a hyperedge for each node of H whose incident nodes are those corresponding to the hyperedges of H incident to the node. Our rules can be viewed as hyperedge replacements in the dual hypergraph with the additional condition that the degree of each node is exactly two (i.e. that the dual of the hypergraph is a graph).

Notice that we can consider our algorithm as an extension of the membership algorithm of Lautemann [7] to the subgraph matching problem. We will not elaborate on this due to space limitation.

4 Normal Form of the Rules

In order to simplify the description of the algorithm, we will assume next replacement graph consists of exactly two nodes (and an arbitrary number of edges between them). If all replacement graphs have this property, we call the rules in *normal form*. In the following, we show that general replacement rules as defined above can always be reduced to a set of rules in normal form. The maximal degree of a rule with replacement graph G increases at most by $\zeta(G) \cdot \Delta(G)$. As discussed before, we can bound $\Delta(G)$ by the maximum of the degrees of the rules and the query graph.

First, we argue that we do not need rules in which the new graph has exactly one node. Such rules can only change the label of the node. We can remove such rules exactly as in the case of context-free (string) grammars when constructing the Chomsky normal form.

Let $(L, L_1, \dots, L_d) \rightarrow ((V, E, N, L_V, L_E), n_1, \dots, n_d)$ be a rule in general form with $V = \{v_1, \dots, v_n\}$, $n \geq 3$ and $E = \{e_1, \dots, e_m\}$. We assume that the edges are ordered such that there is $1 \leq i \leq d$ with $n_1, \dots, n_i = v_1$ and $n_{i+1}, \dots, n_d \neq v_1$. If we only consider rules with connected graphs, we assume that the nodes are numbered by descending depth-first-search (DFS) completion numbers and let $\{v_1, \dots, v_n\}$ be the nodes in this order. This ensures that $G[\{v_2, \dots, v_n\}]$ is connected. We show how we can replace this rule by one rule in normal form and one rule where the replacement graph has $n - 1$ nodes. Iterating this replacement, we can ensure that all rules have the normal form.

The idea is that the first rule generated the node v_1 and a node with a new label ℓ which will be replaced by the remainder of the graph later. The number of edges between v_1 and ℓ corresponds to the number of edges of the graph between v_1 and the remainder of the graph, i.e. $|\delta(v_1)|$. In order to ensure that all edges get the correct endpoints, we introduce new edge-labels of all these edges.

More formally, we introduce a new node-label ℓ and new edge-labels $\ell_{e_1}, \dots, \ell_{e_n}$. In the first rule, the graph of the rule consists of the node v_1 with its label and a new node with label ℓ . This node ℓ will later be replaced by the graph $G[\{v_2, \dots, v_n\}]$. For each edge in $e \in \delta_E(v_1)$, we create an edge between the two nodes with label ℓ_e . The new endpoint for the edges e_1, \dots, e_i is v_1 and for the edges e_{i+1}, \dots, e_d it is the node with label ℓ .

The second rule will replace a node with label ℓ whose incident edges have labels $(L_{i+1}, \dots, L_d, (\ell_e)_{e \in \delta_G(v_1)})$ by the graph $G[\{v_2, \dots, v_n\}]$. The new endpoints of the edges with labels L_{i+1}, \dots, L_d are those of the original rule and the endpoints of the edges with labels ℓ_e for $e \in \delta_G(v_1)$ are the endpoints different from v_1 in G . See Fig. 3 for a pictorial description.

As the second rule will be the only rule for the node label ℓ , we can only apply the two rules in common and hence exactly replace the original rule.

We now argue that we can bound the degree of all rules created by the sum of the original degree of the rule and $\Delta(G) \cdot \zeta(G)$ for G being the replacement graph. We distinguish between the contribution of the edges defining the original

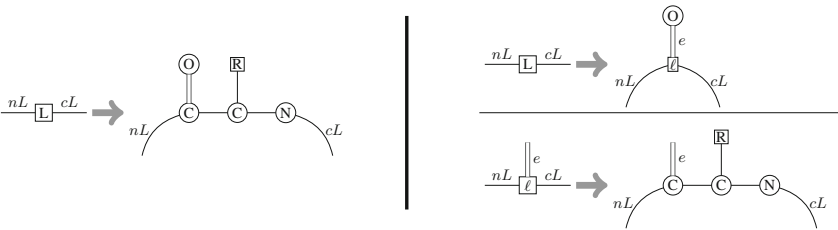


Fig. 3. The rule on the left-hand side is replaced by the two rules on the right-hand side. In the first rule, only the node with label O is created and a node with label ℓ is added as a placeholder for the remaining graph. The edge between these two nodes gets label e in order to ensure that it gets the correct endpoint in the end. In the second rule, the node with label ℓ is replaced by the remaining graph and the edges with labels nL , cL and e get the correct endpoints.

degree and the additional edges due to the replacement. The original edges were partitioned into the two created rules and hence their number can not increase in any rule. For the additional edges, we argue as follows: any constructed rule either creates a node of G or a subgraph $G[\{v_i, \dots, v_n\}]$. In the first case, the number of additional edges is bounded by the degree of the nodes. In the second case, the nodes $\{v_i, \dots, v_n\}$ are the nodes of a subtree of the DFS-tree that is cut at v_i , as we numbered the nodes according to descending DFS completion numbers and v_i is the root of the tree. We have to bound the number of edges of G with exactly one endpoint in v_i, \dots, v_n . Consider now the DFS-tree as an undirected tree rooted at v_i . The nodes of each subtree of v_i that is not contained in v_{i+1}, \dots, v_n form a minimal cut and contribute at most $\zeta(G)$ to the degree of the rule. Hence the degree of the rule is bounded by $\Delta(G) \cdot \zeta(G)$.

5 An Algorithm for the Subgraph Matching

We start with an exponential-time algorithm that simply enumerates backward substitutions in all possible ways until either the starting label is found or no further backward substitutions are possible. We store all subgraphs used to generate a label in a table in order to avoid multiple enumerations of the same structures. Afterwards, we characterize the graphs on which the algorithm runs in polynomial time. We assume that the rules are in normal form in the following.

If G can be obtained from the graph H by a single application of a rule, we say that H is a possible predecessor of G . Notice that if the rules and the input graph have bounded degree, given a graph G , we can simply enumerate all possible predecessors in polynomial time by iterating over all rules and trying to apply them backward on all pairs of nodes.

5.1 An Exponential Algorithm

Let (S, P) be a graph rewriting system and $G = (V, E, N, L_V, L_E)$ be a query graph. For $L \in L_V$ and $L_1, \dots, L_d \in L_E$, let $G(L, L_1, \dots, L_d)$ be the star with d edges with labels L_1, \dots, L_d , whose center has label L (the labels of the other nodes are not specified).

Our exponential algorithm stores the following dynamic programming table:

$$DP(L, U, e_1, \dots, e_d) \in \{\text{true}, \text{false}\},$$

where L is a node label, U is a subset of the nodes, d is the degree of a rule replacing node-label L , and e_1, \dots, e_d are edges of the graph with one endpoint in U , which is true, if it is possible to obtain a subgraph of $G[U]$ unioned with the edges e_1, \dots, e_d from $G(L, L_E(e_1), \dots, L_E(e_d))$. Notice we have derived from $G(L, L_E(e_1), \dots, L_E(e_d))$ and not from a subgraph of it. Furthermore, the labels of the endpoints of e_1, \dots, e_d that are not in U do not matter for this derivation. A subgraph of G is in $\mathcal{G}(S, P)$, if there is a table entry $DP(S, U, e_1, \dots, e_d)$ set to true for the starting symbol S and arbitrary U and e_1, \dots, e_d .

As our problem asks whether there is a subgraph of G in $\mathcal{G}(S, P)$, we only have to consider minimal subsets U , i.e. if $DP(L, U, e_1, \dots, e_d)$ is true, then $DP(L, U', e_1, \dots, e_d)$ should be true for each $U' \supseteq U$. We will not set these to true, as every entry of the table that can be set to true after some steps from $DP(L, U', e_1, \dots, e_d)$ can also be set to true from $DP(L, U, e_1, \dots, e_d)$.

In the beginning, we set $DP(L_V(v), \{v\}, e_1, \dots, e_i) = \text{true}$ for all nodes v , $1 \leq i \leq |\delta(v)|$, and e_1, \dots, e_i being a subset of cardinality i of the edges incident to v . All other values are set to false. Notice that, intuitively, we select the subgraph (V', E') by deriving true for the table entry with node set V' and the empty set of edges from the entries of DP for the node set $\{v\}$ and the edge set $\delta(v) \cap E'$ for all $v \in V'$.

Then we iterate over all pairs of entries $(L^1, U^1, e_1^1, \dots, e_{d_1}^1)$ and $(L^2, U^2, e_1^2, \dots, e_{d_2}^2)$ labeled true with $U^1 \cap U^2 = \emptyset$ and try to mark further entries with true by computing all possible predecessors when we apply an arbitrary rule $(L, L_1, \dots, L_d) \rightarrow (H, n_1, \dots, n_d)$ with $V^H = \{u_1, u_2\}$ and $H = (\{u_1, u_2\}, E_H)$ of our grammar. Let $\bar{E} = \{e_1^1, \dots, e_{d_1}^1\} \cap \{e_1^2, \dots, e_{d_2}^2\}$ and $E^i = \{e_1^i, \dots, e_{d_i}^i\} \setminus \bar{E}$. We can label $(L, U^1 \cup U^2, e_1, \dots, e_d)$ to true using such a rule, if

- the two nodes of H have labels L^1 and L^2 , respectively. Let u_i be the node with label L^i ($i = 1, 2$),
- there is a one-to-one correspondence $m : \bar{E} \rightarrow E_H$ of the edges in \bar{E} and the edges of H respecting such that $L_E(e) = L_{E^H}(m(e))$ for all edges $e \in \bar{E}$,
- there is a one-to-one correspondence $m' : E^1 \cup E^2 \rightarrow \{1, \dots, d\}$ between the edges $E^1 \cup E^2$ and the indices $1, \dots, d$ such that for an edge e of E^i , its label is $L_{m'(e)}$ and $n_{m'(e)} = u_i$,

Notice that if all rules have bounded degree d , the running time of the algorithm is polynomial in the size of the table, which is $|L^V| \cdot 2^{|V|} \cdot |E|^d$.

5.2 Reducing the Number of Considered Subsets

In the following, we only consider rules whose graphs are connected graphs. In this case, the sets U always induce connected subsets of the graph.

Let (L, U, e_1, \dots, e_d) be an entry with value true in the dynamic programming table DP . Remove the edges $\delta(U) \setminus \{e_1, \dots, e_d\}$ from G . If this graph has more than one connected component, we will not be able to apply a rule which contains nodes from different components. Hence, we can add all nodes not reachable from the component containing U to U . In the following, we restrict to entries $DP(L, U, e_1, \dots, e_d)$ such that U can not be enlarged in this manner.

If the input graph G is a tree, for each subset $\{e_1, \dots, e_d\}$ of edges there is at most one set U of nodes to be considered, namely the maximal subtree containing e_1, \dots, e_d as edges leading to leaves, if such a tree exists (with the exception that we have two possible sets, if the set of edges has cardinality 1). Hence our algorithm runs in polynomial time on trees if the degrees of the rules are bounded. In the next section, we show that the problem is NP-complete if the rules and the query graph can have arbitrary degree.

More generally, we can characterize a set of nodes U by the edges of its cut $\delta(U)$ (again, each set of edges can characterize two sets of nodes - the two sides of the cut). As we are only interested in subsets that can not be enlarged, we can characterize subset U uniquely by the edges $\{e_1, \dots, e_d\}$ plus the edges on $\delta(U)$ that are reachable in $G[V \setminus U]$ from one of the endpoints of $\{e_1, \dots, e_d\}$ that are not in U , where we only need to report one edge for parallel edges.

We now argue that we only have a polynomial number of subsets U to be considered if $\Delta(G) \cdot \zeta(G)$ becomes bounded by constants by showing that at most $\zeta(G) \cdot \Delta(G)$ edges are needed to describe the cut. Assume that we need more than $\zeta(G) \cdot \Delta(G)$ edges to describe the set U . This means that for at least one edge e_i , we have more than $\zeta(G)$ edges with different endpoints in $V \setminus U$ that are reachable from the endpoint u of e_i not in U in the graph $G[V \setminus U]$. Take the set U' of nodes reachable from u in $G[V \setminus U]$ as one set and U as the other set. Both are connected sets and we have $\delta(U, U') > \zeta(G)$, a contradiction.

6 NP-completeness for Rules of Unbounded Degree on Stars

In this section, we show that the graph and subgraph matching problem becomes NP-complete if we do not bound the degree of the rules and the query graph, even if the input graphs are stars. A star is a tree with only one internal node, called the center of the star.

The proof uses a reduction of 1in3SAT to our problem. Recall that in the 1in3SAT problem, we are given a formula $\phi = c_1 \wedge \dots \wedge c_m$ in conjunctive normal form such that each clause c_i consists of three literals, i.e. $c_i = l_1^i \vee l_2^i \vee l_3^i$ over a set $\{x_1, \dots, x_n\}$ of variables, i.e. $l_j^i = x_k$ or $l_j^i = \neg x_k$. We are interested whether there is a truth-assignment of the variables such that exactly one literal is true for each clause. This problem was proven to be NP-complete by Schaefer [10] and is already listed by Garey and Johnson [6] as NP-complete problem LO4.

Let $\phi = c_1 \wedge \dots \wedge c_m$ with $c_i = l_1^i \vee l_2^i \vee l_3^i$ be a formula over the variables $\{x_1, \dots, x_n\}$. The set of node-labels will be $\{c_1, \dots, c_m\} \cup \{l_j^i \mid 0 \leq i \leq n \text{ and } 0 \leq j \leq m\}$ and all edges will have the same label l_e . The query graph is the m -star whose center has label l_m^0 and the other nodes have labels c_1, \dots, c_m . The starting label is l_0^0 (see Fig. 5 for an example of the construction). The intuition is that the label l_0^0 can be replaced by two rules, one corresponding to setting x_1 to true, the other to setting x_1 to false. Applying the rule corresponding to setting $x_1 = \text{true}$ will add the neighbors c_i to the center for all clauses having literal x_1 . We will change the label of the center to the label l_j^1 , where j is used to count the number of edges incident to the center, i.e. the number of clauses that are satisfied by the truth-assignment in the first step. Such a label can be replaced by the rule corresponding to setting x_1 to a truth value. Hence the upper index of the label of the center indicates which variables already have a truth-value. The lower index is used to count the number of edges we already added to the star, i.e. if there are j clauses having literal x_i , the new label of the center is l_j^1 . If we derive the l_m^m , we created a star with m incident edges.

This graph matches the query graph if each label c_i was created exactly once and hence, for each clause exactly one literal was true.

Formally, we define the rules as follows. Let C_t^i be the indices of all clauses with literal x_i and C_f^i the indices of all clauses with literal $\neg x_i$. For each $0 \leq i < n$ and each $0 \leq j \leq m$, we have the following two rules:

- $(\underbrace{l_j^i, l_e, \dots, l_e}_{j\text{-times}}) \rightarrow (\underbrace{G, c, \dots, c}_{j\text{-times}})$, where G is the graph with center c having label $l_{j+|C_t^i|}^{i+1}$ and $|C_t^i|$ further nodes, one with label c_k for each $k \in C_t^i$. We call this rule $r_j^{i,t}$.
- $(\underbrace{l_j^i, l_e, \dots, l_e}_{j\text{-times}}) \rightarrow (\underbrace{G, c, \dots, c}_{j\text{-times}})$, where G is the graph with center c having label $l_{j+|C_f^i|}^{i+1}$ and $|C_f^i|$ further nodes, one with label c_k for each $k \in C_f^i$. We call this rule $r_j^{i,f}$.

The construction can clearly be performed in polynomial time. We now argue that the 1in3SAT-problem has a solution if and only if there is a subgraph of the input star that can be constructed by the graph rewriting system.

We first show that if the 1in3SAT problem has a solution then our problem has one. For this, let $\pi : \{x_1, \dots, x_n\} \rightarrow \{\text{true}, \text{false}\}$ be an assignment such that each clause has exactly one literal that is satisfied. In the following, we will identify true with t and false with f . Starting with l_0^0 , we use the rules $r_0^{0,\pi(x_1)}, r_1^{1,\pi(x_2)}, \dots, r_{\sum_{i=1}^{n-1} |C_{\pi(x_i)}^i|}^{n-1,\pi(x_n)}$ and get the input star at the end.

On the other hand, assume that a subgraph of the star can be constructed from l_0^0 using the rules. Notice that from the construction, it is clear that we

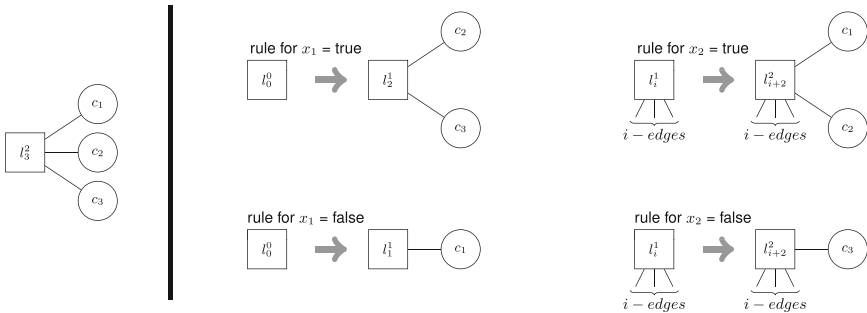


Fig. 4. The construction for the 2-SAT formula $c_1 \wedge c_2 \wedge c_3$ with $c_1 \equiv \neg x_1 \vee x_2$, $c_2 \equiv x_1 \vee x_2$ and $c_3 \equiv x_1 \vee \neg x_2$. The query graph is depicted on the left, the rules on the right hand side. With the rule corresponding to setting x_1 to true, we can add nodes with label c_2 and c_3 to the star and similarly for the other truth assignments. In this example it is not possible to select rules corresponding to the truth-value of x_1 and x_2 and constructing the star with labels c_1 , c_2 , and c_3 for the leaves and l_3^2 for the center.

always have a star whose center has a label l_j^i and only this node (and its incident edges) can be replaced. Hence, we will use exactly n rules, one for each variable, and can interpret the rules as a truth-assignment. Furthermore, the lower index of the label of the center will always be equal to the number of edges of the star. As the final label has to be l_m^n , we will have constructed the complete input star at the end (and not just a subgraph of it) and the truth-assignment is such that each clause has exactly one literal that is satisfied.

Notice that we use the lower indices of the labels l_j^i to count the number of nodes that are created to eventually ensure that the complete star is constructed. Hence, this construction can be used for the subgraph matching problem as well as for the graph matching problem. For the graph matching problem, the lower indices can be dropped (Fig. 4).

7 W[1]-Hardness in the Degree for Trees

In this section, we show that the graph matching problem is W[1]-hard in the degree even for trees. This is a strong hint that we will not find a FPT algorithm in the degree even for trees. We show this by a reduction from the longest common subsequence problem, which is shown to be W[1]-hard in the number of sequences, even for fixed sized alphabets, in [8].

Recall that in the longest common subsequence problem, we are given d sequences s_1, \dots, s_d over an alphabet Σ and a number k . The question is whether there is a sequence s of length k which is a subsequence of each of the d strings s_1, \dots, s_d .

We have labels $\{c, t, h, e\} \cup \Sigma$. We construct a tree and rules as follows. The tree consists of a center node with label c , from which d paths evolve with labels corresponding to the strings, terminated by a node with label e and a further path of k nodes of label h followed by one of label t . See Fig. 5 for a sketch of the construction.

We have rules that, when considered in a backward direction, allow to shrinking of a node with a label from Σ into the center and rules which shrink a star of d nodes of the same label from Σ and a node with label h to the center. Finally, we have the terminating rule which shrinks the center with d nodes of label e and a node of label t to the starting symbol.

We now argue that the query graph can be constructed if there is a common substring of length k . We do it again by showing how the query graph can be reduced to the starting label by backward substitutions. Starting from the query graph, we shrink all nodes into the center that are not part of the common substring of length k until the first remaining node of each string is of the common substring (in arbitrary order). Then we shrink the d nodes corresponding to the first letter of the common substring and one node of label h to the center. We continue this way until all nodes of the strings but the last ones are shrunken into the center. Finally, we can use the rule that creates the starting label from this graph.

Finally, we argue that there is a common substring of length k if a (sub)graph of the query graph can be constructed. Here, we use the forward direction when

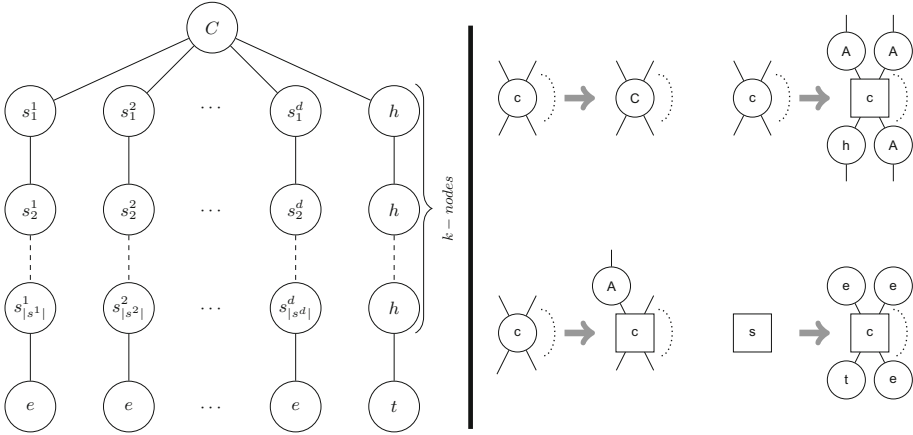


Fig. 5. The sketch of the input graph on the left and the rules on the right. The rules have to be created for all $A \in \Sigma$. The rule at the top left replaces an arbitrary string by an arbitrary symbol. The rule at the bottom left extends an arbitrary string by an arbitrary symbol. The rule at the top right extends all strings by the same symbol and the counter for the number of matching symbols increases by one. The rule at the bottom right generates the starting configuration.

arguing. We can only start with the rule that creates the star with d labels e and one label t for the leaves and label c for the center. By the construction, it is only possible to emerge new nodes from the node with label c and there will be only one such node. Hence, we can only match the query graph and not a subgraph of it and we have to create exactly the given strings, either by adding a single letter to a string or by adding the same letter to all strings and one node of label h . Notice that we have to add the node with label h always to the part terminated by t , as otherwise we will not be able to construct (a subgraph of) the query graph. During the creation of the strings, we have to create k nodes of label h and hence, the corresponding substrings build a common substring of length k .

Notice that the rules only depend on d and the alphabet Σ , i.e. the hardness result holds even for a fixed graph grammar. Furthermore, we do not need edge-labels in the normal form of the rules by giving each string a unique copy of the alphabet.

The construction can be used to show the hardness of the graph matching problem and the subgraph matching problem.

8 Estimating the Size of the DP-Table in a Large Database

We showed in Sect. 5.2 that the running time becomes polynomial for a class of graphs with bounded ζ -value. More precisely, we can estimate the size of the

dynamic programming table for a graph $G = (V, E)$ as $|\mathcal{L}^V| \cdot |E|^{\Delta(G) \cdot \zeta(G)}$. We calculated this value over all purchasable structures in the Zinc-database² and report on the statistics in this section. The database contains about 22 million structures and is often scanned through in computational drug discovery. For the vast majority of structures, the parameter value is at most 6, but can become as large as 22. In Table 1, we give the histogram of the distribution of this value.

Table 1. Histogram of $\zeta(G)$ over the all purchasable structures G in the Zinc-Database.

Number	1	2	3	4	5	6	7	8	9	10
ζ	105329	13429106	4027109	3450440	541304	308973	62943	25141	5649	1698

Number	11	12	13	14	15	16	17	18	19	20	21	22
ζ	518	268	152	72	29	35	11	10	1	2	0	1

Furthermore, we report on the number of subsets U of the nodes such that $G[U]$ and $G[V \setminus U]$ are connected. If s is this number, the size of the dynamic programming table can also be bounded by $|\mathcal{L}^V| s^{\Delta(G)}$ and hence, if it is polynomially bounded in the input size and the maximum degree is bounded as well, we obtain a polynomial algorithm, too. For most of the instances, this value is between 16 and 2048, only about 2‰ have a larger value, and only 209 instances have a value exceeding 100000. Two instances have more than 2 million components. See Fig. 6 for a plot of the data. Notice that it is a worst-case estimation and in practice the dynamic programming table will be much smaller.

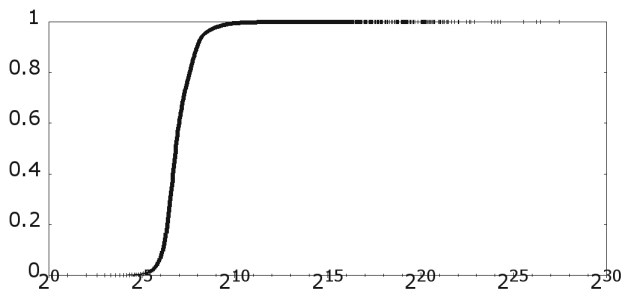


Fig. 6. The graph shows the percentage (y -axis) of the structures G for which the number of subsets U of nodes such that $G[U]$ and $G[V \setminus U]$ are connected is below the value on the x -axis. For more than 2‰ of the structures the value lies between 16 and 2048.

The results clearly indicate that these numbers are generally small in this database and hence, we are confident that an implementation of this algorithm will be very efficient.

² <http://zinc.docking.org/>.

On the other hand, there are known structures, such as fullerenes, where the ζ -value becomes very large and hence, the algorithm proposed will not be efficient. These structures typically are difficult as well for other query languages such as SMARTS, which often rely on precomputation of properties such as the smallest set of smallest rings [11] (known as the minimum cycle basis in computer science).

In the future, we will evaluate these values over the database pubchem³ consisting of more than 120 million structures.

9 Conclusion

In this work, we present an algorithm for graph rewriting-based molecular structure search that is polynomial if the degree of the rules of the graph rewriting system and the cut size between each pair of connected components are bounded. We further showed empirically that this assumption is reasonable.

Our algorithm is a generalization of the graph-matching algorithm of Lautemann for hyperedge replacement grammars. It only works on hypergraphs whose duals are graphs, i.e. every node in the hypergraph has exactly two incident edges at any time during the derivation. We will investigate whether it is possible to extend our algorithm to general hypergraphs.

The algorithm is not a fixed-parameter algorithm and we showed that the problem is W[1]-hard even on trees. Nevertheless, it is possible that there is an algorithm that is exponential in the degree, but polynomial in the maximal size of a minimal cut or a related graph parameter. We will investigate this further in future work.

Furthermore, we will implement our algorithm. In order to become efficient enough to scan databases, suitable algorithm-engineering techniques have to be developed and applied.

References

1. Ash, S., Cline, M.A., Homer, R.W., Hurst, T., Smith, G.B.: SYBYL line notation (SLN): a versatile language for chemical structure representation. *J. Chem. Inf. Comput. Sci.* **37**(1), 71–79 (1997)
2. Dehof, A.K., Lenhof, H.P., Hildebrandt, A.: Predicting protein NMR chemical shifts in the presence of ligands and ions using force field-based features. In: *Proceedings of German Conference on Bioinformatics 2011* (2011)
3. Dehof, A.K., Rurainski, A., Bui, Q.B.A., Böcker, S., Lenhof, H.-P., Hildebrandt, A.: Automated bond order assignment as an optimization problem. *Bioinformatics* **27**(5), 619–625 (2011)
4. Dietzen, M., Zotenko, E., Hildebrandt, A., Lengauer, T.: Correction to on the applicability of elastic network normal modes in small-molecule docking. *J. Chem. Inf. Model.* **54**(12), 3453 (2014)

³ <http://pubchem.ncbi.nlm.nih.gov>.

5. Ehrlich, H.-C., Rarey, M.: Systematic benchmark of substructure search in molecular graphs - from Ullmann to VF2. *J. Cheminform.* **4**, 13 (2012)
6. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York (1979)
7. Lautemann, C.: The complexity of graph languages generated by hyperedge replacement. *Acta Inf.* **27**(5), 399–421 (1990)
8. Pietrzak, K.: On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. *J. Comput. Syst. Sci.* **67**(4), 757–771 (2003)
9. Rozenberg, G. (ed.): *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*, vol. 1. World Scientific, Singapore (1997)
10. Schaefer, T.J.: The complexity of satisfiability problems. In: Lipton, R.J., Burkhard, W.A., Savitch, W.J., Friedman, E.P., Aho, A.V. (eds.) *Proceedings of 10th Annual ACM Symposium on Theory of Computing*, 1–3 May 1978, San Diego, California, USA, pp. 216–226. ACM (1978)
11. Zamora, A.: An algorithm for finding the smallest set of smallest rings. *J. Chem. Inf. Comput. Sci.* **16**, 40–43 (1976)

Towards Automatic Generation of Evolution Rules for Model-Driven Optimisation

Alexandru Burdusel^(✉) and Steffen Zschaler

Department of Informatics, King's College London, London WC2B 4BG, UK
alexandru.burdusel@kcl.ac.uk, szschaler@acm.org

Abstract. Over recent years, optimisation and evolutionary search have seen substantial interest in the MDE research community. Many of these techniques require the specification of an optimisation problem to include a set of model transformations for deriving new solution candidates from existing ones. For some problems—for example, planning problems, where the domain only allows specific actions to be taken—this is an appropriate form of problem specification. However, for many optimisation problems there is no such domain constraint. In these cases providing the transformation rules over-specifies the problem. The choice of rules has a substantial impact on the efficiency of the search, and may even cause the search to get stuck in local optima.

In this paper, we propose a new approach to specifying optimisation problems in an MDE context *without the need to explicitly specify evolution rules*. Instead, we demonstrate how these rules can be automatically generated from a problem description that consists of a meta-model for problems and candidate solutions, a list of meta-classes, instances of which describe potential solutions, a set of additional multiplicity constraints to be satisfied by candidate solutions, and a number of objective functions. We show that rules generated in this way lead to optimisation runs that are at least as efficient as those using hand-written rules.

1 Introduction

There has been a good deal of interest in optimisation of models in recent years [1–8]. These approaches aim to provide support for search-based software engineering [9] in an MDE context. Many of these approaches focus on using evolutionary techniques for finding models that optimise some objective function(s)—for example an OCL query or a simulation-based evaluation. To guide the exploration of the search space, a user has to provide a set of model transformations, which can create new candidate solution models from existing ones. Overall, the optimisation problem is thus specified by providing (1) a meta-model, instances of which are candidate solutions, (2) a set of objective functions, (3) additional constraints to be satisfied by valid solutions, and (4) a set of transformations to evolve models.

In some cases, these transformations are an inherent part of the optimisation problem. For example, when using evolutionary search to find an optimal refactoring of model transformations [10], or when finding optimal reconfigurations

of a cloud data centre [11], it is important to ensure that any solutions have been derived from the starting point only through the application of rules from a pre-defined set. In the latter case, we may even have an objective function based on the number of transformation steps that have been applied. However, in many other scenarios specifying the transformation rules as part of the optimisation problem is less natural and leads to over-specification. For example, the well-known class-responsibility assignment (CRA) problem, which was also a problem case at the 2016 Transformation Tool Contest (TTC) [12], simply looks for an optimal allocation of features to classes. How the search algorithm arrives at this allocation is not a natural part of the problem. In fact, when solving this problem in an evolutionary manner, there are different sets of evolution rules that might potentially be applied, and different sets of rules will lead to results of different optimality. Requiring users to specify the rules with the problem, then, forces them to over-specify and risks missing the best solutions.

In this paper, we show how optimisation problems over models can be specified *without the need to specify a set of evolution transformations as well*. We show how a set of rules can be automatically generated from a meta-model, objective functions, and a set of additional constraints. The rule generation algorithm presented in this paper uses an extended variant of the SERGe (SiDiff Edit Rule Generator) algorithm presented in [13]. We demonstrate, using the CRA case study, that we are able to generate rules that enable efficient optimisation runs leading to good results.

The remainder of this paper is structured as follows: Sect. 2 gives a brief description of related work in model optimisation. Then, in Sect. 3, we describe the case study used throughout the paper, followed by Sect. 4, where we present our solution. In Sect. 5, we present an evaluation, including a comparison to the VIATRA-DSE solution to the TTC '16 CRA case [14]. Finally, in Sect. 6, we discuss lessons learned and highlight future research.

2 Related Work

We have introduced MDEOPTIMISER (MDEO) previously in [2, 15]. MDEO performs model-based optimisation by running evolutionary optimisation with candidate solutions represented by model instances of a given meta-model. Evolution steps are obtained by applying endogenous model transformations using Henshin transformation rules [16]. Since our submission to TTC 2016, the tool has been improved, the evaluation in Sect. 5 is based on the most recent version of the tool. A description of the improvements is included in Sect. 4.

In [5] the authors introduce the MOMoT (Marrying Optimisation and Model Transformations) tool. The tool is built in the context of Eclipse Modelling Framework (EMF)¹ and it uses Henshin transformation rules to generate optimisation solutions. The tool uses the MOEA framework² for the implementation of the search algorithms. Alongside the MOEA framework algorithms, MOMoT

¹ <https://eclipse.org/modeling/emf/>.

² <http://moeaframework.org/>.

also supports single-objective and local search optimisation algorithms. It defines a custom DSL for problem descriptions, consisting of a meta-model, a set of Henshin transformation rules, a set of objectives and constraints specified either as Java or OCL implementations and the search algorithm to be used. The output produced consists of a set of analysis artifacts, the resulting models, found objective values and a chain of rule applications used to obtain the solution models. The MOMoT framework is very similar to MDEO, the main difference being that MDEO runs the optimisation directly on models rather than on sequences of rule applications from which models can be generated.

VIATRA-DSE [1] is another tool performing optimisation on models. It uses the VIATRA2 [17] model transformation framework which is built on the EMF. In order to run model optimisation, the tool requires as input an initial model, a set of transformation rules, a set of constraints and a set of objectives. For search space exploration the tool supports several algorithms such as Hill Climbing and Non-dominated Sorting Genetic Algorithm (NSGA-II) [18]. This tool, similarly to MOMoT requires the user to specify the transformation rules as part of the optimisation problem specification.

In [4], the authors propose another model optimisation tool. Crepe Complète is an extension of Crepe [19], which allows multi-objective optimisation of models. It has been developed as an improvement of the Crepe tool which only supported single objective optimisation. Crepe Complète is built on top of the Epsilon Object Language (EOL) and can run multi-objective optimisation on models. Crepe Complète can run optimisation on any problem that can be encoded in a meta-model. The tool supports generic search operators and a generic encoding of models using integer vectors. However, optimisation performance can quickly become sub-optimal as the encoding is non-locality-preserving [20].

There remains a clear gap for approaches that run evolutionary search over models but do not require manual definition of evolution rules as part of the problem specification. In this paper, we propose a first such approach.

3 Running Example

Throughout this paper, we will use a running example to help explain and evaluate our approach. For this, we are reusing the well-studied Class-Responsibility Assignment (CRA) problem, in the form introduced as a challenge case at the 2016 Transformation Tool Contest [12].³ The goal in this problem is to find an optimal set of classes and class-feature allocations that minimise coupling and maximise coherence.

More specifically, a CRA problem is an instance of the CRA meta-model in Fig. 1, without any instances of `Class`. A valid solution is an instance of the same

³ This problem case also required all classes to have unique names. Given that this can be achieved by a simple post-processing step [15], we ignore the requirement for this paper.

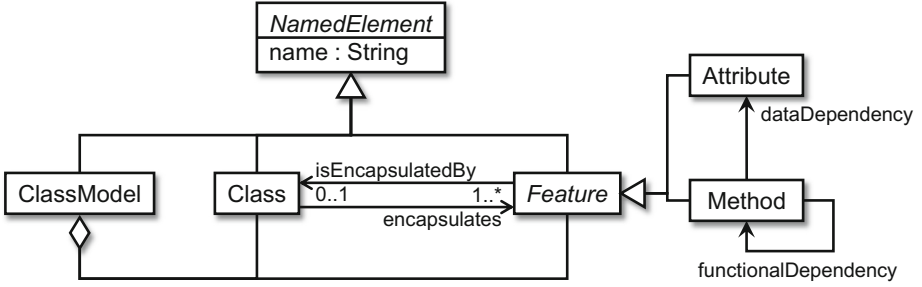


Fig. 1. Problem meta-model for the CRA problem

meta-model with a number of **Class** instances and *all* **Feature** instances allocated to a class via association `isEncapsulatedBy`. Note that this means that for solutions we have a stricter multiplicity constraint than for problem descriptions; in particular, the lower bound multiplicity of `isEncapsulatedBy` is 1 rather than 0. **Features** can be either **Attributes** or **Methods**. Different kinds of dependencies can be represented between features of different kinds. The goal is to allocate **Features** to (newly created) **Classes** to minimise dependencies between classes (coupling) and maximise dependencies within classes (coherence). This is specified as a single objective combining the coupling and coherence objectives into a single so-called CRA measure. Details of the definition of this measure can be found in the TTC case [12]. The case study description includes five input models which can be used to evaluate the proposed case solutions [12]. A summary of these input models, which vary in size and complexity, has been included in Table 1. In Sect. 5 we are discussing the results of our approach using each of these models as input.

This is a common way in which search problems are phrased: an initial model (instance of the problem meta-model) is given to describe a specific search problem. Typically, the elements provided in this model are not meant to be changed as a result of the search. Some elements of the meta-model are not (or only partly) instantiated: these will be used to represent potential solutions. Because the problem meta-model, thus, needs to be a valid meta-model for both problem specifications and candidate solutions, some of its multiplicity constraints (namely those at the boundary between the elements responsible for problem descriptions and those responsible for solution descriptions) may need to be strengthened for valid solutions. This is similar to more traditional ways of specifying optimisation problems, where constraints are a standard part of a problem specification. In model-based optimisation, we can say that the *problem meta-model* is refined by the *solution meta-model* by providing stronger multiplicity bounds in some places. Thus, every candidate solution will also be a valid instance of

Table 1. Summary of input models

	A	B	C	D	E
Attributes	5	10	20	40	80
Methods	4	8	15	40	80
Data Dep.	8	15	50	150	300
Functional Dep.	6	15	50	150	300

the problem meta-model, while a problem specification will typically not yet be a valid instance of the solution meta-model. The task of the search algorithm, then, is to continuously modify the given problem specification until it satisfies all the additional multiplicity constraints of the solution meta-model and, then, to find an optimal solution model.

4 Searching Optimal Models with Generated Rules

We have implemented our approach using our MDEO tool [2, 15]. The tool is an Eclipse plugin allowing the user to specify model optimisation in the EMF context through a DSL. It uses Henshin-encoded endogenous model transformation rules as search mutation operators, to explore the search space. The optimisation algorithms are implemented using the MOEA framework. In earlier versions of the tool, the user was required to manually create the Henshin transformation rules and then specify them in the DSL configuration. In this paper we are using the SERGe [13] meta-tool to automatically generate the initial consistency preserving edit rules (CPERS). For each of the generated rules we then make a copy to which we apply a set of refinements to better guide the evolutionary process by ensuring that edit operations encoded in the rules can be applied to models conforming to both the problem and the solution meta-models.

The rest of this section is structured as follows: In Sect. 4.1 we describe how the optimisation problem for rule generation is specified in our DSL, then in Sect. 4.2 we describe our rule generation algorithm. Section 4.3 describes how we configured our tool to run the experiments for the CRA case.

4.1 Specifying the Optimisation Problem

The problem description required by our DSL consists of the following elements:

1. *A problem meta-model.* Specific search problems are given as instances of this meta-model. In the CRA case this is the meta-model shown in Fig. 1;
2. *Objective functions.* These can be provided as Java implementation or as OCL queries and return a numerical value for a given model. In the context of the CRA case, we have only one objective, namely the CRA value which we are seeking to maximise during the search;
3. *A meta-model subgraph.* Only a subset of the elements from the problem meta-model represent solution information. Instances of these elements can be modified during the search, everything else should be kept constant as it represents problem context only. The *multiplicity refinements* provided next can only apply to elements in this sub-graph of the problem meta-model;
4. *Additional multiplicity constraints.* These constraints, which we also call *multiplicity refinements* are constraints that form the *solution meta-model*, a subset of the problem meta-model. These refinements must be satisfied by all valid solution candidates. In the CRA case, as described in Sect. 3, the requirement is that there are no features which are not encapsulated in a

class, so a refinement is to restrict the multiplicity of the `isEncapsulatedBy` edge from `[0..1]` to `[1..1]`. These multiplicity constraints must refine those in the original problem meta-model;

5. *Constraint functions.* These represent the additional multiplicity constraints in a form that can be used by a search algorithm (*i.e.*, a function that must be zero for valid candidate solutions). In principle, these could be generated from the additional multiplicity constraints, but our prototype currently does not support this;
6. *An optimisation algorithm.* This specifies the algorithm provider⁴, along with the search algorithm to use and the necessary evolutions and population configuration. In this paper we are only using the NSGA-II algorithm with multiple configurations for the evolutions and populations variables. It is beyond the scope of this paper to present a comprehensive comparison between multiple algorithms.

An example of the CRA problem specification using the MDEO DSL can be seen in Fig. 2. The configuration keywords in the DSL are intuitive. The **basepath** element is required, can only be used once and it defines the Eclipse resource set working path, then the **meta-model** element is also required, can be only one and it's used for specifying this optimisation problem meta-model. The next element is the **objective**, which is required and can be used multiple times. This is used to specify the optimisation objectives which can be loaded from Java files or specified as OCL queries. The next keyword is **constraint**, it's optional and it defines the constraints to be used in the optimisation process. Then the **rule generation node** keywords allow the user to specify the nodes for which Henshin transformation rules are going to be generated. Finally the **optimisation** keyword is used to configure the search algorithm and its parameters.

```

1 basepath <src/uk/ac/kcl/mdeoptimiser/gcm2017/models/>
2 metamodel <architectureCRA.ecore>
3 objective CRA maximise java { "uk.ac.kcl.mdeoptimiser.gcm2017.MaximiseCRA" }
4 constraint MinimiseClasslessFeatures java {
5   "uk.ac.kcl.mdeoptimiser.gcm2017.MinimiseClasslessFeatures" }
6 rule generation nodes { "Class" }
7 refined multiplicity node "Feature" edge "isEncapsulatedBy" lower "1" upper "1"
8 optimisation provider moea algorithm NSGAI evolutions 10000 population 50

```

Fig. 2. MDEO CRA problem specification with automatic rule generation

4.2 Generating the Rules

In this section, we discuss how we generate the evolution transformation rules from the information provided above.

⁴ As registered in the underlying instance of the MOEA Framework.

Previous work on automatic generation of transformation rules from meta-model information has been reported in [13]. SERGe is a meta-tool which generates a complete set of complete and consistency-preserving edit operations (CPEOs) for a given meta-model. The tool has been developed in the EMF context and the generated transformation rules are encoded in Henshin. The algorithm implemented in SERGe is designed to ensure that for any rules it generates, any change between two models is always consistency preserving with regards to the meta-model. The SERGe tool, by default, provides an extensive set of configuration options. A complete description of the rule generation process supported is described in [21]. However for the purpose of proving our approach with the CRA case we have restricted the generation of rules to only a subset of all the possible operations. A complete list of the SERGe rules generated for our case can be seen in Table 2. To generate the rules, SERGe performs the following steps:

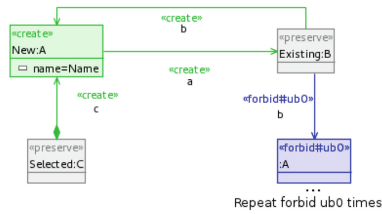
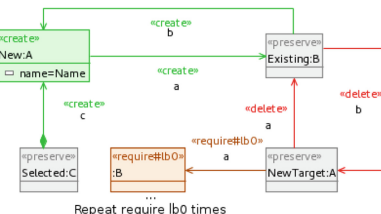
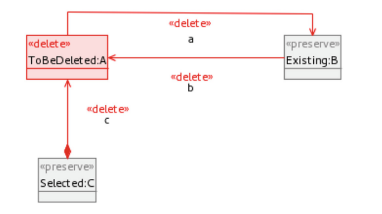
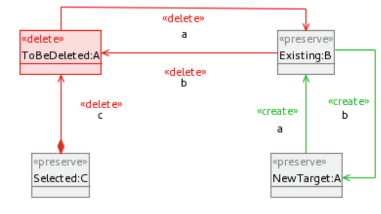
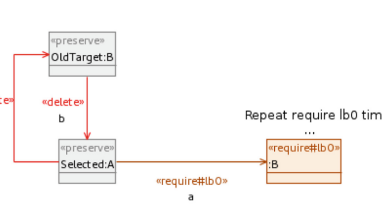
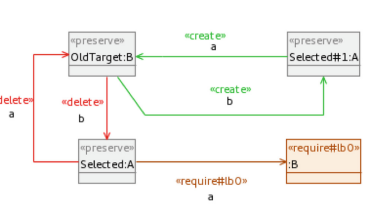
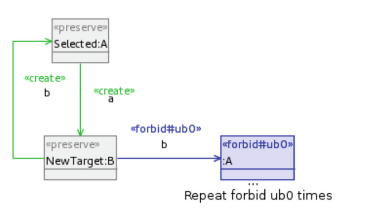
1. *Create node.* For each non-root type A with mandatory neighbours and no children, SERGe will generate a create node rule. The rule also connects the node A to its mandatory children, if any are present in the metamodel, by creating a containment edge. The node is connected to its mandatory neighbours B by creating edges of type a . If edge a has an opposite edge of type b then this will also be created. If there is an upper bound multiplicity in the meta-model for edge b , then a Negative Application Condition (NAC) will be generated to ensure that the multiplicity is respected when connecting nodes of type B to nodes of type A ;
2. *Delete node.* The node delete rules are created as inverses of the Create rules, by swapping the left and right-hand side of the Henshin rule graph. The generated rule will delete the node A and all its mandatory children. The node is also disconnected from its mandatory neighbours. If multiplicity constraints are present, then a Positive Application Condition (PAC) will be generated to ensure that node A can be deleted safely without invalidating the meta-model lower bound multiplicities between the deleted node and its neighbours;
3. *Add node edge.* The add edge rule is generated for each reference a or opposing reference b of a node A . If the reference is not a containment and if the lower bound multiplicity is not equal to the upper bound multiplicity then, a rule is generated to add an edge a between type A and the type at the opposing end B . If edge a has an opposite edge of type b then this will also be created. If there is an upper bound multiplicity in the meta-model for edge b , then a NAC will also be generated to ensure that the multiplicity is respected when connecting nodes of type B to nodes of type A ;
4. *Remove node edge.* Similar to the delete node rule the remove node edge rule is generated by swapping the left-hand side with the right-hand side of the add edge rule. In the case of remove edge rules, NAC applications are not required, but if multiplicity constraints are present, then a PAC will be generated to ensure that the edge can be deleted safely without invalidating the model;

5. *Change node edge*. This rule type is a simplified version of the combined application chain of a *Remove node edge* and *Create node edge* rules, performing the individual steps of each of these rules in a single application. This rule is generated by SERGe when ran with a meta-model that has a fixed multiplicity between two nodes. The generated rule is the same as the *Refined Remove Edge Rule* in Table 2.

When using the transformation rules generated by SERGe for the problem meta-model, the optimisation process has a tendency to get stuck in local optima. This is because the SERGe rules are generated so that the produced models are consistent w.r.t to the problem meta-model only. The solution meta-model is a subset of the problem meta-model, as a result of the refined multiplicity constraints applied to the problem meta-model. It is still possible for solution meta-model instances to be discovered using the rules generated by SERGe for the problem meta-model, however when using these rules to transform instances of the solution meta-model, the validity of the resulting model instances cannot be ensured. The validity depends on which part of the problem meta-model the transformation output model conforms to, the valid solution meta-model subset which satisfies the problem constraints or the rest of the problem meta-model which includes all possible solutions, both valid and invalid w.r.t. the problem constraints. During the optimisation process, if a valid solution becomes invalid because of a constraint invalidation, it automatically becomes infeasible and it is dominated by other valid solutions [18, 22]. This happens even if a subsequent transformation on the same solution would make it dominant. A solution is dominant if it is feasible with regards to its constraints and it is at least as good for all objective values as the other solutions and better for at least one objective value [22].

Performing a CRA case optimisation run using these transformation rules generated for the problem meta-model and using a constraint that invalidates solutions with unassigned **Features**, is not sufficient to obtain the best possible results. The evolution gets stuck in local optima after all the **Features** are assigned to a **Class** and the model becomes consistent w.r.t the solution meta-model. Then, the only way to move a **Feature** from a **Class** to another **Class**, is to remove the `isEncapsulatedBy` edge from a **Class** using rule *REMOVE_Class_(encapsulates)_TGT_Feature* and then add it again for another **Class** using rule *ADD_Class_(encapsulates)_TGT_Feature*. While this is done, the specified constraint is invalidated, creating a solution which has one unassigned **Feature**. This constraint violation causes the solution to become infeasible and it becomes dominated by the other solutions which do not have a good CRA value to that point, but are feasible because they don't invalidate the constraint to have no unassigned **Features**. As a result, the new candidate is removed from the population and never explored further. We could try to fix this by encoding constraints as objectives instead. However, while this would allow the search to escape local optima, by not having any solutions considered invalid, it would not guarantee all resulting search solutions to be valid.

Table 2. Generated SERGe and refined rules

SERGe Create Rule	Refined Create Rule
 <p>The diagram shows a flow from a green box 'New:A' (with '«create»' and 'name=Name') to a grey box 'Existing:B' (with '«preserve»'). From 'Existing:B', a blue arrow labeled '«forbid#ub0»' points to a blue box ':A' (with '«forbid#ub0»'). A green arrow labeled '«create»' points from 'New:A' to a grey box 'Selected:C' (with '«preserve»'). Labels 'a', 'b', and 'c' are placed near the arrows. Below the diagram, it says 'Repeat forbid ub0 times'.</p>	 <p>The diagram shows a flow from a green box 'New:A' (with '«create»' and 'name=Name') to a grey box 'Existing:B' (with '«preserve»'). From 'Existing:B', a red arrow labeled '«delete»' points to a grey box 'NewTarget:A' (with '«preserve»'). From 'NewTarget:A', a red arrow labeled '«delete»' points to 'Existing:B'. A green arrow labeled '«create»' points from 'New:A' to a grey box 'Selected:C' (with '«preserve»'). An orange box ':B' (with '«require#lb0»') has a green arrow labeled '«create»' pointing to 'NewTarget:A' and a red arrow labeled '«delete»' pointing to 'Existing:B'. Labels 'a', 'b', and 'c' are placed near the arrows. Below the diagram, it says 'Repeat require lb0 times'.</p>
SERGe Delete Rule	Refined Delete Rule
 <p>The diagram shows a flow from a grey box 'Selected:C' (with '«preserve»') to a red box 'ToBeDeleted:A' (with '«delete»'). From 'ToBeDeleted:A', a red arrow labeled '«delete»' points to a grey box 'Existing:B' (with '«preserve»'). Labels 'a', 'b', and 'c' are placed near the arrows.</p>	 <p>The diagram shows a flow from a grey box 'Selected:C' (with '«preserve»') to a red box 'ToBeDeleted:A' (with '«delete»'). From 'ToBeDeleted:A', a red arrow labeled '«delete»' points to a grey box 'Existing:B' (with '«preserve»'). From 'Existing:B', a green arrow labeled '«create»' points to a grey box 'NewTarget:A' (with '«preserve»'). From 'NewTarget:A', a green arrow labeled '«create»' points to 'Existing:B'. Labels 'a', 'b', and 'c' are placed near the arrows.</p>
SERGe Remove Edge Rule	Refined Remove Edge Rule
 <p>The diagram shows a flow from a grey box 'Selected:A' (with '«preserve»') to an orange box ':B' (with '«require#lb0»'). A red arrow labeled '«delete»' points from 'Selected:A' to a grey box 'OldTarget:B' (with '«preserve»'). From 'OldTarget:B', a red arrow labeled '«delete»' points to 'Selected:A'. Labels 'a' and 'b' are placed near the arrows. Below the diagram, it says 'Repeat require lb0 times'.</p>	 <p>The diagram shows a flow from a grey box 'Selected:A' (with '«preserve»') to an orange box ':B' (with '«require#lb0»'). A red arrow labeled '«delete»' points from 'Selected:A' to a grey box 'OldTarget:B' (with '«preserve»'). From 'OldTarget:B', a green arrow labeled '«create»' points to a grey box 'Selected#1:A' (with '«preserve»'). From 'Selected#1:A', a green arrow labeled '«create»' points to 'OldTarget:B'. Labels 'a' and 'b' are placed near the arrows.</p>
SERGe Add Edge Rule	
 <p>The diagram shows a flow from a grey box 'Selected:A' (with '«preserve»') to a grey box 'NewTarget:B' (with '«preserve»'). From 'NewTarget:B', a blue arrow labeled '«forbid#ub0»' points to a blue box ':A' (with '«forbid#ub0»'). A green arrow labeled '«create»' points from 'Selected:A' to 'NewTarget:B'. Labels 'a' and 'b' are placed near the arrows. Below the diagram, it says 'Repeat forbid ub0 times'.</p>	<p>No refinements necessary for this rule.</p>

Generally, the problem here is that we are running SERGe with the problem meta-model and that the solution meta-model introduces additional multiplicity constraints. These are not taken into account by the rules generated. Running SERGe with the solution meta-model does offer a solution to the problem: for

the case of a $[1 \dots 1]$ multiplicity constraint (as between **Feature** and **Class** in the CRA case), with the right configuration settings, SERGe can generate a *change edge* rule. In addition to this rule, two other rules are generated: add an unassigned **Feature** to an existing **Class** and create a **Class** and assign an unassigned **Feature** to it. The problem with these rules however, is that the search space cannot be fully explored once all **Features** have been assigned to a **Class**. After this happens, the only possible operation is to apply the *change Feature* rule to the search models and move **Features** between classes, but the transformations cannot perform any create and delete **Class** solution model changes. This limitation of rule applications on valid solutions leads to an incomplete search space exploration.

What we need, is an algorithm that generates transformation rules that are applicable to an instance model of the problem meta-model, allowing all types of transformations for the nodes we are interested in, but that ensure that any model edit operations will not introduce additional invalidations of the solution meta-model constraints. The generated rules must be able to perform the same edit operations on models conforming with both the problem meta-model and the solution meta-model. We ensure this by post-processing the rules produced by SERGe for the problem meta-model.

We have adapted the SERGe meta-tool by applying a set of refinements to the generated rules to ensure that the search process does not get stuck in local optima due to constraint invalidation. Our refinements, are additions to copies of generated SERGe rules, to ensure that when an instance of the solution meta-model is found, the search process can still evolve it by applying CPEOs to it, without breaking the constraints defined in the solution meta-model.

In our approach, we have implemented refinements aimed at solving the CRA case, therefore the list presented in this paper is not exhaustive and we aim to implement the remaining refinements in future work.

The overall rule-refinement process is the following:

1. For all nodes with given multiplicity-constraint refinements, check the validity of the refinements. In this step we ensure that the given refinement constraints are valid w.r.t. the problem meta-model, by ensuring that they specify a solution meta-model which is a subset of the problem meta-model;
2. Generate a new meta-model including only the upper-bound refinements. This meta-model is then used in the following step to generate the SERGe rules. Note that SERGe already handles upper-bound refinements the way we need them. Lower-bound refinements require post-processing of rules;
3. Run the SERGe meta-tool with the new meta-model and generate rules for the nodes specified in the problem specification. In this step, we run the SERGe algorithm with the meta-model having the specified upper-bound refinements set. This generates the rules as seen in the SERGe rules column in Table 2;
4. Create a copy of each of the generated rules and apply the following refinements to them, each refinement resulting in a new rule. For each of the refinements described in the following list, a before and after comparison can be found in Table 2:

- (a) If the rule is creating a new node type **A** and there is a lower bound refinement of an edge **a** or **b** at either side, then find another existing node of type **A** and for each created edge between the new node and the existing mandatory neighbours, add a delete edge between the existing node type **A** and the existing mandatory neighbours **B**. These refinements allow the rule to create a new node when there are no mandatory neighbours available to be assigned, by taking one from an existing node of the same type;
 - (b) If the rule is deleting a node type **A** and there is a lower-bound refinement of an edge **a** or **b** at either side of them, then find another existing node **A** and for each deleted edge between the deleted node **A** and the existing mandatory neighbours, add a create edge between the existing node **A** and the existing mandatory neighbours. The rule changes added by these refinements allow a node to be deleted and not leave mandatory neighbours dangling, by moving them to other existing nodes of the same type as the deleted node;
 - (c) If the rule is deleting an edge **a** between node type **A** and node type **B** and there is a lower bound refinement at either side of **A** or **B** then find another node of type **A** and create the deleted edge between **B** and the found node type **A**. This refinement results in a Change edge rule, which SERGe can also generate if configured to do so and the edge has a fixed multiplicity;
5. Remove duplicate rules by using the SERGe duplicate checker.

4.3 Running the Optimisation

Once the rule refinements are generated, MDEO groups the rules generated for the meta-model with the upper bound refinements and the new refined rules and runs the optimisation process with the complete set of generated rules. To run the optimisation for the CRA case we have implemented our proof of concept as a new feature of the MDEO tool. For this experiment we have created a standalone launcher for the tool to allow us to run the optimisation without having to run the tool as an Eclipse plugin for each of the configurations.

5 Evaluation

Our aim is to automatically generate evolvers from a metamodel so that we can then run MDEO to perform evolutionary optimisation on models without having to design the rules manually. We describe the ideal solution meta-model and which sections should be transformed and then the tool automatically generates the necessary transformation rules so that models can be evolved to become valid solution candidates. We have evaluated our solution starting from the following research question: Can we generate evolvers that perform optimisation as well as or better than the ones defined manually?

Table 3. Summary of MDEO TTC '16 input models results

	A	B	C	D	E
MDEO M I					
Best CRA	3.0	2.999	2.015	N/A	N/A
Mean CRA	1.978	1.954	1.232	N/A	N/A
Mean time	0 min 0 s 505 ms	0 min 1 s 083 ms	0 min 2 s 705 ms	0 min 8 s 946 ms	0 min 18 s 906 ms
MDEO M II					
Best CRA	3.0	3.104	2.910	5.531	3.098
Mean CRA	1.950	1.911	1.972	4.103	0.816
Mean time	0 min 2 s 464 ms	0 min 5 s 337 ms	0 min 12 s 293 ms	0 min 54 s 193 ms	3 min 7 s 864 ms
MDEO R I					
Best CRA	3.0	3.166	1.858	N/A	N/A
Mean CRA	2.627	2.114	0.327	N/A	N/A
Mean time	0 min 1 s 188 ms	0 min 1 s 892 ms	0 min 3 s 816 ms	0 min 9 s 760 ms	0 min 18 s 234 ms
MDEO R II					
Best CRA	3.0	4.083	3.177	5.794	2.618
Mean CRA	2.478	2.424	2.033	3.703	-0.035
Mean time	0 min 5 s 650 ms	0 min 10 s 290 ms	0 min 18 s 358 ms	1 min 05 s 752 ms	3 min 13 s 674 ms
MDEO S I					
Best CRA	1.75	0.791	-0.930	-2.646	N/A
Mean CRA	0.654	-0.629	-4.207	-8.293	N/A
Mean time	0 min 0 s 566 ms	0 min 1 s 117 ms	0 min 2 s 390 ms	0 min 6 s 767 ms	0 min 13 s 723 ms
MDEO S II					
Best CRA	2.333	0.983	-0.601	-3.785	-4.855
Mean CRA	0.783	-0.523	-4.732	-7.647	-11.555
Mean time	0 min 2 s 745 ms	0 min 5 s 491 ms	0 min 13 s 331 ms	0 min 44 s 963 ms	2 min 21 s 917 ms
MDEO C I					
Best CRA	3.0	2.833	2.017	N/A	N/A
Mean CRA	1.936	1.964	0.908	N/A	N/A
Mean time	0 min 0 s 958 ms	0 min 1 s 448 ms	0 min 3 s 290 ms	0 min 9 s 385 ms	0 min 18 s 256 ms
MDEO C II					
Best CRA	3.0	3.104	3.634	6.436	3.011
Mean CRA	2.072	2.050	2.454	4.770	0.401
Mean time	0 min 3 s 560 ms	0 min 7 s 480 ms	0 min 19 s 614 ms	1 min 12 s 672 ms	3 min 57 s 076 ms

In this section we compare the results obtained with the latest version of MDEO running with manual user defined evolvers, the SERGe generated evolvers for both the problem and the solution meta-models and the automatically generated evolvers with our refinements. By doing this comparison we show that the automatically generated rules are just as good as the user defined rules. We also compare our CRA case results with VIATRA-DSE results from TTC '16 [14]. Our configuration can produce solutions with empty classes in some instances. Although models with empty classes are not valid instances of the CRA metamodel, we ignore this requirement as part of this evaluation because it can be resolved with a post-processing step to remove empty classes from the solutions.

Experiment Setup. We ran our experiments for the CRA case using three Henshin transformation rules (evolvers) configurations:

MDEO Manual (MDEO M). Using the user defined evolvers, specified by us and previously used in the TTC 2016 Submission [15];

MDEO Refined (MDEO R). Using the evolvers automatically generated by the SERGe generated rules improvements described in this paper;

MDEO SERGe (MDEO S). Using the evolvers generated by SERGe without any of our refinements; and

MDEO SERGe Solution Metamodel (MDEO C). Using the evolvers generated by SERGe from the solution meta-model without any of our refinements.

For each evolver configuration we ran 30 experiments using the NSGA-II algorithm and the following parameters: **I** 100 evolutions and population size of 40; and **II** 500 evolutions and population size of 40. We have chosen the values for configuration **I** to have a comparison configuration with the solution proposed by VIATRA-DSE [14] for TTC 2016. The authors presented the results of 30 experiments running with a population of 40 and 100 evolutions.

The source code of the experiment together with the discovered solutions for all experiments can be found on GitHub⁵. All the experiments have been executed in headless mode on an AWS EC2 c4.large spot instances running Amazon Linux 4.4.2331.54.amzn1. x86_64 and Java 1.8.0_121 openjdk.

Results. In all configurations, computation time is partly given by the number of evolvers that have to be applied to a model in order to find mutation matches. Fewer evolvers require less computations to identify potential matches. This execution time difference can be observed between the MDEO R configurations which has seven evolvers and the other configurations which have three (MDEO C) and four evolvers (MDEO M, MDEO S), respectively. The NSGA-II algorithm used for our experiments requires more computation time when there are less convergent dominant solutions than the expected population size and it has to spend more time on sorting though crowded solutions.

In Table 3 we can see that all configurations have been able to find the same maximum CRA value for input model A, except for MDEO S. By inspecting the generated solutions and the average CRA value we can observe that MDEO R has found the highest overall values for model A, followed by MDEO C. The execution time is smaller for MDEO C than for MDEO R in both configurations. For input model B we can see that the best results are also obtained by MDEO R in both configurations. MDEO R also found the most good solutions overall, having the highest mean CRA value.

MDEO C found the best CRA value for input model C. Because the generated solutions are crowded and not diverse, the MDEO C **II** configuration takes more time than MDEO R and MDEO M to find the results, despite having only three evolvers compared to MDEO R which has seven.

For models D and E we note that configuration **I** does not actually find a solution. This is likely because not all features can be allocated to classes during

⁵ <https://github.com/mde-optimiser/gcm-2017-experiments>.

the first 100 evolutions. For configuration **II** we can find valid solutions. For model D, MDEO R finds a better CRA than MDEO M but worse than MDEO C. For model E, we suspect that MDEO R needs even more evolutions to produce good CRA values due to the large number of evolvers. However, by comparing the average CRA we can observe that MDEO M found the best overall solutions and is closely followed by MDEO C and MDEO R.

For all the input models evaluated, the MDEO S configuration may be getting stuck in local optima, because there are no rules to allow it to move a feature without invalidating a solution consistent with the solution meta-model. The best solutions it can find are given by the ones where all the features are assigned to classes the first time, when the solution becomes valid, during the evolution process. After this step, the solutions cannot generate better candidates through the mutations allowed by the generated evolvers for this configuration.

Table 4. Summary of VIATRA-DSE TTC '16 input models results

	A	B	C	D	E
Best CRA	3	4	3.002	5.08	8.0811
Mean CRA	3	3.75	19992	2.8531	5.0188
Mean time	0 min 4 s 729 ms	0 min 13 s 891 ms	0 min 17 s 707 ms	1 min 19 s 136 ms	9 min 14 s 769 ms

Comparing the MDEO R results with the VIATRA-DSE results for the CRA case included in Table 4, we can see that for configuration **I**, MDEO R found an equal CRA value for model A, but a worse mean CRA. For configuration **I**, MDEO R found worse CRA values for all other input models. For configuration **II**, MDEO R found again an equal CRA for model A and a better CRA value for all other models except E. The mean CRA values are worse for models A, B, and E and better for C and D. However it is worth noting that for configuration **II**, MDEO R ran for 500 evolutions compared to VIATRA-DSE which only ran for 100 evolutions. Also, the conditions under which the experiments for both solutions have been performed are very different, therefore a performance comparison of the two solutions is not possible.

Because we seek to apply optimisation directly on models through endogenous transformations, 100 evolutions is not enough to fully explore solutions which have close to or more than 100 features that have to be assigned. This can be observed in the results obtained by the MDEO R, which has seven evolvers compared to MDEO C which has only three or MDEO M which has four. However, given enough evolutions, the MDEO C and MDEO M are at a disadvantage when compared to MDEO R on the quality and diversity of explored solutions, because once all features are assigned to a class, no new classes can be created. This can lead to a limitation in search space exploration. This behaviour can be observed by analysing the mean CRA values of the smaller models (A-B) for configurations **I** and **II**.

In summary, we can say that the rule generation approach proposed in this paper produces rules that are comparable to manually written evolution rules. We can see that the obtained results for MDEO R are close to MDEO M or in

some cases, better. The main drawback for the refined rules configuration is that the number of evolvers is larger than the ones manually defined, this ending up as requiring a longer time and more evolutions to find good solutions.

We have only experimented with the CRA case so far. We are aware that the presented approach may not be valid for other cases in its current form, but we are encouraged by the results obtained and we are planning to extend it to support other cases in future work.

6 Conclusions and Outlook

In this paper we have shown an approach to specifying optimisation problems in an MDE context without the need to explicitly specify evolution rules. We have shown an algorithm to generate the evolution rules from a problem specification consisting of a meta-model, a set of additional multiplicity constraints, a set of objectives and a list of meta-classes.

We have been encouraged by the results. At this point, we do not yet have a proof of the correctness of our refinements (e.g., to show that they do indeed never create invalid candidate solutions). In future work, we plan to create such a proof and to extend our tool so it can be used for additional types of constraints beyond multiplicity constraints. Another improvement we are interested in adding to MDEOPTIMISER is support for a hyperheuristic algorithm to determine the best set of rule applications during an optimisation (e.g., using different rule sets during start up and during later stages of the search or by assigning weights to rules to prioritise their execution based on the results they find), so that we improve design space exploration in our optimisation process [23].

References

1. Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D.: A model-driven framework for guided design space exploration. In: Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 173–182, November 2011
2. Zschaler, S., Mandow, L.: Towards model-based optimisation: using domain knowledge explicitly. In: Proceedings of Workshop on Model-Driven Engineering, Logic and Optimization (MELO 2016) (2016)
3. Mészáros, T., Mezei, G., Levendovszky, T., Asztalos, M.: Manual and automated performance optimization of model transformation systems. *Int. J. Softw. Tools Technol. Transf.* **12**(3), 231–243 (2010)
4. Efstathiou, D., Williams, J.R., Zschaler, S.: Crepe complete: multi-objective optimisation for your models. In: Proceedings of 1st International Workshop on Combining Modelling with Search- and Example-Based Approaches (CMSEBA 2014) (2014)
5. Fleck, M., Troya, J., Wimmer, M.: Marrying search-based optimization and model transformation technology. In: Proceedings of 1st North American Search Based Software Engineering Symposium (NasBASE 2015) (2015, preprint). <http://martin-fleck.github.io/momot/downloads/NasBASE.MOMoT.pdf>

6. Drago, M.L., Ghezzi, C., Mirandola, R.: A quality driven extension to the QVT-relations transformation language. *Comput. Sci. - Res. Dev.* **30**(1), 1–20 (2015). First online: 24 November 2011
7. Burton, F.R., Paige, R.F., Rose, L.M., Kolovos, D.S., Poulding, S., Smith, S.: Solving acquisition problems using model-driven engineering. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) *ECMFA 2012*. LNCS, vol. 7349, pp. 428–443. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_32
8. Abdeen, H., Varró, D., Sahraoui, H., Nagy, A.S., Debreceni, C., Hegedüs, Á., Horváth, Á.: Multi-objective optimization in rule-based design space exploration. In: Crnkovic, I., Chechik, M., Grünbacher, P. (eds.) *Proceedings of 29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014)*, pp. 289–300. ACM (2014)
9. Harman, M., Jones, B.F.: Search-based software engineering. *Inf. Softw. Technol.* **43**(14), 833–839 (2001)
10. Fleck, M., Troya, J., Kessentini, M., Wimmer, M., Alkhazi, B.: Model transformation modularization as a many-objective optimization problem. *IEEE Trans. Softw. Eng.* **43**(11), 1009–1032 (2017). <https://doi.org/10.1109/TSE.2017.2654255>
11. Chatziprimou, K., Lano, K., Zschaler, S.: Surrogate-assisted online optimisation of cloud IaaS configurations. In: *IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 138–145 (2014)
12. Fleck, M., Troya, J., Wimmer, M.: The class responsibility assignment case, pp. 1–8 [24] (2016)
13. 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
14. Nagy, A.S., Szárnyas, G.: Class responsibility assignment case: a VIATRA-DSE solution, pp. 39–44 [24] (2016)
15. Burdusel, A., Zschaler, S.: Model optimisation for feature class allocation using MDEOptimiser: a TTC 2016 submission, pp. 33–38 [24] (2016)
16. 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
17. Eclipse.org: Viatra Project. <http://eclipse.org/viatra/>
18. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* **6**(2), 182–197 (2002)
19. Williams, J.R.: A novel representation for search-based model-driven engineering. Ph.D. thesis, University of York, UK (2013)
20. Mandow, L., Montenegro, J.A., Zschaler, S.: Mejora de una representación genética genérica para modelos. In: *Actas de la XVII Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA 2016)* (2016, in press)
21. Kehrer, T.: Calculation and propagation of model changes based on user-level edit operations. Ph.D. thesis, University of Siegen (2015)
22. Deb, K.: Multi-objective genetic algorithms: problem difficulties and construction of test problems. *Evol. Comput.* **7**(3), 205–230 (1999)
23. Cowling, P., Kendall, G., Soubeiga, E.: A hyperheuristic approach to scheduling a sales summit. In: Burke, E., Erben, W. (eds.) *PATAT 2000*. LNCS, vol. 2079, pp. 176–190. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44629-X_11
24. Garcia-Dominguez, A., Krikava, F., Rose, L.M. (eds.): *Proceedings of 9th Transformation Tool Contest*, vol. 1758. CEUR (2016)

Generating Efficient Predictive Shift-Reduce Parsers for Hyperedge Replacement Grammars

Berthold Hoffmann¹ and Mark Minas²(✉)

¹ Universität Bremen, Bremen, Germany
hof@informatik.uni-bremen.de

² Universität der Bundeswehr München, Neubiberg, Germany
mark.minas@unibw.de

Abstract. Predictive shift-reduce (PSR) parsing for a subclass of hyperedge replacement graph grammars has recently been devised by Frank Drewes and the authors. This paper describes in detail how efficient PSR parsers are generated with the *Grappa* parser generator implemented by Mark Minas. Measurements confirm that the generated parsers run in linear time.

Keywords: Hyperedge replacement grammar · Graph parsing
Parser generator

1 Introduction

Since the processing of diagram languages with computers becomes more and more common, the question whether a diagram adheres to the rules of such a language gets more and more important. If the rules of a language go beyond validity wrt. a metamodel, the notion of grammars gains relevance. Here, graph grammars are a natural candidate, in particular if they are context-free, like those defined by hyperedge replacement (HR) [10]. Unfortunately, general HR parsers, like the adaptation of the Cocke-Younger-Kasami (CYK) parser to graphs [13], do not scale to graphs of the size used in modern applications, e.g., in model transformation. So it is worthwhile to identify subclasses of HR grammars that have efficient parsers. After devising *predictive top down parsing* (PTD) [4], Frank Drewes and the authors have recently proposed *predictive shift-reduce parsing* (PSR) [6], its bottom-up counterpart, which lifts *SLR(1)* string parsing to graphs. Now Mark Minas has completed his implementation of *Grappa*, a generator for PTD and PSR parsers.¹

In order to keep the paper self-contained, we start with a brief account of HR grammars in Sect. 2, introduce PSR parsing in Sect. 3, and sketch conflict analysis in Sect. 4. (More details can be found in [6].) Then we describe the implementation of efficient PSR parsers generated by *Grappa* in Sect. 5. Evaluation of their efficiency in Sect. 6, also in comparison to PTD and CYK parsers,

¹ *Grappa* is available at www.unibw.de/inf2/grappa.

confirm that they run in linear time. Finally we point out some future work, in Sect. 7.

2 Hyperedge Replacement Grammars

We use the wellknown relation between graphs and logic [1] to define graphs and hyperedge replacement grammars in a form suited to define parsing.

Definition 1 (Graph). Let Σ be a vocabulary of *symbols* that comes with an *arity function* $\text{arity}: \Sigma \rightarrow \mathbb{N}$, and let X be an infinite set of *variables*. We assume that Σ is the disjoint union of *nonterminals* N and *terminals* T .

A *literal* $e = \ell(x_1, \dots, x_k)$ consists of a symbol $\ell \in \Sigma$ and $k = \text{arity}(\ell)$ variables x_1, \dots, x_k from X . A *graph* is a sequence $G = e_1 \dots e_n$ of literals. With $\Sigma(G)$ and $X(G)$ we denote the *symbols* and *variables* occurring in G , respectively.

$X(G)$ represents the *nodes* of G , and a literal $e = \ell(x_1, \dots, x_k)$ represents a *hyperedge* (*edge*, for short) that carries the *label* ℓ and is attached to the nodes x_1, \dots, x_k . If necessary, isolated nodes can be represented by a literal $\nu(x)$, where ν is a *fictitious node symbol* with $\text{arity}(\nu) = 1$. Multiple occurrences of literals represent parallel edges.

Definition 2 (HR Grammar). A pair $r = (L, R)$ of graphs is a *hyperedge replacement rule* (*rule* for short) if its *left-hand side* L consists of a single non-terminal literal and if its *right-hand side* R satisfies $X(L) \subseteq X(R)$; we usually denote a rule as $r = L \rightarrow R$.

An injective function $\varrho: X \rightarrow X$ is a *renaming*; G^ϱ denotes the graph obtained by replacing all variables in G according to ϱ .

Consider a graph G and a rule r as above. A renaming $\mu: X \rightarrow X$ *matches* r to the i th literal of G if $L^\mu = e_i$ for some $1 \leq i \leq n$ and $X(G) \cap X(R^\mu) \subseteq X(L^\mu)$. A match μ of r *rewrites* G to the graph $H = e_1 \dots e_{i-1} R^\mu e_{i+1} \dots e_n$. This is denoted as $G \Rightarrow_{r, \mu} H$, or just as $G \Rightarrow_r H$. We write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_r H$ for some rule r taken from a finite set \mathcal{R} of rules, and denote the reflexive and transitive closure of this relation by $\Rightarrow_{\mathcal{R}}^*$, as usual.

A *hyperedge replacement grammar* $\Gamma = (\Sigma, T, \mathcal{R}, Z)$ (*HR grammar* for short) consists of a finite set \mathcal{R} of rules over Σ , and of a start graph $Z = S()$ with $S \in N$ and $X(Z) = \emptyset$. Γ generates the language $\mathcal{L}(\Gamma) = \{G \mid Z \Rightarrow_{\mathcal{R}}^* G, \Sigma(G) \subseteq T\}$.

Example 1 (Nested Triangles). Consider nonterminals S and \blacktriangle and the terminal Δ . We use $\ell^{x_1 \dots x_k}$ as a shorthand for literals $\ell(x_1, \dots, x_k)$. (Here ε denotes the empty variable sequence.) Then the rules

$$S^\varepsilon \rightarrow \blacktriangle^{xyz} \quad \blacktriangle^{xyz} \rightarrow \Delta^{xuv} \Delta^{uyw} \Delta^{vuz} \quad \blacktriangle^{uvw} \quad \blacktriangle^{xyz} \rightarrow \Delta^{xyz}$$

(which are numbered 1, 2, and 3) generate a nested triangle:

$$\begin{aligned} S^\varepsilon &\xRightarrow{1} \blacktriangle^{123} \xRightarrow{2} \Delta^{145} \Delta^{426} \Delta^{563} \blacktriangle^{465} \xRightarrow{2} \Delta^{145} \Delta^{426} \Delta^{563} \Delta^{478} \Delta^{769} \Delta^{895} \blacktriangle^{798} \\ &\xRightarrow{3} \Delta^{145} \Delta^{426} \Delta^{563} \Delta^{478} \Delta^{769} \Delta^{895} \Delta^{798} \end{aligned}$$

In Fig. 1, the graphs of this derivation are drawn as diagrams.

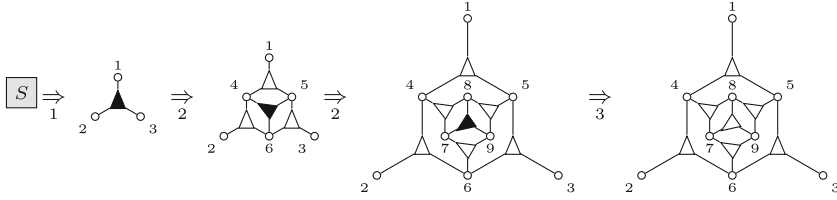


Fig. 1. Diagrams of a derivation of a nested triangles. Circles represent nodes, boxes and triangles represent edges of triangle graphs, which are connected to their attached nodes by lines; these lines are ordered clockwise around the edge, starting at the sharper corner of the triangle.

3 Predictive Shift-Reduce Parsing for HR Grammars

A parser attempts to construct a derivation for a given input according to some grammar. In our case, graphs shall be parsed according to an HR grammar. A bottom-up parser constructs the derivation by an operation called *reduction*: the right-hand side of a rule is matched in the input graph, and replaced by its left-hand side. A bottom-up parser for the nested triangles in Example 1 may reduce every triangle Δ^{abc} according to rule 3. However, only a single reduction, of the “central” triangle, will lead to a successful chain of reductions reaching the start graph S^ϵ . Cocke-Younger-Kasami parsers use this idea (after transforming grammars into Chomsky normal form). Even if this works for small graphs with up to hundred edges [13], it does not scale to bigger graphs. See our evaluation in Sect. 6 below.

PSR parsers borrow an idea of context-free bottom-up string parsers: they consume edges of a graph in exactly the order in which they would be constructed by a derivation. An operation, called *shift*, puts terminal edges onto a stack, to be considered for reduction later. A PSR parser will reduce rule 3 only once, after all other edges have been shifted; further reductions of rule 2, and finally of rule 1, may then lead to a successful parse.

A predictive bottom-up shift-reduce parser uses a *characteristic finite automaton* (CFA) to control its actions. We describe its construction at hand of the running example. The states of the CFA are defined as sets of *items*, which are rules where a dot indicates how far the right-hand side has been shifted onto the stack. Consider the item $\blacktriangle^{xyz} \rightarrow \Delta^{xuv} \Delta^{uyw} \Delta^{vzw} \cdot \blacktriangle^{uvw}$ of rule 2: Here the parser has shifted all terminals, but not the nonterminal. This item will constitute a *kernel item* of some state of the CFA, say q_3 . All variables x, y, z, u, v, w of the rule are known in this situation. So we consider them as *parameters* of the state, and denote it as $q_3(x, y, z, u, v, w)$. Before the missing \blacktriangle^{uvw} can be shifted, the parser must recursively parse rule 2 or 3. So the items $\blacktriangle^{uvw} \rightarrow \cdot \Delta^{urs} \Delta^{rwt} \Delta^{stv} \blacktriangle^{rts}$ and $\blacktriangle^{uvw} \rightarrow \cdot \Delta^{uvw}$ are added to q_3 as *closure items*. The dots at the start of these items indicate that nothing of these rules has been shifted in this state. We have to rename variables in order to avoid name clashes with the kernel item.

Like every CFA state, q_3 has transitions under every symbol appearing after the dot in some of its items. A transition under Δ^{urs} leads from q_3 to a state with the kernel item $\blacktriangle^{uvw} \rightarrow \Delta^{urs} \blacktriangle^{rwt} \Delta^{stv} \blacktriangle^{rts}$. No closure items arise in this state since the dot is in front of a terminal. This state would be denoted as $q'_1(u, w, v, r, s)$, but if there is already a state $q_1(x, y, z, u, v)$ that is equal to q'_1 up to variable names, we redirect the transition to this state and write a “call” $q_1(u, w, v, r, s)$ on the transition to specify how parameters should be passed along. Another transition, under Δ^{uvw} , leads to a state with kernel item $\blacktriangle^{uvw} \rightarrow \Delta^{uvw} \cdot$, say $q_5(u, w, v)$. This transition matches a terminal where all nodes are known; so it differs from that under Δ^{urs} that has to match two nodes r and s to hitherto unconsumed nodes. Finally, a transition under \blacktriangle^{xyz} leads from q_3 to a state, say $q_4(x, y, z, u, v, w)$ with the kernel item $\blacktriangle^{xyz} \rightarrow \Delta^{xuv} \Delta^{uyw} \Delta^{v wz} \blacktriangle^{uvw} \cdot$.

A special case arises in the start state q_0 . In order to work without backtracking, some nodes of the start rule must be uniquely determined in the input graph before parsing starts. In our example, all nodes x, y, z match the unique nodes a, b, c that are attached to just one edge, with their first, second, and third attachment, respectively. If the input graph does not have exactly three nodes like that, it cannot be a nested triangle, and parsing fails immediately. Otherwise the start state is called with $q_0(a, b, c)$. Unique start nodes can be determined by a procedure devised in [5, Sect. 4], which computes the possible incidences of all nodes created by a grammar.

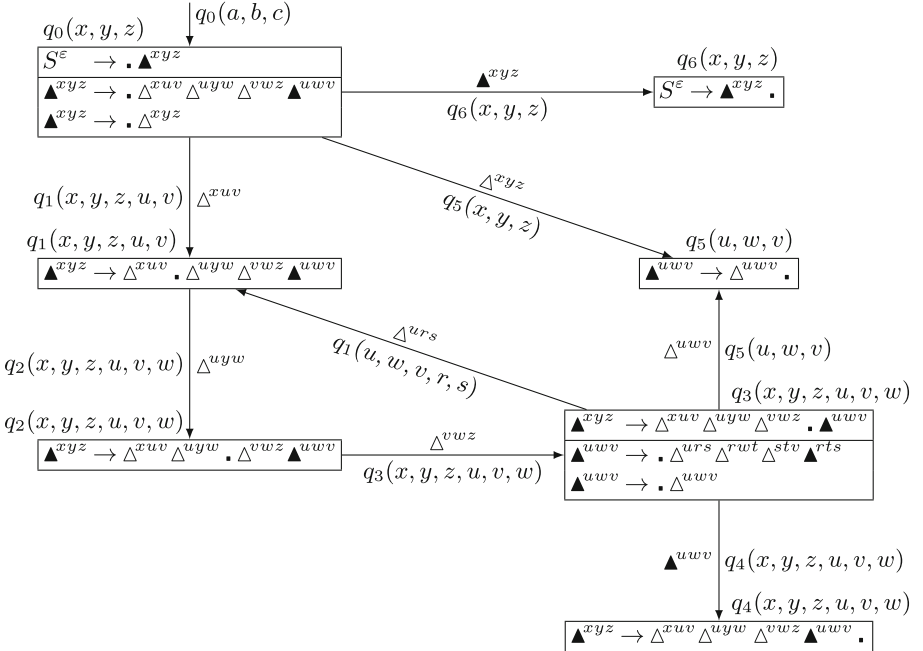


Fig. 2. The characteristic finite automaton for nested triangles

Example 2 (A CFA for Nested Triangles). In Fig. 2 we show the transition diagram of the CFA of Example 1. The states q_1, q_3, q_4, q_5 are as discussed above.

A PSR parser pushes concrete states and transitions of its CFA onto a stack while it performs transitions. In the states and transitions stored on the stack, variables in the abstract states and transitions of the CFA are replaced by the concrete nodes matching them in the input graph.

The topmost stack entry, a state, determines the next action of the parser. This may be a shift under a terminal literal, or a reduction of some rule. (Transitions under nonterminal literals are handled as final part of a reduction.) The actions for some topmost state q are as follows:

Shift: If q calls for a terminal transition under some literal $\ell(x_1, \dots, x_k)$, lookup the concrete nodes of q matching some of these variables, and match the literal with an edge $e = \ell(v_1, \dots, v_k)$ in the host graph. Push e onto the stack, remove it from the input, and push the target state, replacing variables with the concrete nodes determined by q and e .

Reduce: If q calls for a reduction of some rule r , pop all literals of the right-hand side of r from the stack, with their corresponding states. The state that is on top has a transition under the left-hand side of r ; push the left-hand side and its target state, replacing the variables with the nodes determined by the popped states. If the rule r is the start rule, and the input graph is empty, *accept* the input as a graph of the language.

Note that the parser has to choose the next action in states that allow for different shifts and/or reductions; in our example, q_0 and q_3 allow two shifts, see Fig. 2. The parser predicts the next step by inspecting the unconsumed edges. We will discuss in the next section how the conditions to be used for inspection are computed. In the example, the shifts of Δ^{xyz} in state q_0 and of Δ^{uvw} in state q_3 have to be chosen if and only if the graph contains unconsumed Δ -edges visiting the corresponding nodes.

Even if a particular shift transition has been chosen, a PSR parser may still have to choose between different edges matching the literal. (Such a situation does not occur in our example, but with the trees in [6, Sect. 4]). In such a case, the PSR parser generator has to make sure that the *free edge choice* property holds, i.e., that any of the matching edges can be chosen without changing the result of the parser.

Example 3 (A PSR Parse for Nested Triangles). A parse of the graph derived in Example 1 is shown in Fig. 3. We write the parameters of states as exponents, just as for literals.

4 Conflict Analysis

A CFA can be constructed for every HR grammar; the general procedure works essentially as described above. In this paper, we focus on the implementation of

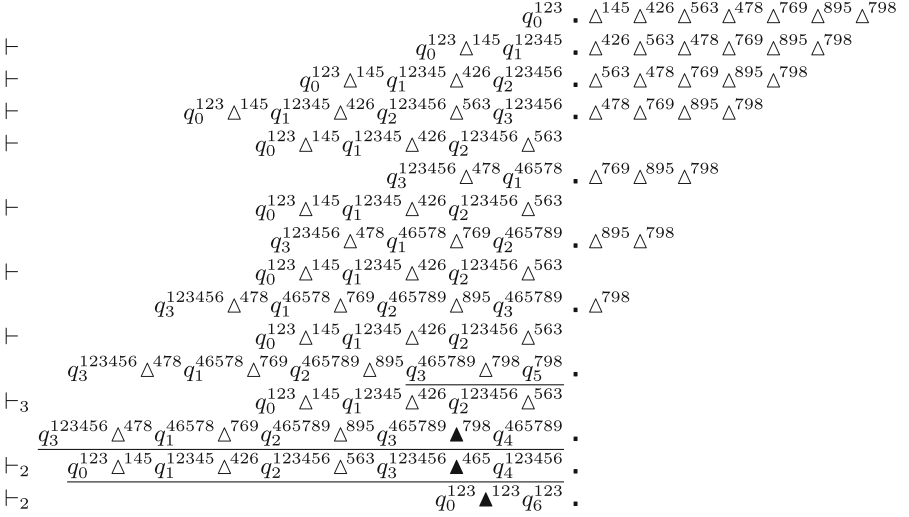


Fig. 3. A PSR parse for the triangle derived in Example 1

parsers for HR grammars that are PSR-parsable. Criteria for an HR grammar to be PSR-parsable have been discussed in [6]. In particular, such a grammar must be *conflict-free*. In the following, we roughly recall this concept since it is needed for the implementation of efficient PSR parsers, which is described in Sect. 5.

A graph parser must choose the next edge to be consumed from a set of appropriate unconsumed edges. We define a *conflict* as a situation where an unconsumed edge is appropriate for one action, but could be consumed also if another action was chosen. Obviously, the parser can always predict the correct action if the grammar is free of conflicts.

We now discuss how to identify host edges that are appropriate for the action caused by an item. For this purpose, let us first define items in PSR parsing more formally: An item $I = \langle L \rightarrow \bar{R} \cdot R \mid P \rangle$ consists of a rule $L \rightarrow \bar{R}R \in \mathcal{R}$ with a dot indicating a position in the right-hand side, and of the set P of parameters, i.e., those nodes in the item which do already have matching nodes in the host graph. These host nodes are not yet known when we construct the CFA and the PSR parser, but we can interpret parameters as abstract host nodes. A “real” host node assigned to a parameter during parsing is mapped to the corresponding abstract node. All other host nodes are mapped to a special abstract node $-$. Edges of the host graph are mapped to abstract edges being attached to abstract nodes, i.e., $P \cup \{-\}$, and each abstract edge can be represented by an *abstract (edge) literal* in the usual way. Note that the number of different abstract literals is finite because $P \cup \{-\}$ is finite.

Consider any valid host graph G in $\mathcal{L}(T)$, generated by the derivation $S = G_1 \Rightarrow \dots \Rightarrow G_n = G$. We then select any mapping of nodes in G to abstract nodes $P \cup \{-\}$ such that no node in P is the image of two different host nodes. Edge literals are mapped to the corresponding abstract literals. The resulting

sequence of literals can then be viewed as a derivation in a context-free string grammar $\Gamma(P)$ that can be effectively constructed from Γ in the same way as described in [5, Sect. 4]; details are omitted here because of space restrictions. $\Gamma(P)$ has the nice property that we can use this context-free string grammar instead of Γ to inspect conflicts. This is shown in the following.

Consider an item $I = \langle L \rightarrow \bar{R} \cdot R \mid P \rangle$. Each edge literal $e = l(n_1, \dots, n_k)$ has the corresponding abstract literal $\text{abstr}_P(e) = l(m_1, \dots, m_k)$ where $m_i = n_i$ if $n_i \in P$, and $m_i = -$ otherwise, for $1 \leq i \leq k$. Let us now determine all host edges, represented by their abstract literals, which can be consumed next if the action caused by this item is selected. The host edge consumed next will be the abstract literal $\text{First}_P(R) := \text{abstr}_P(e)$ if I is a shift item, i.e., R starts with a terminal literal e . If I , however, causes a reduction, i.e., $R = \varepsilon$, we can make use of $\Gamma(P)$. Any host edge consumed next must correspond to an abstract literal that is a follower of the abstract literal of L in $\Gamma(P)$. We refer to [6] for a discussion of the general case. Here, we discuss the concept at hand of our running example.

As an example, consider state $q_3(x, y, z, u, v, w)$ in Fig. 2 with its items $I_i = \langle L_i \rightarrow \bar{R}_i \cdot R_i \mid P_i \rangle$, $i = 1, 2, 3$, with $P_1 = \{x, y, z, u, v, w\}$ and $P_2 = P_3 = \{u, v, w\}$. For the second item, one can compute

$$\text{First}_{P_2}(R_2) = \text{abstr}_{P_2}(\Delta^{urs}) = \Delta^{u--},$$

i.e., the shifted edge in this step (called *shift step 1* in the following) must be a triangle edge being attached to the host node assigned to u with its first arm, and nodes that have not yet been consumed by the parser with its other arms.

For the third item, one can compute

$$\text{First}_{P_3}(R_3) = \text{abstr}_{P_3}(\Delta^{uvw}) = \Delta^{uvw},$$

i.e., the shifted edge in this step (called *shift step 2* in the following) must be a Δ edge being attached to host nodes that are assigned to u , w , and v , respectively.

The parser needs a criterion for deciding the correct step when it has reached $q_3(x, y, z, u, v, w)$. It is clear that shift step 1 must be taken when the host graph contains an unconsumed edge matching Δ^{u--} , but not Δ^{uvw} , and shift step 2 if it contains an unconsumed edge matching Δ^{uvw} , but not Δ^{u--} . However, the parser would be unable to decide the next step if the host graph contained two unconsumed edges matching Δ^{u--} and Δ^{uvw} , respectively. Conflict analysis makes sure that this situation, called *shift-shift conflict*, cannot occur here. This is outlined in the following.

Let us assume that this conflicting situation occurs, i.e., Δ^{uvw} may follow later when shift step 1 is taken, or Δ^{u--} may follow later when shift step 2 is taken. Conflict analysis, therefore, must compute from $\Gamma(P)$ the (finite) set of all abstract edges that may follow when either shift step is taken. Let us denote this set as $\text{Follow}_P^*(I_i)$, $i = 2, 3$. Its computation is straightforward and well-known from string grammars. In our example

$$\begin{aligned} \text{Follow}_{P_2}^*(I_2) &= \{\Delta^{u--}, \Delta^{-w-}, \Delta^{-v-}, \Delta^{---}\} \\ \text{Follow}_{P_3}^*(I_3) &= \{\Delta^{uvw}\} \end{aligned}$$

Of course, $First_{P_i}(R_i) \in Follow_{P_i}^*(I_i)$ for $i = 2, 3$. As one can see, $\Delta^{u--} \notin Follow_{P_3}^*(I_3)$ and $\Delta^{uvw} \notin Follow_{P_2}^*(I_2)$, i.e., such a shift-shift conflict cannot occur, and the parser can decide in state q_3 which step shall be taken, by checking whether there exists a yet unconsumed edge matching Δ^{u--} or Δ^{uvw} .

Similar arguments apply when the parser has to decide between a shift and a reduce step and between two reduce steps, potentially causing *shift-reduce* or *reduce-reduce conflicts* as discussed in [6]. But these situations do not occur in the CFA of our running example.

5 Efficient Implementation of PSR Parsers

We shall now describe how PSR parsers can be implemented efficiently so that their runtime is linear in the size of the input graph. We shall first describe the implementation at hand of our running example, and then the general procedure.

In the following, we assume that nodes and edges are represented by separate data structures. Each edge keeps track of its label and all attached nodes. We will discuss later what information must be stored at node objects to make parsing efficient.

The implementation of the parser outlined in Sect. 3 is rather straightforward: The parsing stack described in Sect. 3 holds (terminal) host edges as well as nonterminal edges produced by reduce steps, and CFA states with their parameters bound to host nodes that have already been consumed by the parser. In Example 3, we have represented such a state $q_i(x_1, \dots, x_k)$ together with its binding match $\mu: \{x_1, \dots, x_k\} \rightarrow X(G)$ by $q_i^{\mu(x_1)\dots\mu(x_k)}$. In the implementation, we represent each state just by its number i , and its binding by an array *params* of host nodes such that $params[j] = \mu(x_j)$ for each $j = 1, \dots, k$. And, instead of just a single stack, we shall use three stacks: a *stateStack* of state numbers, a *paramStack* of node arrays representing binding matches, and an *edgeStack* of (terminal) host edges and nonterminal edges produced by reduce steps. The elements stored in *stateStack* and in *paramStack* correspond to each other; each corresponding pair represents a state $q_i^{\mu(x_1)\dots\mu(x_k)}$ with a binding match μ . The parser is then implemented as the procedure *parse* shown in Fig. 4. The start nodes, which have been determined before parsing begins (see Sect. 3), represented by an array *startNodes*, are passed as a parameter. The parser initializes its stacks with the start state, which we assume to have number 0, together with the binding match defined by the start nodes. The actual parsing actions are implemented in procedures *action_i*, one for each CFA state $q_i(x_1, \dots, x_k)$. It is their task to operate on the stacks and to terminate the seemingly infinite loop.

Figure 5 shows the *action*-procedure for the accept state $q_6(x, y, z)$ of the nested triangle CFA (Fig. 2); the parser terminates with success iff all nodes and edges of the host graph have been consumed when this state is reached. The action procedures for states $q_3(x, y, z, u, v, w)$ and $q_5(u, w, v)$ are shown in Fig. 6. Procedure *action₃* must check which of the two shift transitions leaving q_3 must be taken. The third transition leaving q_3 , labeled with nonterminal edge \blacktriangle^{uvw} , is implemented in procedure *goto₃* and described later. Procedure *action₃*

```

procedure parse (startNodes: array of Node)
  push 0 on stateStack;
  push startNodes on paramStack;
  while true do
    i  $\leftarrow$  top of stateStack;
    call actioni;
  end
end

```

Fig. 4. The parsing procedure

```

1  procedure action3
2    params  $\leftarrow$  top of paramStack;
3    u  $\leftarrow$  params[4]; v  $\leftarrow$  params[5];
4    w  $\leftarrow$  params[6];
5    /* shift  $\Delta^{urs}$  */
6    e  $\leftarrow$  any unconsumed edge  $\Delta^{\alpha\beta\gamma}$ 
7      in the host graph with  $\alpha = u$ ;
8    if e exists and  $\beta, \gamma$  are unconsumed
9    then
10     mark e,  $\beta$ , and  $\gamma$  as consumed;
11     nextParams  $\leftarrow$ 
12       new array {u, w, v,  $\beta$ ,  $\gamma$ };
13     push e on edgeStack;
14     push 1 on stateStack;
15     push nextParams on paramStack;
16     return
17   end;
18   /* shift  $\Delta^{uvw}$  */
19   if e exists and  $\beta = w$  and  $\gamma = v$ 
20   then
21     mark e as consumed;
22     nextParams  $\leftarrow$ 
23       new array {u, w, v};
24     push e on edgeStack;
25     push 5 on stateStack;
26     push nextParams on paramStack;
27     return
28   end;
29   stop with error
30 end

```

Fig. 6. Some procedures implementing the PSR parser for nested triangles.

first tries to find a yet unconsumed edge of the host graph that corresponds to Δ^{urs} , where *u* is a parameter node, i.e., bound to a host node that is stored at position 4 of the current parameter array, whereas *r* and *s* must correspond to host nodes that have not yet been consumed. Such a host edge is looked for in lines 6–7. Grammar analysis shows that a host graph in the language of

```

procedure action6
  if all edges and nodes
    have been consumed
  then
    stop with success
  end;
  stop with error
end

```

Fig. 5. Action for state q_6

```

31 procedure goto3(e : Edge)
32   params  $\leftarrow$  top of paramStack;
33   x  $\leftarrow$  params[1]; y  $\leftarrow$  params[2];
34   z  $\leftarrow$  params[3]; u  $\leftarrow$  params[4];
35   v  $\leftarrow$  params[5]; w  $\leftarrow$  params[6];
36   /*  $\blacktriangle^{uvw}$  */
37   if e has label  $\blacktriangle$  and
38     visits nodes (u, w, v)
39   then
40     nextParams  $\leftarrow$ 
41       new array {x, y, z, u, v, w};
42     push e on edgeStack;
43     push 4 on stateStack;
44     push nextParams on paramStack;
45     return
46   end;
47   stop with error
48 end
49 procedure action5
50   params  $\leftarrow$  top of paramStack;
51   u  $\leftarrow$  params[1]; w  $\leftarrow$  params[2];
52   v  $\leftarrow$  params[3];
53   /* reduce rule 3 */
54   e  $\leftarrow$  new edge with label  $\blacktriangle$  and
55     visiting nodes (u, w, v);
56   pop 1 element from edgeStack,
57     stateStack, and paramStack;
58   i  $\leftarrow$  top of stateStack;
59   call gotoi(e)
60 end

```

nested triangles cannot contain more than one edge like that. (We will discuss later how the parser generator can do so.) The parser, therefore, looks whether it finds any such edge and, if successful, stores it in e . The parser takes the corresponding shift transition to $q_1(u, w, v, r, s)$ if such an edge exists and if its other two connected nodes have not been consumed before (line 8). The shift step marks the identified edge e and nodes β and γ as consumed and computes the parameter array of the next state $q_1(u, w, v, r, s)$. (Details on the corresponding data structures are discussed later.) The host nodes corresponding to u , w , and v are already known from the current parameter array, but r and s (at positions 4 and 5 of the array) are the nodes β and γ visited by edge e . The procedure then returns, and the parsing loop can continue with the next iteration.

Procedure $action_3$ checks the other shift transition if the test in line 8 fails. Lines 19–28 are similar to lines 8–17. Note that the parser need not look for another edge than the one found in lines 6–7 once such an edge has been found. Finally, the parser must stop with an error if the test in line 19 also fails, because a valid host graph must contain an edge satisfying one of the two conditions.

Procedure $action_5$ shows the implementation of the reduce step to be taken in state $q_5(u, w, v)$. Lines 54–55 create the nonterminal edge corresponding to \blacktriangle^{uvw} produced by the reduce step (see Fig. 2). Lines 56–57 pop the elements corresponding to the right-hand side of the rule from the stacks, i.e., the CFA returns to state q_i (line 58) where a transition labeled with the newly created edge e must be taken (line 59). Such a *goto*-procedure for state $q_3(x, y, z, u, v, w)$ is shown in lines 31–48. Lines 37–38 check whether the parameter edge matches the one defining the transition. This is actually not necessary here where we have just a single transition with a nonterminal edge; but in general, there may be several leaving transitions with different nonterminal edges, and the *goto*-procedure must select the correct one. Lines 40–44 move the parser into the next state $q_4(x, y, z, u, v, w)$.

The other *action* and *goto* procedures are similar to the presented ones. Let us now consider the general case. Each state of a CFA provides a set of operations which can be *shift*, *reduce*, *accept*, or *goto* operations. *Shift* and *goto* operations correspond to transitions to other states; *shift* transitions are labeled by terminal edge literals and *goto* transitions by nonterminal edge literals. The latter are easily implemented by *goto* procedures similar to $goto_3$ in Fig. 6. Each *goto* procedure must check a fixed number of different cases, which can be performed in constant time. The *action* procedures are responsible for choosing among the *shift*, *reduce*, and *accept* operations provided by the corresponding states. For each of these operations, the parser generator must identify a condition that controls when the parser shall select its operation to be executed next. In the example, procedure $action_3$ selects the shift over Δ^{wrs} iff the condition in line 8 is satisfied, and a shift over Δ^{uvw} if the condition in line 8 is not satisfied, but the one in line 19. Moreover, the parser must be able to efficiently check these conditions, i.e., in constant time.

The conditions can be easily derived from the conflict analysis described in the previous section. Conflict analysis determines, for each item of a state, a

finite characterization of terminal edges of the host graph that must be consumed next (for *shift* steps) or that may be consumed in later steps (for *reduce* steps). There are no conflicts if an HR grammar is PSR, i.e., the parser can use these conditions to always correctly select the next operation.

Now, how do these conditions look like? Each one is an abstract literal characterizing (yet unconsumed) terminal edges of the host graph determining their label and some of their nodes, whereas the other attached nodes are yet unconsumed. A naïve procedure for searching for such an edge would be to iterate over all unconsumed edges and to select an edge that has attached nodes as specified. This would take linear time instead of the required constant time; proper preprocessing of the host graph prior to parsing is necessary. The parser generator knows about all abstract literals that are used in any condition. The idea is to preprocess the host graph so that each abstract literal corresponds to a data structure of all unconsumed edges that match the abstract literal, and to update the corresponding data structures whenever an edge is consumed. The parser, when searching for an unconsumed edge matching an abstract literal, then just has to look into the corresponding data structure. Hash tables are an appropriate data structure for this purpose. For each abstract literal $a = l(m_1, \dots, m_k)$, we assign a hash table to label l . Prior to parsing, each edge e matching this literal is added to this hash table. More specifically, all nodes attached to e and being determined by the abstract literal define a tuple $Key_a(e)$ of nodes. $Key_a(e)$ is mapped to a list in the hash table; this list contains e and (when the complete graph has been preprocessed) all edges that have the same key $Key_a(e)$. Searching for an edge being attached to some predefined nodes specified by an abstract literal then consists of just computing the corresponding key and looking up the mapped list in the appropriate hash table.

The time for looking up this list is constant on average² because the hash table is fixed after preprocessing; only the contents of the list are modified when edges are consumed. This can be done in constant time, too, when lists are implemented by doubly linked lists and each edge keeps track of the list nodes of all lists in which the edge is stored. Because the number of abstract literals is fixed for a grammar, all these data structures (hash table, lists, and keeping track of list nodes) require linear space in the size of the host graph if each hash table size is chosen proportional to the number of all edges. Moreover, setting up these data structures requires, on average, linear time in the size of the host graph.

However, looking up unconsumed edges matching certain host nodes specified by an abstract literal can be simplified in many cases (for instance in our running example of nested triangles) so that hash tables become obsolete in many cases, or altogether: Grammar analysis may reveal, for a state $q_m(x_1, \dots, x_k)$ of the CFA, a terminal edge label $l \in T$, a parameter node x_i and an “arm”

² As a worst case, a single lookup can take linear time in the number of hash table entries when the hash function produces too many collisions. However, the parser has to look up all edges. The overall lookup time, therefore, tends towards the average case.

j , $1 \leq j \leq \text{arity}(l)$, that a host graph cannot have more than one unconsumed edge with label l and being attached to $\mu(x_i)$ with its j -th arm when the parser has reached q_m^μ . If an abstract literal used for edge lookup refers to such a node, called *determining node* in the following, one can use just this node as a key in the hash table. However, this makes the hash table unnecessary: Instead of mapping a node to a list of attached edges, one can simply keep this list in the node data structure. Such a list is just a plain association list which must be maintained when consuming edges.

Grammar analysis can identify determining nodes by using the same techniques as for conflict analysis, which computes sets of abstract edges that may follow during parsing. If this computation shows that a parameter node is attached to a shift edge, but to no other edge with the same label and using the same arm later in the derivation process, one can conclude that such a parameter node is determining for the corresponding edge label and arm. Amazingly, experiments with many PSR grammars have shown that almost all of them can be processed without any hash table.

Finally note that PSR parsers not only process syntactically correct graphs in linear time, but also erroneous graphs. This is so because each step of the parser still takes constant time (at least on average), and the maximum number of steps linearly depends on the graph size.

6 Evaluation of Generated PSR Parsers

In order to demonstrate that PSR parsing is linear in the size of the host graph, we have conducted some experiments with some example HR grammars. For each grammar, we generated a PSR as well as a PTD parser using *Grappa*, and also a CYK parser using *DiaGen*.³ We then measured parsing time for input graphs of different size for each of these parsers. The results can be found at www.unibw.de/inf2/grappa; here, we present the results for our running example of nested triangles and also for Nassi-Shneiderman diagrams as well as “blowball graphs”.

Each triangle graph consists, for some positive integer n , of $3n$ nodes and $3n - 2$ edges. Figure 7a shows the runtime of the PSR and PTD parsers when processing triangle graphs with varying value n . Runtime has been measured on a MacBook Pro 2013, 2,7 GHz Intel Core i7, Java 1.8.0, and is shown in milliseconds on the y -axis while n is shown on the x -axis. Note the apparent linear behavior of the PSR parser and the, slightly slower, PTD parser. Figure 7b shows the corresponding diagram for the CYK parser. Note that the runtime of the CYK parser is not linear in the size of the triangle graph. Note also that PTD parsing and, in particular, PSR parsing is, by several orders of magnitude, faster than CYK parsing. For instance, the CYK parser needs 700 ms to parse a triangle graph with $n = 1000$ whereas the PTD parser needs just 0.97 ms, and the PSR parser just 0.44 ms.

³ Homepage: www.unibw.de/inf2/DiaGen.

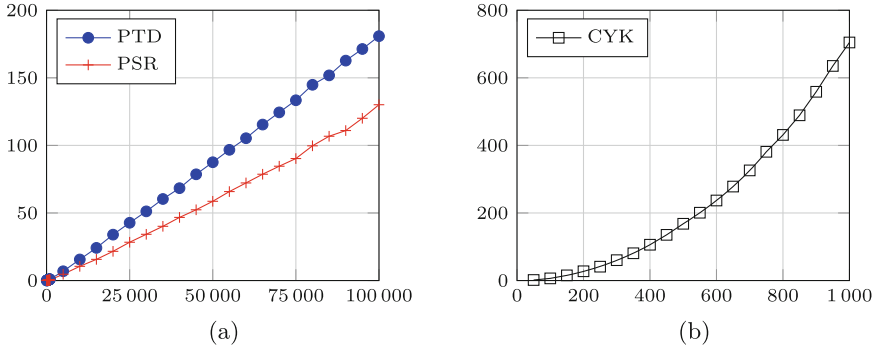


Fig. 7. Runtime (in milliseconds) of the PSR as well as the PTD parser (a) and the CYK parser (b) for nested triangles. Note that the scales in (a) and (b) differ.

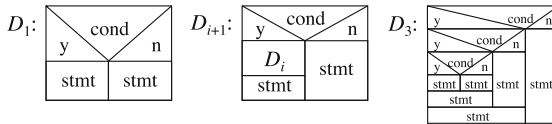


Fig. 8. Nassi-Shneiderman diagrams D_i , $i = 1, 2, 3, \dots$

We also conducted experiments with the more complicated language of Nassi-Shneiderman diagrams, which represent structured programs with conditional statements and while loops [12]. Figure 8 shows such diagrams. Each diagram can be modelled by a graph where statement, condition, and while blocks are represented by edges of type *stmt*, *cond*, and *while*, respectively. Diagram D_1 in Fig. 8, for instance, is represented by a graph $cond^{abcd} stmt^{cefg} stmt^{edgh}$. The language of all *Nassi-Shneiderman graphs* is defined by an HR grammar with the following rules:

$$\begin{aligned}
 S^\varepsilon &\rightarrow NSD^{xyuv} \\
 NSD^{xyuv} &\rightarrow NSD^{xyrs} Stmt^{rsuv} \mid Stmt^{xyuv} \\
 Stmt^{xyuv} &\rightarrow stmt^{xyuv} \mid cond^{xyrs} NSD^{rmun} NSD^{msnv} \mid while^{xyrsut} NSD^{rstv}
 \end{aligned}$$

We use the shorthand notation $L \rightarrow R_1 \mid R_2$ to represent rules $L \rightarrow R_1$ and $L \rightarrow R_2$ with the same left-hand side.

Runtime of the different parsers has been measured for Nassi-Shneiderman graphs D_n with varying values n . Figure 8 recursively defines these graphs D_i for $i = 1, 2, 3, \dots$ and also shows D_3 as an example. Each diagram D_i consists of $2 + 6i$ nodes and $3i$ edges.

Figure 9a shows the runtime of the PSR and the PTD parser for graphs D_n with n being shown on the x -axis and the runtime in milliseconds on the y -axis. Figure 9b shows the corresponding diagram for the CYK parser. The PSR parser and the CYK parser have been generated from the HR grammar presented above. For generating the PTD parser, a slightly modified grammar with *merging rules* [4] had to be used because the presented grammar is not PTD.

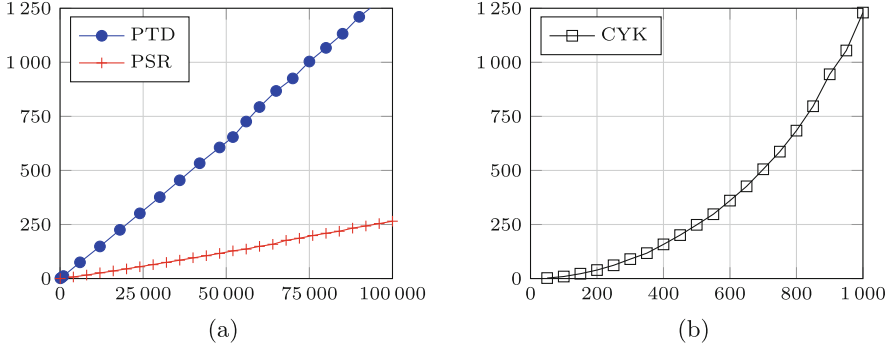


Fig. 9. Runtime (in milliseconds) of the PSR as well as the PTD parser (a) and the CYK parser (b) for Nassi-Shneiderman graphs built as shown in Fig. 8. Note that the scales in (a) and (b) differ.

Note that the runtime of the PSR parser and the slower PTD parser is linear in the size of the input graph whereas the runtime of the CYK parser is not linear. Note again that the scales in the diagrams shown in Fig. 9a and b differ and that PTD parsing and, in particular, PSR parsing is, by several orders of magnitude, faster than CYK parsing. For instance, the CYK parser needs 1.2 s to parse D_{1000} whereas the PTD parser needs just 12ms, and the PSR parser just 1.0 ms.

The PSR parsers for triangle and Nassi-Shneiderman graphs make use of determining nodes and, therefore, do not require hash tables to obtain linear parsing time. In order to demonstrate the speed-up produced by hash tables, we constructed an HR grammar (see Fig. 10), called *blowball grammar* because of the shapes of its graphs. Its PSR parser must perform some edge look-ups without determining nodes. *Grappa* has been used to generate two versions of a PSR parser: Version *PSR (hash)* uses hash tables to speed up these edge look-ups, whereas version *PSR (no hash)* iterates over lists of candidates instead. Moreover, a PTD and a CYK parser have been generated. For the experiments, we considered blowball graphs B_n , $n \geq 1$, like B_{10} shown in Fig. 11: B_n consists of n pair edges (represented by arrows in Fig. 11), one in the center and the rest forming stars where the number of edges in each star is as close to the number of stars as possible. Runtime of the different parsers has been measured for these graphs B_n with varying values n . Figure 12a shows the results of the two PSR parsers. The *PSR (no hash)* parser has quadratic parsing time and is much slower than the *PSR (hash)* parser with linear parsing time. For instance, *PSR (no hash)* needs 360 ms to parse B_{10000} , whereas *PSR (hash)* needs just 10ms. Parsing time of the PTD parser is similar to the *PSR (no hash)* parser and is not shown here. Figure 12b shows the results of the CYK parser, which is again by several orders of magnitude slower than the other parsers. For instance, the CYK parser needs 1.6 s to parse B_{16} whereas the PTD parser needs just $9 \mu\text{s}$, and the PSR parsers (both versions) just $5 \mu\text{s}$.

$$\begin{aligned}
 S^\varepsilon &\rightarrow Tree^{xy} \\
 Tree^{xy} &\rightarrow pair^{xy} \mid \\
 &\quad Child^{xyu} Tree^{xy} \\
 Child^{xyu} &\rightarrow edge^{xyuv} Tree^{uv} Next^{xyu} \\
 Next^{xyu} &\rightarrow Child^{xyu} \mid \\
 &\quad \varepsilon
 \end{aligned}$$

Fig. 10. Blowball graph grammar.

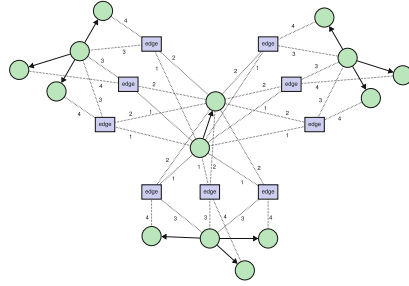


Fig. 11. Blowball graph B_{10} .

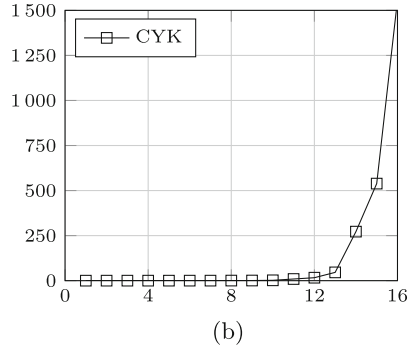
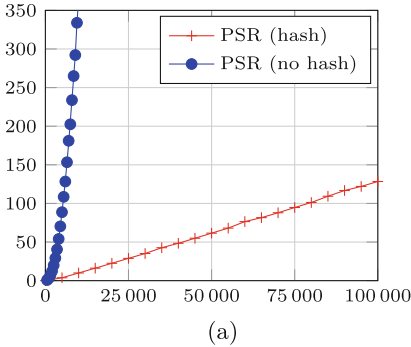


Fig. 12. Runtime (in milliseconds) of the PSR parser (a) with hash tables (faster) and without hash tables (slower) and the CYK parser (b) for blowball graphs B_n . Note that the scales in (a) and (b) differ.

7 Conclusions

We have described the implementation of efficient predictive shift-reduce parsers for HR grammars. Such PSR parsers are too complicated to be implemented by hand. The *Grappa* parser generator has been implemented to assist users in two ways: The generator first checks PSR-parsability of an HR grammar, and it generates a PSR parser only if the grammar is PSR-parsable. Measurements for the generated parsers confirm that they run in linear time, as postulated in [6]. In that paper, we have established some relationship between HR grammars generating string graphs: PSR parsing turned out to be a true extension of De Remer’s *SLR(1)* parsing. Unfortunately, Theorem 1 in that paper, stating that PTD string graph grammars are PSR, is wrong. We are about to devise a restricted version of PTD so that this theorem holds. This does not affect the results of this paper.

Earlier, now abandoned work on predictive graph parsers [9,11] has been based on fairly restricted subclasses of node replacement grammars [8] and on edge precedence relations.

Like PTD parsing, PSR parsing can be lifted to contextual HR grammars [2, 3], a class of graph grammars that is more relevant for the practical definition of graph languages. This remains as part of future work. Moreover, it might be worthwhile to extend PSR to the more powerful Earley-style parsers that use a more general kind of control automaton, and pursue several goals in parallel [7].

References

1. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach. *Encyclopedia of Mathematics and its Applications*, vol. 138. Cambridge University Press, Cambridge (2012)
2. Drewes, F., Hoffmann, B.: Contextual hyperedge replacement. *Acta Inform.* **52**, 497–524 (2015)
3. Drewes, F., Hoffmann, B., Minas, M.: Contextual hyperedge replacement. In: Schürr, A., Varró, D., Varró, G. (eds.) *AGTIVE 2011*. LNCS, vol. 7233, pp. 182–197. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34176-2_16
4. Drewes, F., Hoffmann, B., Minas, M.: Predictive top-down parsing for hyperedge replacement grammars. In: Parisi-Presicce, F., Westfechtel, B. (eds.) *ICGT 2015*. LNCS, vol. 9151, pp. 19–34. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_2
5. Drewes, F., Hoffmann, B., Minas, M.: Approximating Parikh images for generating deterministic graph parsers. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) *STAF 2016*. LNCS, vol. 9946, pp. 112–128. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_9
6. Drewes, F., Hoffmann, B., Minas, M.: Predictive shift-reduce parsing for hyperedge replacement grammars. In: de Lara, J., Plump, D. (eds.) *ICGT 2017*. LNCS, vol. 10373, pp. 106–122. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_7
7. Earley, J., Sturgis, H.: A formalism for translator interactions. *Commun. ACM* **13**(10), 607–617 (1970)
8. Engelfriet, J., Rozenberg, G.: Node replacement graph grammars. In: Rozenberg, G. (ed.) *Handbook of Graph Grammars and Computing by Graph Transformation*. Foundations, vol. I, Chap. 1, pp. 1–94. World Scientific, Singapore (1997)
9. Franck, R.: A class of linearly parsable graph grammars. *Acta Inform.* **10**(2), 175–201 (1978)
10. Habel, A.: Hyperedge Replacement: Grammars and Languages. LNCS, vol. 643. Springer, Heidelberg (1992). <https://doi.org/10.1007/BFb0013875>
11. Kaul, M.: Practical applications of precedence graph grammars. In: Ehrig, H., Nagl, M., Rozenberg, G., Rosenfeld, A. (eds.) *Graph Grammars 1986*. LNCS, vol. 291, pp. 326–342. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18771-5_62
12. Minas, M.: Diagram editing with hypergraph parser support. In: *Proceedings of 1997 IEEE Symposium on Visual Languages (VL 1997)*, Capri, Italy, pp. 226–233. IEEE Computer Society Press (1997)
13. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. *Sci. Comput. Program.* **44**(2), 157–180 (2002)

Checking Graph Programs for Confluence

Ivaylo Hristakiev^(✉) and Detlef Plump

University of York, York, UK
ivaylo.hristakiev@gmail.com

Abstract. We present a method for statically verifying confluence (functional behaviour) of terminating sets of rules in the graph programming language GP 2, which is undecidable in general. In contrast to other work about attributed graph transformation, we do not impose syntactic restrictions on the rules except for left-linearity. Our checking method relies on constructing the symbolic critical pairs of a rule set using an E-unification algorithm and subsequently checking whether all pairs are strongly joinable with symbolic derivations. The correctness of this method is a consequence of the main technical result of this paper, viz. that a set of left-linear attributed rules is locally confluent if all symbolic critical pairs are strongly joinable, and our previous results on the completeness and finiteness of the set of symbolic critical pairs. We also show that for checking strong joinability, it is not necessary to compute all graphs derivable from a critical pair. Instead, it suffices to focus on the pair's persistent reducts. In a case study, we use our method to verify the confluence of a graph program that calculates shortest distances.

1 Introduction

A common programming pattern in the graph programming language GP 2 [18] is to apply a set of attributed graph transformation rules as long as possible. To execute a set of rules $\{r_1, \dots, r_n\}$ for as long as possible on a host graph, in each iteration an applicable rule is selected and applied. As rule selection and rule matching are non-deterministic, different graphs may result from such an iteration. Thus, if the programmer wants the loop to implement a function, a static analysis that establishes or refutes functional behaviour would be desirable.

GP 2 is based on the double-pushout approach to graph transformation with relabelling [6]. Programs can perform computations on labels by using rules labelled with expressions (also known as attributed rules). GP 2's label algebra consists of integers, character strings, and heterogeneous lists of integers and strings. Rule application can be seen as a two-stage process where rules are first instantiated, by replacing variables with values and evaluating the resulting expressions, and then applied as usual. Hence rules are actually rule schemata.

Conventional confluence analysis in the double-pushout approach to graph transformation is based on *critical pairs*, which represent conflicts in minimal context [4, 17]. A conflict between two rule applications arises, roughly speaking, when one of the steps cannot be applied to the result of the other. In the presence

of termination, one can check if all critical pairs are *strongly joinable*, and thus establish that the set of transformation rules is confluent.

In our previous paper [12], we developed the notion of symbolic critical pairs for GP 2 rule schemata which are minimal conflicting derivations, labelled with expressions. The set of such pairs is finite and complete, in the sense that they represent all possible conflicts that may arise during computation. Furthermore, we gave an algorithm for constructing the set of symbolic critical pairs induced by a set of schemata, which uses our E-unification algorithm of [9]. The approach does not place severe restrictions on labels appearing in rules, as the attributed setting of [3]. What remains to be shown is how to use such critical pairs in the context of confluence analysis.

In this paper, we present our method for statically verifying confluence of terminating sets of GP 2 rules. We introduce a notion of symbolic rewriting that allows us to rewrite the graphs of critical pairs, and show how it is used for confluence analysis. The correctness of our analysis is a consequence of the main technical result of this paper, namely that a set of left-linear attributed rules is locally confluent if all symbolic critical pairs are strongly joinable. We also show that for checking strong joinability, it is not necessary to compute all graphs derivable from a critical pair but it suffices to focus on the pair's persistent reducts. In a case study, we use our method to verify the confluence of a graph program that calculates shortest distances.

We assume the reader to be familiar with basic notions of the double-pushout approach to graph transformation (see [3]). The long version of this paper [11] contains the technical proofs together with the full shortest distances case study.

2 Graphs and Graph Programs

In this section, we present the approach of GP 2 [1, 18], a domain-specific language for rule-based graph manipulation. The principal programming units of GP 2 are rule schemata $\langle L \leftarrow K \rightarrow R \rangle$ labelled with expressions that operate on host graphs (or input graphs) labelled with concrete values. The language also allows to combine schemata into programs. The definition of GP 2's latest version, together with a formal operational semantics, can be found in [1]. We start by recalling the basic notions of partially labelled graphs and their morphisms.

Labelled Graphs. A (partially) labelled graph G consists of finite sets V_G and E_G of nodes and edges (graph items for short), source and target functions for edges $s_G, t_G: E_G \rightarrow V_G$, and a partial node/edge labelling function $l_G: V_G + E_G \rightarrow \mathcal{L}$ over a (possibly infinite) label set \mathcal{L} . Given an item x , $l_G(x) = \perp$ expresses that $l_G(x)$ is undefined. The graph G is totally labelled if l_G is a total function. The classes of partially and totally labelled graphs over \mathcal{L} are denoted as $\mathcal{G}_\perp(\mathcal{L})$ and $\mathcal{G}(\mathcal{L})$.

A premorphism $g: G \rightarrow H$ consists of two functions $g_V: V_G \rightarrow V_H$ and $g_E: E_G \rightarrow E_H$ that preserve sources and targets, and is a graph morphism if it preserves labels of graph items, that is $l_H(g(x)) = l_G(x)$ for all $x \in \text{Dom}(l_G)$.

A morphism g preserves undefinedness if it maps unlabelled items of G to unlabelled items in H . A morphism g is an inclusion if $g(x) = x$ for all items x in G . Note that inclusions need not preserve undefinedness. A morphism g is injective (surjective) if g_V and g_E are injective (surjective), and is an isomorphism (denoted by \cong) if it is injective, surjective and preserves undefinedness. The class of injective label preserving morphisms is denoted as \mathcal{M} for short, and the class of injective label and undefinedness preserving morphisms is denoted as \mathcal{N} .

Partially labelled graphs and label-preserving morphisms constitute a category [6, 7]. Composition of morphisms is defined componentwise. In this category not all pushouts exist, and not all pushouts along \mathcal{M} -morphisms are natural¹.

GP 2 Labels. The types `int` and `string` represent integers and character strings. The type `atom` is the union of `int` and `string`, and `list` represents lists of atoms. Given lists l_1 and l_2 , we write $l_1 : l_2$ for the concatenation of l_1 and l_2 (not to be confused with the list-cons operator in Haskell). Atoms are lists of length one. The empty list is denoted by `empty`. Variables may appear in labels in rules and are typed over the above categories. Labels in rule schemata are built up from constant values, variables, and operators - the standard arithmetic operators for integer expressions (including the unary minus), string/list concatenation for string/list expressions, `indegree` and `outdegree` operators for nodes. In pictures of graphs, graph items that are shown without a label are implicitly labelled with the empty list, while unlabelled items in interfaces are labelled with \perp to avoid confusion.

Additionally, a label may contain an optional *mark* which is represented graphically as a colour. For example, the grey node of the rule schema `init` in Fig. 3 has the label $(x : 0, \text{grey})$.

Rule Schemata and Direct Derivations. In order to compute with labels, it is necessary that graph items can be relabelled during computation. The double-pushout approach with partially labelled interface graphs is used as a formal basis [6]. This approach is also the foundation of GP 2.

To apply a rule schema to a graph, the schema is first instantiated by evaluating its labels according to some assignment α . An assignment α maps each variable occurring in a given schema to a value in GP 2's label algebra. Its unique extension α^* evaluates the schema's label expressions according to α . For short, we denote GP 2's label algebra as A . Its corresponding term algebra over the same signature is denoted as $T(X)$, and its terms are used as graph labels in rule schemata. Here X is the set of variables occurring in schemata. A substitution σ maps variables to terms. To avoid an inflation of symbols, we sometimes equate A or $T(X)$ with the union of its carrier sets.

A GP 2 rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of two inclusions $K \rightarrow L$ and $K \rightarrow R$ such that L and R are graphs in $\mathcal{G}(T(X))$ and K is a graph in $\mathcal{G}_\perp(T(X))$. Consider a graph G in $\mathcal{G}_\perp(T(X))$ and an assignment $\alpha : X \rightarrow A$. The instance G^α is the graph in $\mathcal{G}_\perp(A)$ obtained from G by replacing each label

¹ A pushout is natural if it is also a pullback.

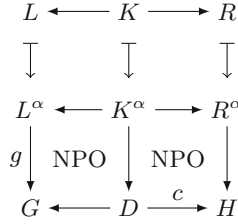


Fig. 1. A direct derivation

l with $\alpha^*(l)$. The instance of a rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ is the rule $r^\alpha = \langle L^\alpha \leftarrow K^\alpha \rightarrow R^\alpha \rangle$.

A direct derivation via rule schema r and assignment α between host graphs $G, H \in \mathcal{G}(A)$ consists of two natural pushouts as in Fig. 1. We denote such a derivation by $G \xrightarrow{r, g, \alpha} H$. Later we will allow for rules to be applied to graphs in $\mathcal{G}(T(X))$. In [6] it is shown that in case the interface graph K has unlabelled items, their images in the intermediate graph D are also unlabelled. By [6, Theorem 1], given a rule r and a graph G together with an injective match $g: L \rightarrow G$ satisfying the *dangling condition* (no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$), there exists a unique double natural pushout as in Fig. 1. The *track* morphism allows to “follow items through derivations”: $track_{G \Rightarrow H}: G \rightarrow H$ is the partial premorphism defined by $track(x) = \mathbf{if } x \in D \mathbf{ then } c(x) \mathbf{ else } \mathbf{undefined}$ where c is the inclusion $D \rightarrow H$, and $track_{G \Rightarrow^* H}$ of an arbitrary-length derivation is the composition of partial premorphisms. Note $track$ may not preserve labels due to relabelling.

For an example rule schema and graph program, see the start of Sect. 5. When a rule schema is graphically declared as done in Fig. 3, the interface is represented by the node numbers in L and R . Nodes without numbers in L are to be deleted and nodes without numbers in R are to be created. All variables in R have to occur in L so that for a given match of L in a host graph, applying the rule schema produces a graph that is unique up to isomorphism.

Program Constructs. The language GP 2 offers several operators for combining programs - the postfix operator ‘!’ iterates a program as long as possible; sequential composition ‘P; Q’; a rule set $\{r_1, \dots, r_n\}$ tries to non-deterministically apply any of the schemata (failing if none are applicable); **if** C **then** P **else** Q allows for conditional branching (C, P, Q are arbitrary programs) meaning that if the program C succeeds on a copy of the input graph then P is executed on the original, if C fails then Q is executed on the original input graph.

Confluence. A set of rule schemata \mathcal{R} is *confluent* if for all graphs G, H_1, H_2 with derivations $H_1 \leftarrow_{\mathcal{R}}^* G \Rightarrow_{\mathcal{R}}^* H_2$ there is a graph M with $H_1 \Rightarrow_{\mathcal{R}}^* M \leftarrow_{\mathcal{R}}^* H_2$. \mathcal{R} is *locally confluent* if this property holds for direct derivations $H_1 \leftarrow_{\mathcal{R}} G \Rightarrow_{\mathcal{R}} H_2$. Finally, \mathcal{R} is *terminating* if there is no infinite sequence $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \dots$ of direct derivations.

Assumptions. Our previous results on critical pairs [12] involve several restrictions. Firstly, the proper treatment of GP 2 *conditional* rule schemata requires extra results about shifting of conditions along morphisms and rules, which we do not treat here formally. We give an intuition of how to deal with conditions on labels in our shortest distances case study in Sect. 5. Secondly, we assume rule schemata to be *left-linear*, meaning no list variables are shared between items in schemata. This ensures that overlapping graphs with expressions results in a finite set of critical pairs. Thirdly, we allow interfaces in rules to contain edges and labels, which is a deviation from the GP 2 convention of unlabelled node-only interfaces. This reduces the number of potential conflicts.

3 Symbolic Critical Pairs

Confluence [17] is a property of a rewrite system that ensures that any pair of derivations on the same host graph can be joined again thus leading to the same result, and is an important property for many kinds of graph transformation systems. A confluent computation is globally deterministic despite local non-determinism. The main technique for confluence analysis is based on the study of critical pairs which are conflicts in minimal context.

In our previous paper [12], we defined critical pairs for GP 2 that are labelled with expressions rather than from a concrete data domain. Each symbolic critical pair represents a possibly infinite set of conflicting host graph derivations. Hence, it is possible to foresee each conflict by computing all critical pairs statically. What is special about our critical pairs is that they show the conflict in the most abstract way. Informally, a pair of derivations $T_1 \xleftarrow{r_1, m_1, \sigma} S \xrightarrow{r_2, m_2, \sigma} T_2$ between graphs labelled with expressions is a symbolic critical pair if it is in conflict and minimal. Two direct derivations are independent if neither derivation deletes or relabels any common item, and in conflict if otherwise. Independent derivations have the Church-Rosser property as shown in [10] for the case of rule schemata.

Minimality of a pair of derivations means the pair of matches (m_1, m_2) is jointly surjective – the graph S can be considered as a suitable overlap of L_1^σ and L_2^σ . Formally, overlapping graphs L_1 and L_2 via premorphisms $m_1 : L_1 \rightarrow S, m_2 : L_2 \rightarrow S$ induces a system of unification problems:

$$\text{EQ}(L_1 \xrightarrow{m_1} S \xleftarrow{m_2} L_2) = \{l_{L_1}(a) \stackrel{?}{=} l_{L_2}(b) \mid (a, b) \in L_1 \times L_2 \text{ with } m_1(a) = m_2(b)\}$$

The substitution σ above is taken from the complete set of unifiers of the above system computed by our unification algorithm of [9], and is used to instantiate the schemata to a critical pair.

Definition 1 (Symbolic Critical Pair [12]). *A symbolic critical pair is a pair of direct derivations $T_1 \xleftarrow{r_1, m_1, \sigma} S \xrightarrow{r_2, m_2, \sigma} T_2$ on graphs labelled with expressions such that:*

- (1) σ is a substitution from a complete set of unifiers of $(\text{EQ}(L_1 \xrightarrow{m_1} S \xleftarrow{m_2} L_2))$ where L_1 and L_2 are the left-hand graphs of r_1 and r_2 , m_1 and m_2 are premorphisms, and

- (2) the pair of derivations is in conflict, and
 (3) $S = m_1(L_1^\sigma) \cup m_2(L_2^\sigma)$, meaning S is minimal, and
 (4) $r_1^\sigma = r_2^\sigma$ implies $m_1 \neq m_2$. □

We assume that the variables occurring in different rule schemata are distinct, which can always be achieved by variable renaming. The derivations have to be via left-linear rule schemata in order for our unification algorithm to work on the systems of unification problems EQ. For example critical pairs, see Sect. 5.

Properties of Symbolic Critical Pairs. Below we present the properties of critical pairs as proven in [12]. Symbolic critical pairs are complete, meaning that each pair of conflicting direct derivations is an instance of a symbolic critical pair. Additionally, the set of symbolic critical pairs is finite.

Theorem 1 (Completeness and Finiteness of Critical Pairs [12]). *For each pair of conflicting rule schema applications $H_1 \xleftarrow{r_1, m_1, \alpha} G \xrightarrow{r_2, m_2, \alpha} H_2$ between left-linear schemata r_1 and r_2 there exists a symbolic critical pair $T_1 \xleftarrow{r_1} S \xrightarrow{r_2} T_2$ with the (extension) diagrams (1) and (2) between $H_1 \leftarrow G \Rightarrow H_2$ and an instance of $T_1 \leftarrow S \Rightarrow T_2$. Moreover, the set of symbolic critical pairs induced by r_1 and r_2 is finite.*

$$\begin{array}{ccccc}
 T_1 & \leftarrow & S & \Rightarrow & T_2 \\
 \downarrow & & \downarrow & & \downarrow \\
 Q_1 & \leftarrow & P & \Rightarrow & Q_2 \\
 \downarrow & (1) & \downarrow & (2) & \downarrow \\
 H_1 & \leftarrow & G & \Rightarrow & H_2
 \end{array}$$

4 Symbolic Rewriting and Joinability

Symbolic critical pairs consist of graphs labelled with expressions. This is necessary for making the set of critical pairs finite as GP 2's infinite label algebra induces an infinite set of conflicts at the instance (host) level. How to rewrite such graphs is what we focus on next as the current GP 2 framework only defines rewriting of host graphs.

In this section we propose *symbolic rewriting* of GP 2 graphs to overcome the above limitation. This allows for the representation of multiple host graph direct derivations. The overall aim is to use symbolic rewriting for establishing the (strong) joinability of critical pairs.

4.1 Symbolic Rewriting

Informally, symbolic rewriting introduces a relation on rule graphs (\Rightarrow) where matching is done by treating variables as typed symbols/constants. What is special in our setting is that rules cannot introduce new variables. Furthermore, since application conditions cannot usually be checked for satisfiability as values for variables are not known at analysis time, they are only recorded as assumptions to be resolved later. This type of rewriting is very similar to symbolic graph transformation, e.g. as in [16].

Isomorphism and Label Equivalence. Since we now consider graphs in $\mathcal{G}(T(X))$ involving GP 2 label expressions, we relax the definition of isomorphism presented in Sect. 2 by replacing label equality with *equivalence*. Furthermore, since graph isomorphism is central to the discussion of joinability of critical pairs, we define how isomorphism relates to a critical pair’s set of persistent nodes.

Two graphs $G, H \in \mathcal{G}(T(X))$ are E-isomorphic (denoted by $G \cong_E H$) if there exists a bijective premorphism $i : G \rightarrow H$ such that $l_H(i(x)) \approx_E l_G(x)$ for all items of G . Here \approx_E is the equivalence relation on GP2 expressions given by all the equations valid in GP 2’s label algebra of integer arithmetic and list/string concatenation.

For an example of why a more general notion of isomorphism is needed, consider the schemata $r_1: \boxed{m:n} \Rightarrow \boxed{m+n}$ and $r_2: \boxed{m:n} \Rightarrow \boxed{n+m}$ which both match a node labelled with a list of two integers (m and n) but relabel the node to (syntactically) different expressions. The derivations $\boxed{n+m} \xleftarrow{r_1} \boxed{m:n} \xrightarrow{r_2} \boxed{m+n}$ represent a symbolic critical pair (conflict due to relabelling). The resulting graphs are normal forms, and isomorphic only if one considers the commutativity of addition.

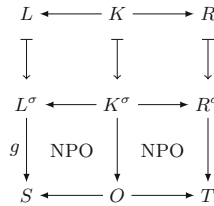


Fig. 2. Symbolic direct derivation.

Symbolic Derivation. The essence of symbolic rewriting is to allow rule schemata to be applied to graphs labelled with expressions, i.e. graphs in $\mathcal{G}(T(X))$. In the terminology of Sect. 2, assignments become substitutions $\sigma : X \rightarrow T(X)$. We call such a derivation *symbolic*. For example, the critical pairs in Fig. 4 involve such symbolic derivations. Operationally, constructing symbolic derivations involves obtaining a substitution σ for the variables of L given a premorphism $L \rightarrow S$, and then constructing a direct derivation with relabelling as in Sect. 2.

Definition 2 (Symbolic direct derivation). A symbolic direct derivation via rule schema r , substitution σ between graphs $S, T \in \mathcal{G}(T(X))$ consists of two natural pushouts via match $g : L^\sigma \rightarrow S$ as in Fig. 2.

We denote symbolic derivations by $S \xRightarrow{r,g,\sigma} T$. Note that variables occurring in S cannot be modified. This kind of rewriting is incomplete in that not all host graph derivations can be represented by symbolic derivations.

Symbolic derivations allow for the representation of multiple host graph direct derivations, and can be seen as transformations of specifications. The proposition below states that the application of symbolic rule schema coincides, in some sense, with respect to the host graph derivations it represents. For its proof see [11].

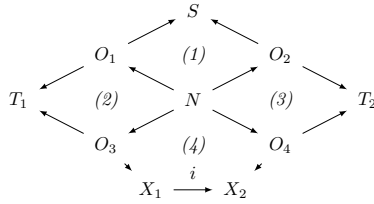
Lemma 1 (Soundness of symbolic rewriting). *For each symbolic derivation $S \xrightarrow{r,g,\sigma} T$ and each host graph $G = S^\lambda$ and assignment λ , there exists a direct derivation $G \xrightarrow{r,g,\alpha} H$ where $H = T^\lambda$ and $\alpha = \lambda \circ \sigma$.*

4.2 Joinability

Confluence analysis is based on the joinability of critical pairs. Informally, a symbolic critical pair $T_1 \Leftarrow S \Rightarrow T_2$ is *joinable* if there exist symbolic derivations from T_1 and T_2 to a common graph. However, it is known that joinability of all critical pairs is not sufficient to prove local confluence [17]. Instead, one needs to consider a slightly stronger notion called *strong* joinability that requires a set of *persistent* nodes in a critical pair to be preserved by the joining derivations. The set of persistent items of a critical pair consists of all nodes in S that are preserved by both steps, and are defined in terms of the pullback N of the intermediate graphs O_1 and O_2 of the critical pair².

Definition 3 (Strong joinability). *A symbolic critical pair $T_1 \Leftarrow S \Rightarrow T_2$ is strongly joinable if we have the following:*

1. *joinability: there exist symbolic derivations $T_1 \Rightarrow^* X_1 \cong_E X_2 \Leftarrow^* T_2$ where $i : X_1 \rightarrow X_2$ is an E -isomorphism.*
2. *strictness: let N be the pullback object of $O_1 \rightarrow S \leftarrow O_2$ (1). Then there exist morphisms $N \rightarrow O_3$ and $N \rightarrow O_4$ such that the squares (2), (3) and (4) commute:*



The strictness condition can be restated in terms of the track morphisms of the joining derivations, as in [17]: the track morphisms $track_{S \Rightarrow T_1 \Rightarrow^* X_1}$ and $track_{S \Rightarrow T_2 \Rightarrow^* X_2}$ are defined and commute on the persistent items of the critical pair, i.e. $i(track_{S \Rightarrow T_1 \Rightarrow^* X_1}(x)) = track_{S \Rightarrow T_2 \Rightarrow^* X_2}(x)$ for each $x \in N$. See the first author's thesis for a proof of equivalence [8]. The graphs O_3 and O_4 in the above definition are the derived spans of the joining derivations as the joining derivations are of arbitrary length, e.g. see [3, 11].

² For the construction of pullbacks over partially labelled graphs see [7, Sect. 4].

Lemma 2 (Joinability preservation). *If a symbolic critical pair $T_1 \Leftarrow S \Rightarrow T_2$ is strongly joinable, then each of its instances according to some assignment λ ($T_1^\lambda \Leftarrow S^\lambda \Rightarrow T_2^\lambda$) is also strongly joinable.*

Here we consider the critical pair instances to be critical pairs over the rule instances in their own right. For proof(s), see [11].

5 Case Study: Shortest Distances

The shortest distances problem is about calculating the paths between a given node (the *source* node) and all other nodes in a graph such that the sum of the edge weights on each path is minimized. The Bellman–Ford algorithm [2] is an algorithm that solves that problem. It is based on *relaxation* in which the current distance to a node is gradually replaced by more accurate values until eventually reaching the optimal solution. An assumption made is that there is no *negative cycle* (a cycle whose edge weights sum to a negative value) that is reachable from the source, in which case there is no shortest path.

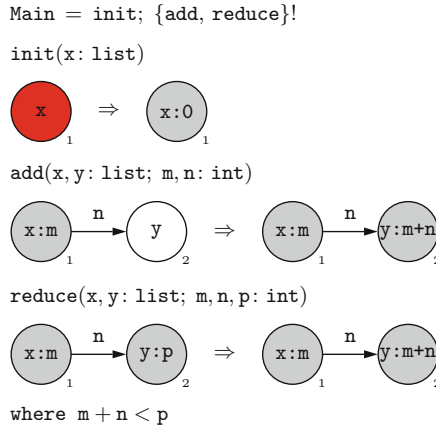


Fig. 3. Shortest distances program

GP 2 Implementation. A GP 2 program that implements the above algorithm is shown in Fig. 3. Distances from the source node are recorded by appending the distance value to each node’s label. Nodes *marks* are used: the source node is red, visited nodes are gray, and unvisited nodes are unmarked. Given an input graph G with a unique source node and no negative cycle, the program initializes the distance of the source node to 0. The **add** rule explores the unvisited neighbours of any visited nodes, assigns them a tentative distance and marks them as visited to avoid non-termination. The **reduce** rule finds occurrences of visited nodes whose current distance is higher than alternative distances, i.e. only when the application condition ($m + n < p$) is satisfied by the schema instantiation. The program terminates when neither **add** or **reduce** rules can be further applied.

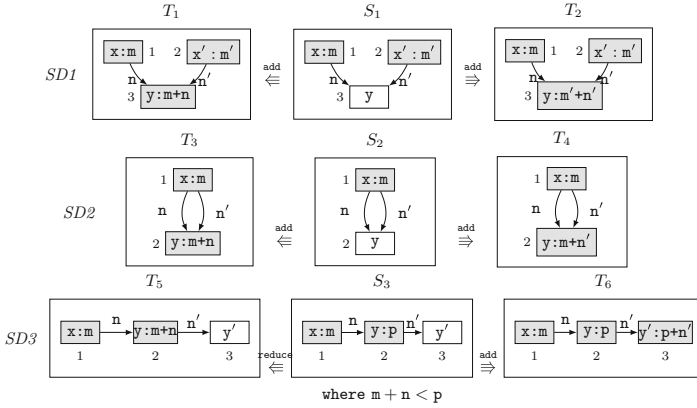


Fig. 4. Shortest distances critical pairs involving `add`.

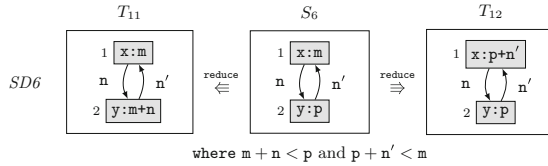


Fig. 5. A `reduce` critical pair requiring a semantic joinability argument.

However, since rule application is non-deterministic, different graphs may result from a program execution. The above algorithm is correct only if the loop `{add,reduce}!` is confluent. In the absence of a full program verification, a programmer may want to check that this loop indeed returns unique results.

Critical Pairs. There are 7 critical pairs in total for the above program: two between `add` with itself (SD1/2), one between `add` and `reduce` (SD3), and four between `reduce` with itself (SD4-7). Figure 4 gives the first three critical pairs, whereas the `reduce` critical pairs are very similar to those and are omitted for space reasons. The only interesting `reduce` critical pair involves a 2-cycle where either node gets its distance updated by `reduce` and is given in Fig. 5. All of the conflicts are due to relabelling of a common node. Note that due to the semantics of GP 2 marks (marked cannot match unmarked), other conflicts are not possible. Variables have been renamed where necessary. The persistent items of all critical pairs are the graph items of `S` since the rules do not delete any graph items, and the common node of each critical pair that gets relabelled (e.g. node 3 in SD1) does not have a label in the pullback graph `N` of Definition 3.

The critical pairs SD1/2 are between the rule `add` with itself where an unvisited node can get initialized with different distance values, either from 2 neighbouring nodes or from the same node but different (parallel) edges. In SD3 the distance of a node in a path is used in different ways: either to initialize the distance of a

neighbouring node (via **add**), or to have its own distance updated (via **reduce**). Note that the application condition is recorded as part of the critical pair. The critical pairs SD4/5 represent a conflict of **reduce** with itself where a node may get different updated distance values depending on which path is chosen, similar to SD1/2. SD6 involves a 2-cycle where either node gets its distance updated by **reduce**. SD7 involves a sequence of three nodes, similar to SD3.

Joinability Analysis. Due to space limitations, here we only give a top-level explanation of why each of the critical pairs are strongly joinable. See [11] for the full details. The result of the analysis is that all critical pairs are strongly joinable except the 2-cycle critical pair SD6 whose label condition is unsatisfiable assuming non-negative cycles and the semantic argument that both schemata do not modify edge labels. (Without using this information, the critical pair is not joinable.) Hence the loop $\{\mathbf{add}, \mathbf{reduce}\}!$ is confluent.

```

(define-fun T1_T2() Bool
  (forall ((m1 Int) (m2 Int)
           (n1 Int) (n2 Int))
    (= (+ m1 n1) (+ m2 n2)))
(assert T1_T2)

(define-fun T77_T888 () Bool
  (forall ((m Int) (p Int)
           (n1 Int) (n2 Int))
    (=> (< (+ m n1) p)
        (< (+ m n1 n2) (+ p n2))))
(assert T77_T888)

```

(a) Label equivalence example for SD1.

(b) Implication checking for SD3.

Fig. 6. Z3 code for label equivalence analysis of shortest distances.

An interesting practical aspect of joinability is that it involves, in most cases, checking label equivalences for validity. (We check for validity rather than satisfiability since we need that all instances of a strongly joinable critical pair to be strongly joinable rather than at least one.) For this purpose, we use the SMT solver Z3 [15]. It provides support for (linear) integer arithmetic, arrays, bit vectors, quantifiers, implications, etc.

For the critical pair SD1, the result graphs T_1 and T_2 are isomorphic only if the label equivalence $m + n = m' + n'$ is valid, which it is not (encoded as a **forall** expression in Fig. 6a where variables have been renamed). The analysis proceeds by applying **reduce** to both T_1 and T_2 , and the semantics of the **reduce** condition (containing comparison of integer expressions) guarantees a strong isomorphism between the results. Note that **reduce** is necessary for the joining derivations, meaning the rule **add** is not confluent on its own. The analysis of SD2 proceeds in a similar way as SD1 with the same conclusion. For SD3, one needs to check *implications* between conditions to ensure strong joinability between a pair of derivable graphs. An implication that shows up during the analysis is shown in Fig. 6b which Z3 reports to be valid. Therefore the critical pair is strongly joinable. The analysis for the critical pairs SD4/5 is the same as for SD1/2.

The critical pair SD6 is different than the rest - its label condition is satisfiable only when the sum of the edge labels is negative ($\mathbf{n} + \mathbf{n}' < 0$), which is not

possible under the assumption of no negative cycles and the observation that no rules modify edge labels. Without this semantic information, it is possible to instantiate the critical pair to a concrete graph with non-isomorphic normal forms, and thus obtain an example of non-confluence.

6 Local Confluence

In this section we present the Local Confluence Theorem which establishes the local confluence of \mathcal{R} if all symbolic critical pairs are strongly joinable. It was first shown in [17] for the (hyper)graph case and later extended to (weak) adhesive categories in [5]. We also discuss our method for confluence checking based on symbolic critical pairs.

Theorem 2 (Local Confluence Theorem). *A set \mathcal{R} of left-linear rule schemata is locally confluent if all of its symbolic critical pairs are strongly joinable.*

The full proof closely follows the Local Confluence Theorem proof of [3, Theorem 6.28], which requires several properties of \mathcal{M} and \mathcal{N} established in [7]. Due to space limitations, here we give only an outline containing the important steps.

Proof Outline. For a given pair of direct derivations $H_1 \xrightarrow{r_1, m_1, \alpha} G \xrightarrow{r_2, m_2, \alpha} H_2$, we have to show the existence of derivations $H_1 \Rightarrow^*_{\mathcal{R}} X'_1 \cong X''_1 \Leftarrow^*_{\mathcal{R}} H_2$ as the outer part of Fig. 7a. If the given pair is independent, this follows from the Church-Rosser Theorem for rule schemata [10]. If the given pair is in conflict,

Theorem 1 implies the existence of a symbolic critical pair $T_1 \xrightarrow{r_1, m'_1, \sigma} S \xrightarrow{r_2, m'_2, \sigma} T_2$

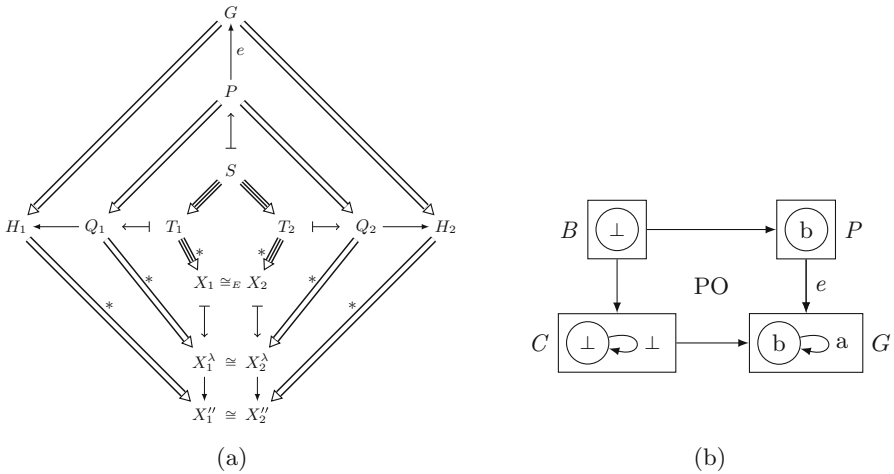


Fig. 7. Local Confluence Diagram (a) and Initial pushout in \mathcal{G}_\perp (b).

with extension diagrams as in the upper half of Fig. 7a involving an instance of the critical pair, and the extension morphism $e : P \rightarrow G \in \mathcal{N}$. By assumption, this critical pair is strongly joinable. By Lemma 2, the critical pair instance $Q_1 \leftarrow P \Rightarrow Q_2$ is also strongly joinable, leading to derivations $P \Rightarrow Q_1 \Rightarrow^* X_1^\lambda \cong X_2^\lambda \leftarrow^* Q_2 \leftarrow P$ where λ is the instantiation of the symbolic critical pair.

The next step is to show that the joining derivations $P \Rightarrow Q_1 \Rightarrow^* X_1^\lambda \cong X_2^\lambda \leftarrow^* Q_2 \leftarrow P$ can be extended via the morphism $e : P \rightarrow G \in \mathcal{N}$. This involves constructing the initial pushout of e and showing that the joining derivations preserve its boundary graph. Here the commutativity of the squares (2) and (3) in Definition 3 is used, together with the properties of pullbacks and initial pushouts. The final step involves showing that $X_1'' \cong X_2''$. This is due to the commutativity of (4) in Definition 3 and that pushouts are unique up to isomorphism. \square

Remark. The full proof of the theorem requires the construction of the boundary/context graph of $e : P \rightarrow G \in \mathcal{N}$. This is always possible in our setting - use the same definition as in the unlabelled case (e.g. see [3, Example 6.2]) and omit labels as done in Fig. 7b. Other necessary results include the Embedding and Extension Theorems, which are easily obtained by inspecting the proofs in [5] which already considers categories with a special set of vertical morphisms.

Confluence Analysis. Next we give our decision procedure for confluence based on symbolic critical pairs. In the following, we consider only terminating sets of rules \mathcal{R} since a non-terminating rule set may be locally confluent but not confluent. We begin by discussing persistent reducts.

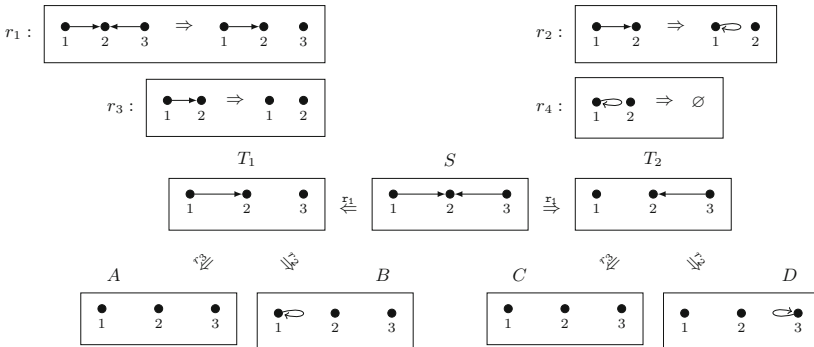


Fig. 8. Joinability analysis with persistent reducts.

In the context of a critical pair $T_1 \leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2$ with a set of persistent items N , a graph X derivable from T_1 or T_2 is a *persistent reduct* if the only rules applicable to X would delete the image of a common persistent item, i.e. an item in $track_{S \Rightarrow T_i} \Rightarrow^* X(N)$. Such graphs are useful when searching for joining

derivations – one need not consider graphs derivable from such reducts because strong joinability requires the existence of all persistent items, expressed as the following proposition. For its proof see [11].

Proposition 1. *If a critical pair $T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2$ is strongly joinable, then there exists a pair of E-isomorphic graphs X_1 and X_2 such that X_i is a persistent reduct derivable from T_i , $i = 1, 2$. Moreover, the isomorphism commutes on the persistent items of the critical pair in the sense of Definition 3.*

However, it is not enough to nondeterministically compute a pair of reducts and then compare them for strong joinability. Instead, one needs to consider *all* reducts. Consider the terminating set of rules in Fig. 8. This system is non-confluent because of the derivations $A \Leftarrow^* S \Rightarrow^* D \xrightarrow{T_4} \bullet$, which are two non-isomorphic normal forms. However, a confluence checker needs to search for strong joinability of critical pairs first. A strongly joinable critical pair $T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2$ is given. All the nodes of S are persistent nodes, and there are no persistent edges. The graphs T_1 and T_2 have multiple persistent reducts - T_1 reduces to A and B while T_2 reduces to C and D . The isomorphism $A \cong C$ demonstrates strong joinability, $B \cong D$ but violates the strictness condition, $A \not\cong D$ and $B \not\cong C$, thus a confluence checker needs to compare all persistent reducts for isomorphism.

Confluence Algorithm. Given a set of symbolic critical pairs and a terminating rewrite relation \mathcal{R} , Algorithm 1 checks whether all symbolic critical pairs are strongly joinable (Definition 3) by computing persistent reducts and then checking for isomorphisms (that are compatible with the joining derivations according to Definition 3). If that is the case, then the symbolic critical pair is strongly joinable. It is sufficient to consider persistent reducts due to Proposition 1. If all critical pairs are found to be strongly joinable, the algorithm reports \mathcal{R} to be confluent. Otherwise, it reports “unknown”.

Isomorphism checking is an integral part of joinability analysis. Since at the host graph level every label is taken from the concrete GP 2 label algebra without variables, checking for isomorphism (\cong) is decidable. However, when analysing graphs at the symbolic level, the problem of E-isomorphism (\cong_E) involves deciding validity of equations in Peano arithmetic. To the best of our knowledge, the problem is open for pure equations (no negation). Nevertheless, decidable fragments exist such as Presburger Arithmetic, whose decision procedures can be used during the analysis of the shortest distances case study (shown in [11]).

Note that the confluence algorithm does not determine non-confluence. This is due to the limitations of symbolic rewriting: not every host graph derivation can be represented by a symbolic derivation. However, in certain cases it is possible to use a combination of unification and satisfiability checking to determine that two non-isomorphic persistent reducts represent non-isomorphic normal forms at the host graph level. In these cases the algorithm could report non-confluence, which is a topic of ongoing work.

Input : A terminating set of left-linear rules \mathcal{R} with a set of critical pairs CP

```

1 foreach  $cp = (T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2)$  in CP do
2   for  $i = 1, 2$  do
3     | construct all derivations  $T_i \Rightarrow_{\mathcal{R}}^* X_i$  where  $X_i$  is a persistent reduct
4     | {let  $PR_i$  be the set of all persistent reducts  $X_i$ }
5   end
6   foreach pair of graphs  $(A, B)$  in  $PR_1 \times PR_2$  do
7     | if there exists a strong isomorphism  $A \rightarrow B$  then
8     | | mark  $cp$  as strongly joinable
9     | end
10  end
11 end
12 if all critical pairs in CP are strongly joinable then
13 | return “confluent”
14 else
15 | return “unknown”
16 end

```

Algorithm 1. Confluence Analysis Algorithm

Related Work. It is important to stress the differences with the symbolic approach of [14] which also defines symbolic critical pairs. That paper is in the context of symbolic graph transformation [16] where whole classes of attributed graphs are transformed via symbolic rules (rules equipped with first-order logical formulas). Symbolic critical pairs represent conflicts between such symbolic rules. However, no construction algorithm is given for these critical pairs. In fact, that paper treats attribute algebras as parametric, and thus a general construction algorithm is an undecidable problem. Joinability and local confluence are not considered. Symbolic rewriting is used to check critical pairs for strong confluence (joinability with 1/0-length derivations), which serves as an inspiration for validity checking in our case study.

The differences with critical pairs in the attributed setting of [3] are similar to the above. In this setting, graph attributes are represented via special *data* nodes and linked to ordinary graph items via attribution edges, giving rise to infinite graphs. The critical pair construction however is restricted to rules whose attributes are variables or variable-free. The algorithmic aspects of confluence analysis are not considered.

7 Conclusion

We have presented a method for statically verifying confluence (functional behaviour) of terminating sets of GP 2 rules, based on constructing the symbolic critical pairs of a rule set and checking that all pairs are strongly joinable with symbolic derivations. The correctness of this method is a consequence of the main technical result, namely that a set of left-linear attributed rules is locally confluent if all symbolic critical pairs are strongly joinable. We have also shown it is sufficient to focus on the persistent reducts when checking strong joinability.

In a case study, we used our method to verify the confluence of a graph program that calculates shortest distances.

An interesting topic of future work is the extension of confluence analysis to handle GP 2 program constructs other than looping, e.g. conditional branching. Other topics are the practical aspects of joinability analysis, namely developing decision procedures for label equivalence (e.g. see [13]), and the theoretical treatment of conditional rule schemata.

References

1. Bak, C.: GP 2: efficient implementation of a graph programming language. Ph.D. thesis, University of York (2015). <http://etheses.whiterose.ac.uk/id/eprint/12586>
2. Bang-Jensen, J., Gutin, G.: *Digraphs: Theory, Algorithms and Applications*, Second edn. Springer, Heidelberg (2009). <https://doi.org/10.1007/978-1-84800-998-1>
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science, Springer, Heidelberg (2006). <https://doi.org/10.1007/3-540-31188-2>
4. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: \mathcal{M} -adhesive transformation systems with nested application conditions: part 2: embedding, critical pairs and local confluence. *Fundamenta Informaticae* **118**(1–2), 35–63 (2012). <https://doi.org/10.3233/FI-2012-705>
5. Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive high-level replacement categories and systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 144–160. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_12
6. Habel, A., Plump, D.: Relabelling in graph transformation. In: Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 135–147. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45832-8_12
7. Habel, A., Plump, D.: \mathcal{M}, \mathcal{N} -adhesive transformation systems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 218–233. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_15
8. Hristakiev, I.: Confluence analysis for a graph programming language. Ph.D. thesis, University of York (2017, to appear)
9. Hristakiev, I., Plump, D.: A unification algorithm for GP 2. In: *Graph Computation Models (GCM 2014)*, Revised Selected Papers. Electronic Communications of the EASST, vol. 71 (2015). <https://doi.org/10.14279/tuj.eceasst.71.1002>
10. Hristakiev, I., Plump, D.: Attributed graph transformation via rule schemata: Church-Rosser theorem. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 145–160. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-50230-4_11
11. Hristakiev, I., Plump, D.: Checking graph programs for confluence (long version) (2017). <https://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.GCM17.Long.pdf>
12. Hristakiev, I., Plump, D.: Towards critical pair analysis for the graph programming language GP 2. In: James, P., Roggenbach, M. (eds.) WADT 2016. LNCS, vol. 10644, pp. 153–169. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-72044-9_11. Long version available at: <https://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.WADT16.Long.pdf>

13. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Texts in Theoretical Computer Science, Springer, Second edn. (2016). <https://doi.org/10.1007/978-3-662-50497-0>
14. Kulcsár, G., Deckwerth, F., Lochau, M., Varró, G., Schürr, A.: Improved conflict detection for graph transformation with attributes. In: Rensink, A., Zambon, E. (eds.) Proceedings of Graphs as Models (GaM 2015). Electronic Proceedings in Theoretical Computer Science, vol. 181, pp. 97–112 (2015). <https://doi.org/10.4204/EPTCS.181.7>
15. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24, <https://github.com/Z3Prover/z3>
16. Orejas, F., Lambers, L.: Lazy graph transformation. *Fundamenta Informaticae* **118**(1–2), 65–96 (2012). <https://doi.org/10.3233/FI-2012-706>
17. Plump, D.: Confluence of graph transformation revisited. In: Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R. (eds.) Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday. LNCS, vol. 3838, pp. 280–308. Springer, Heidelberg (2005). https://doi.org/10.1007/11601548_16
18. Plump, D.: The design of GP 2. In: Escobar, S. (ed.) Proceedings of International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011). Electronic Proceedings in Theoretical Computer Science, vol. 82, pp. 1–16 (2012). <https://doi.org/10.4204/EPTCS.82.1>

Loose Graph Simulations

Alessio Mansutti¹(✉), Marino Miculan¹, and Marco Peressotti²

¹ Department of Mathematics, Computer Science and Physics,
University of Udine, Udine, Italy

`alessio.mansutti@lsv.fr`, `marino.miculan@uniud.it`

² Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark
`peressotti@imada.sdu.dk`

Abstract. We introduce *loose graph simulations* (LGS), a new notion about labelled graphs which subsumes in an intuitive and natural way *subgraph isomorphism* (SGI), *regular language pattern matching* (RLPM) and *graph simulation* (GS). Being a unification of all these notions, LGS allows us to express directly also problems which are “mixed” instances of previous ones, and hence which would not fit easily in any of them. After the definition and some examples, we show that the problem of finding loose graph simulations is NP-complete, we provide formal translation of SGI, RLPM, and GS into LGSs, and we give the representation of a problem which extends both SGI and RLPM. Finally, we identify a subclass of the LGS problem that is polynomial.

1 Introduction

Graph pattern matching is the problem of finding patterns satisfying a specific property, inside a given graph. This problem arises naturally in many research fields: for instance, in computer science it is used in automatic system verification, network analysis and data mining [5, 15, 25, 28]; in computational biology it is applied to protein sequencing [24]; in cheminformatics it is used to study molecular systems and predict their evolution [1, 4]. As a consequence, many definitions of patterns have been proposed; for instance, these patterns can be specified by another graph, by a formal language, by a logical predicate, etc. This situation has led to different notions of graph pattern matching, such as *subgraph isomorphism* (SGI), *regular language pattern matching* (RLPM) and *graph simulation* (GS). Each of these notions has been studied in depth, yielding similar but different theories, algorithms and tools.

A drawback of this situation is that it is difficult to deal with matching problems which do not fit directly in any of these variants. In fact, often we need

M. Miculan—Partially supported by PRID 2017 *ENCASE* of the University of Udine.

M. Peressotti—Partially supported by the Open Data Framework project at the University of Southern Denmark, and by the Independent Research Fund Denmark, Natural Sciences, grant no. DFF-7014-00041.

to search for patterns that can be expressed as compositions of several graph pattern matching notions. An example is when we have to find a pattern which has to satisfy multiple notions of graph pattern matching at once; due to the lack of proper tools, these notions can only be checked one by one with a worsening of the performances. Another example can be found in [9], where extensions of RLPM and their application in network analysis and graph databases are discussed. A mixed problem between SGI and RLPM is presented in [2].

This situation would benefit from a more general notion of graph pattern matching, able to subsume naturally the more specific ones found in literature. This general notion would be a common ground to study specific problems and their relationships, as well as to develop common techniques for them. Moreover, a more general pattern matching notion would pave the way for more general algorithms, which would deal more efficiently with “mixed” problems.

To this end, in this paper we propose a new notion about labelled graphs, called *loose graph simulation* (LGS, Sect. 2). The semantics of its pattern queries allow us to check properties from different classical notions of pattern matching, at once and without cumbersome encodings. LGS queries have a natural graphical representation that simplifies the understanding of their semantic; moreover, they can be composed using a sound and complete algebra (Sect. 3). Various notions of graph pattern matching can be naturally reduced to LGSs, as we will formally prove in Sects. 4, 5 and 6; in particular, the encoding of subgraph isomorphism allows us to prove that computing LGSs is an NP-complete problem. Moreover, “mixed” matching problems can be easily represented as LGS queries; in fact, these problems can be obtained compositionally from simpler ones by means of the query algebra, as we will show in Sect. 7 where we solve a simplified version of the problem in [2]. Lastly (Sect. 8), we study a polynomial-time fragment of LGS that can still be used to compute various notions of graph pattern matching. Final conclusions and directions for further work (such as a distributed algorithm for computing LGSs) are in Sect. 9.

2 Hosts, Guests and Loose Graph Simulations

Loose graph simulations are a generalization of pattern matching for certain labelled graphs. As often proposed in the literature, the structures that need to be checked for properties are called *hosts*, whereas the structures that represent said properties are called *guests*.

Definition 1. A host graph (herein also simply called graph) is a triple (Σ, V, E) consisting of a finite set of symbols Σ (also called alphabet), a finite set V of nodes and a set $E \subseteq V \times \Sigma \times V$ of edges. For an edge $e = (v, l, v')$ write $s(e)$, $\sigma(e)$, and $t(e)$ for its source node v , label l , and target node v' , respectively. For a vertex v write $\text{in}(v)$ and $\text{out}(v)$ for the sets $\{e \mid t(e) = v\}$ and $\{e \mid s(e) = v\}$ of its incoming and outgoing edges.

Definition 2. A guest $G = (\Sigma, V, E, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ is a (host) graph (Σ, V, E) additionally equipped with:

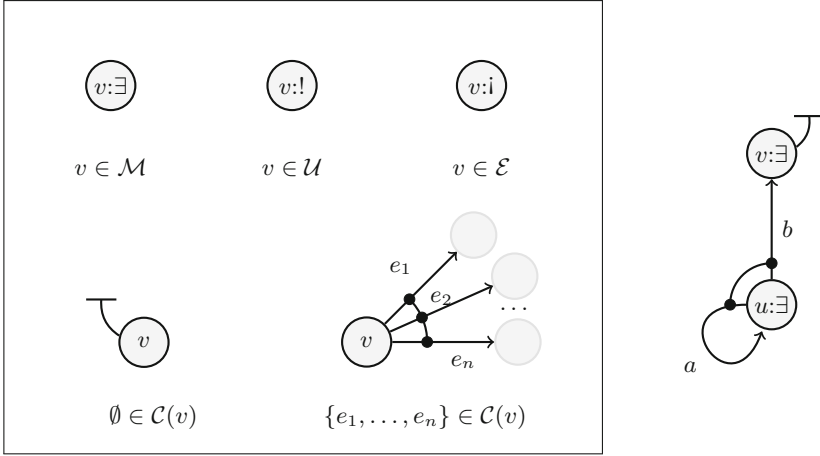


Fig. 1. The guest graphic notation (left) and an example (right).

- three sets $\mathcal{M}, \mathcal{U}, \mathcal{E} \subseteq V$, called respectively must, unique and exclusive set.
- a choice function $\mathcal{C} : V \rightarrow \mathcal{P}(\mathcal{P}(E))$, s.t. $\bigcup \mathcal{C}(v) = \text{out}(v)$ for each $v \in V$.

Roughly speaking, a guest is graph whose:

- nodes are decorated with usage constraints telling whether they must appear in the host, if their occurrence should be unique, and whether their occurrences can also be occurrences of other nodes or are exclusive;
- edges are grouped into possible “choices of sets of ongoing edges” for any given source node to be considered by a simulation.

The semantics of the three sets $\mathcal{M}, \mathcal{U}, \mathcal{E}$ and the choice function \mathcal{C} will be presented formally in the definition of loose graph simulations (Definition 5).

Guests can be conveniently represented using the graphical notation shown in Fig. 1 (a formal algebra is discussed in Sect. 3). A node belonging to the must, unique or exclusive set is decorated with the symbols \exists , $!$ and i , respectively. Choice sets are represented by arcs with dots placed on the intersection with each edge that belongs to the given choice set. The empty choice set ($\emptyset \in \mathcal{C}(v)$) is represented by the “corked edge” (\curvearrowright).

Example 1. Figure 1 shows the graphical representation of a guest with two nodes u and v . The must set is $\{u, v\}$, the unique and exclusive sets are both empty, and the choice function takes u to $\{(u, a, u), (u, b, v)\}$ and v to $\{\emptyset\}$.

Before we formalise the notion of loose graph simulation, we need some auxiliary definitions. The following one fix the notation for paths in a graph.

Definition 3. For $M = (\Sigma, V, E)$, define \mathbb{P}_M as the set of all paths in M , i.e. $\bigcup_{n \in \mathbb{N}} \{(e_0, \dots, e_n) \in E^n \mid \forall i \in \{1, \dots, n\} s(e_i) = t(e_{i-1})\}$. Source ($s : \mathbb{P}_M \rightarrow V$),

target $(t: \mathbb{P}_M \rightarrow V)$, and label $(\sigma: \mathbb{P}_M \rightarrow \Sigma^+)$ functions are extended accordingly: $s((e_0, \dots, e_n)) \triangleq s(e_0)$, $t((e_0, \dots, e_n)) \triangleq t(e_n)$, and $\sigma((e_0, \dots, e_n)) \triangleq \sigma(e_0) \dots \sigma(e_n)$. Lastly, for any $v, v' \in V$, define $\mathbb{P}_M(v, v')$ as the set of all paths from v to v' , formally $\mathbb{P}_M(v, v') \triangleq \{\rho \in \mathbb{P}_M \mid s(\rho) = v \wedge t(\rho) = v'\}$.

Akin to graph simulations (Definition 11), LGSs are subgraphs of the product of guest and host that are coherent with the additional data prescribing node and edge usage.

Definition 4. Let $M_1 = (\Sigma_1, V_1, E_1)$ and $M_2 = (\Sigma_2, V_2, E_2)$ be two graphs. The tensor product graph $M_1 \times M_2$ is the graph $(\Sigma_1 \cap \Sigma_2, V_1 \times V_2, E^\times)$ where $E^\times \triangleq \{((u, u'), a, (v, v')) \mid (u, a, v) \in E_1 \wedge (u', a, v') \in E_2\}$.

When clear from the context, we denote host graphs and their components as H and as (Σ_H, V_H, E_H) (and variations thereof). We adopt the convention of denoting guests as G (and variations thereof) and writing $(\Sigma_G, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ for the components of the guest G . We are now ready to define the notion of loose graph simulation.

Definition 5. A loose graph simulation (LGS for short) of G in H is a subgraph $(\Sigma_G \cap \Sigma_H, V^{G \rightarrow H}, E^{G \rightarrow H})$ of $G \times H$ subject to the following conditions:

- (LGS1) vertices of G in the must set occur in $V^{G \rightarrow H}$, i.e. for each $u \in \mathcal{M}$ there exists $u' \in V_H$ such that $(u, u') \in V^{G \rightarrow H}$;
- (LGS2) vertices in the unique set are assigned to at most one vertex of H , i.e. for each $u \in \mathcal{U}$ and all $u', v' \in V_H$, if $(u, u') \in V^{G \rightarrow H}$ and $(u, v') \in V^{G \rightarrow H}$ then $u' = v'$;
- (LGS3) vertices of H assigned to a vertex in the exclusive set cannot be assigned to other vertices, i.e. for each $u \in \mathcal{E}$, $v \in V_G$ and $u' \in V_H$, if $(u, u') \in V^{G \rightarrow H}$ and $(v, u') \in V^{G \rightarrow H}$ then $u = v$;
- (LGS4) for $(u, u') \in V^{G \rightarrow H}$, there is a set in $\mathcal{C}(u)$ s.t. each of its elements is related to an edge with source u' and only such edges occur in $E^{G \rightarrow H}$. Formally,
 - for each $(u, u') \in V^{G \rightarrow H}$ there exists $\gamma \in \mathcal{C}(u)$ such that for all $(u, a, v) \in \gamma$ it holds that $((u, u'), a, (v, v')) \in E^{G \rightarrow H}$ for some $v' \in V_H$;
 - for each $((u, u'), a, (v, v')) \in E^{G \rightarrow H}$ there exists $\gamma \in \mathcal{C}(u)$ s.t. $(u, a, v) \in \gamma$ and for each $(u, b, w) \in \gamma$ it holds that $((u, u'), b, (w, w')) \in E^{G \rightarrow H}$ for some $w' \in V_H$.
- (LGS5) the simulation preserves the connectivity w.r.t. nodes marked as must: for each $(u, u') \in V^{G \rightarrow H}$ and $v \in \mathcal{M}$ if $\mathbb{P}_G(u, v) \neq \emptyset$ then there exists $v' \in V_H$ such that $\mathbb{P}_{(\Sigma_G \cap \Sigma_H, V^{G \rightarrow H}, E^{G \rightarrow H})}((u, u'), (v, v')) \neq \emptyset$.

The domain of all LGSs for G and H is denoted as $\mathbb{S}^{G \rightarrow H}$.

As already mentioned at the end of Definition 2, the definition of LGS attributes a semantics for the must, unique, exclusive sets and the choice function. Regarding the *unique set*, Condition LGS2 requires that every vertex of the guest in this set to be mapped by at most one element of the host. Similarly,

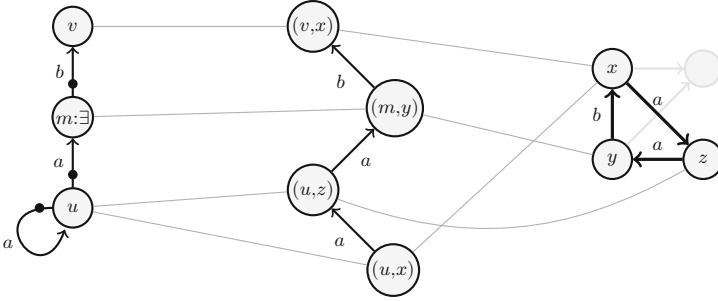


Fig. 2. An LGS (center) between a guest (left) and a host (right).

Condition LGS3 requires the vertices of the host paired in the LGS with a node of the *exclusive set* to be only paired with that node. Condition LGS4 defines the semantics of the choice function: given a pair of vertices $(u, u') \in V^{G \rightarrow H}$, it requires to select at least one set from $\mathcal{C}(u)$. The edges of these selected sets (and only these edges, as stated by the second part of the condition) must be paired in the LGS to edges in H with source u' . This condition can be seen as a generalization of the second condition of *graph simulations* (Definition 11) that requires all outgoing edges from u to be in relation with outgoing edges of u' .

Condition LGS1 and LGS5 formalise the constraints attached to must nodes: the first condition imposes that every vertex in this set must appear in the LGS, while the second condition requires that, for each $(u, u') \in V^{G \rightarrow H}$, each vertex in the must set reachable in the guest from u is also reachable in the LGS, with a path starting from (u, u') .

Example 2. Figure 2 shows a guest and its loose graph simulation over a host. In this example $\mathcal{M} = \{m\}$ and $\mathcal{U} = \mathcal{E} = \emptyset$. Moreover, the choice function is *linear*, i.e. for each vertex u , $\mathcal{C}(u)$ contains a set $\{e\}$ for each edge in $\text{out}(u)$ and \emptyset whenever $\text{out}(u) = \emptyset$, formally $\mathcal{C} = \lambda x. \{\{e\} \mid e \in \text{out}(x)\} \cup \{\emptyset \mid \text{out}(x) = \emptyset\}$. LGSs of this guest represents paths (e_0, e_1, \dots, e_n) of arbitrary length in the host such that $\forall i < n \sigma(e_i) = a$ and $\sigma(e_n) = b$. The guest is therefore similar to the regular language $a^* b$ and a LGS identifies paths in the host labelled with words in this language.

Proposition 1. *Let G be a guest with choice function \mathcal{C} defined as $\lambda x. \{\text{out}(x)\}$, let H be a host and let $S = (\Sigma_G \cap \Sigma_H, V^{G \rightarrow H}, E^{G \rightarrow H})$ be a subgraph of $G \times H$. If S satisfies Condition LGS4 then it also satisfies Condition LGS5.*

Proof. Let $\mathcal{C}(v) = \{\text{out}(v)\}$ for all $v \in V_G$. If $(u, u') \in V^{G \rightarrow H}$ then Condition LGS4 requires that for all $(u, a, v) \in \text{out}(u)$ there exists v' such that $(v, v') \in V^{G \rightarrow H}$ and $((u, u'), a, (v, v')) \in E^{G \rightarrow H}$. Coinductively, since the same will hold for every of those pair (v, v') , it follows that whenever there is a path in G from u to a node $m \in \mathcal{M}$ in the must set, then there must be a path in S from (u, u') to a pair of vertices (m, w) , where $w \in V_H$. Hence, Condition LGS5 holds. \square

3 An Algebra for Guests

Guests are used to specify the patterns to look for inside a host; hence they should be easy to construct and to understand. To this end, besides the graphical notation described in Sect. 2, in this section we introduce an algebra for guests which allows us to construct them in a compositional way.

Definition 6. *A guest is empty whenever it has no vertexes. A guest with only one vertex and no edges is a unary guest and is denoted a*

$$p_A \triangleq (\emptyset, \{p\}, \emptyset, \{p \mid \exists \in \mathcal{A}\}, \{p \mid ! \in \mathcal{A}\}, \{p \mid i \in \mathcal{A}\}, \{p \rightarrow \{\emptyset \mid \emptyset \in \mathcal{A}\}\})$$

where p is the only vertex and $\mathcal{A} \subseteq \{\exists, !, i, \emptyset\}$ state if p is respectively in \mathcal{M} , \mathcal{U} , \mathcal{E} or if $\emptyset \in \mathcal{C}(p)$. For α a name, P and Q unary guests, the arrow operator from P to Q α is defined as

$$P \xrightarrow{\alpha} Q \triangleq (\{\alpha\}, \{p, q\}, \{(p, \alpha, q)\}, \mathcal{M}_P \cup \mathcal{M}_Q, \mathcal{U}_P \cup \mathcal{U}_Q, \mathcal{E}_P \cup \mathcal{E}_Q, \mathcal{C}^{\rightarrow})$$

$$\mathcal{C}^{\rightarrow} \triangleq \lambda x. \begin{cases} c_P \cup \{\{(p, \alpha, q)\}\} \cup c_Q & \text{if } p = q \wedge x = p \\ c_P \cup \{\{(p, \alpha, q)\}\} & \text{if } p \neq q \wedge x = p \\ c_Q & \text{if } p \neq q \wedge x = q \end{cases}$$

A guest is called elementary whenever it is empty, unary, or the result of the arrow operator.

For example, a node p with only a self loop labelled α can be expressed with the term $p \xrightarrow{\alpha} p$. Besides the elementary guests, the algebra is completed by introducing two binary operators used to combine guests.

Definition 7. *Let G_1 and G_2 be two guests. Their addition is the guest:*

$$G_1 \oplus G_2 \triangleq (\Sigma_1 \cup \Sigma_2, V_1 \cup V_2, E_1 \cup E_2, \mathcal{M}_1 \cup \mathcal{M}_2, \mathcal{U}_1 \cup \mathcal{U}_2, \mathcal{E}_1 \cup \mathcal{E}_2, \mathcal{C}^{\oplus})$$

where the choice function \mathcal{C}^{\oplus} is defined as

$$\mathcal{C}^{\oplus} \triangleq \lambda x. \begin{cases} \mathcal{C}_1(x) \cup \mathcal{C}_2(x) & \text{if } x \in V_1 \wedge x \in V_2 \\ \mathcal{C}_1(x) & \text{if } x \in V_1 \\ \mathcal{C}_2(x) & \text{if } x \in V_2 \end{cases}$$

The multiplication of G_1 and G_2 is the guest:

$$G_1 \otimes G_2 \triangleq (\Sigma_1 \cup \Sigma_2, V_1 \cup V_2, E_1 \cup E_2, \mathcal{M}_1 \cup \mathcal{M}_2, \mathcal{U}_1 \cup \mathcal{U}_2, \mathcal{E}_1 \cup \mathcal{E}_2, \mathcal{C}^{\otimes})$$

where the choice function \mathcal{C}^{\otimes} is defined as follows

$$\mathcal{C}^{\otimes} \triangleq \lambda x. \begin{cases} \{\gamma_1 \cup \gamma_2 \mid \gamma_1 \in \mathcal{C}_1(x) \wedge \gamma_2 \in \mathcal{C}_2(x)\} & \text{if } x \in V_1 \wedge x \in V_2 \\ \mathcal{C}_1(x) & \text{if } x \in V_1 \\ \mathcal{C}_2(x) & \text{if } x \in V_2 \end{cases}$$

Notice how addition and multiplication operators differ only by the definition of the choice function for vertices of both G_1 and G_2 . In the case of addition, the resulting choice function is the union of the two choice function \mathcal{C}_1 and \mathcal{C}_2 , whereas for the multiplication, given a vertex $v \in V_1 \cap V_2$, every set of $\mathcal{C}^\otimes(v)$ is the union of a set in $\mathcal{C}_1(v)$ and one in $\mathcal{C}_2(v)$.

Proposition 2. *The operations \oplus and \otimes form an idempotent commutative semiring structure over the set of all guests.*

The algebra offers a clean and modular representation of guests. Modularity, in particular, allows us to combine queries as illustrated in the second part of this work. Furthermore, guests admit normal forms.

Definition 8. *A term G in the algebra of guests is in normal form whenever $G = \bigoplus_{i \in I} \bigotimes_{j \in J_i} G_{i,j}$ where each $G_{i,j}$ is an elementary guest.*

Example 3. Consider the guest $(\{a, b\}, \{p, q\}, \{(p, a, p), (p, b, q)\}, \{p, q\}, \emptyset, \emptyset, \{p \mapsto \{\{(p, a, p), (p, b, q)\}\}, q \mapsto \{\emptyset\}\})$ shown in Fig. 1 on the right. This guest is represented by the term $q_{\{\exists, \emptyset\}} \oplus (p_{\{\exists\}} \xrightarrow{a} p \otimes p \xrightarrow{b} q)$ which is in normal form.

Every guest admits a normal form.

Proposition 3. *For $G = (\Sigma, V, E, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ a guest, its normal form is:*

$$\bigoplus_{v \in V} v_{\{\exists | v \in \mathcal{M}\} \cup \{\exists | v \in \mathcal{U}\} \cup \{\exists | v \in \mathcal{E}\} \cup \{\emptyset | \emptyset \in \mathcal{C}(v)\}} \oplus \bigoplus_{\substack{v \in V \\ \gamma \in \mathcal{C}(v)}} \left(\bigotimes_{e \in \gamma} \left(s(e) \xrightarrow{\sigma(e)} t(e) \right) \right)$$

For $G = (\Sigma, V, E, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ a guest, we write $G[p/q]$ for the guest obtained renaming $p \in V$ as $q \notin V$. In particular, the set of edges and choice function are:

$$E[p/q] = \left\{ (u, a, v) \left| \begin{array}{l} (u', a, v') \in E \\ (u' \neq p \implies u = u') \wedge (u' = p \implies u = q) \\ (v' \neq p \implies v = v') \wedge (v' = p \implies v = q) \end{array} \right. \right\}$$

$$\mathcal{C}[p/q] = \lambda x. \begin{cases} \{S[p/q] \mid S \in \mathcal{C}(x)\} & \text{if } x \neq p \wedge x \neq q \\ \{S[p/q] \mid S \in \mathcal{C}(p)\} & \text{if } x = q \end{cases}$$

4 The LGS Problem is NP-complete

In this section we analyse the complexity of computing LGSs by studying their emptiness problem. Without loss of generality, we will now consider only guests and hosts with the same Σ . In the following, let $G = (\Sigma_G, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ and $H = (\Sigma_H, V_H, E_H)$ be a guest and a host respectively.

Definition 9. *The emptiness problem for LGSs between G and H consists in checking whether $\mathbb{S}^{G \rightarrow H} = \emptyset$.*

Proposition 4. *Computing LGSs, as well as their emptiness problem, is in NP.*

Proof. Let $S = (\Sigma, V^{G \rightarrow H}, E^{G \rightarrow H})$ be a subgraph of $G \times H$. We will now prove that there exists a polynomial algorithm w.r.t. the size of G and H that checks whether S satisfies all the conditions of Definition 5. The satisfiability checking of Condition LGS1 is in $\mathcal{O}(\mathcal{M} \times V^{G \rightarrow H})$ since it is sufficient for every vertex in the must set \mathcal{M} to check whether there is a vertex of the host paired with it. For similar reasons, Conditions LGS2 and LGS3 can also be checked in polynomial time. Moreover, to check Conditions LGS4 it is sufficient to check, for each $(u, v) \in V^{G \rightarrow H}$, whether there is $\gamma \in \mathcal{C}(v)$ s.t. $\gamma \subseteq \pi_1 \circ \text{out}((u, v))$ and if for all $u' \in \pi_1 \circ \text{out}((u, v))$ there exists $\gamma \in \mathcal{C}(v)$ s.t. $u' \in \gamma \subseteq \pi_1 \circ \text{out}((u, v))$. This can be done by a naive algorithm in $\mathcal{O}(V_H \times E_G \times (V_G \times E_H + \mathcal{C} \times E_G^2))$. Lastly, checking whether S satisfies Condition LGS5 requires the evaluation of the reachability relation of G and S and therefore can be computed in $\mathcal{O}(V_G^3 \times V_H^3)$ using the Floyd-Warshall Algorithm [11]. Since every condition can be checked in polynomial time we can conclude that the LGS problem is in NP. \square

4.1 NP-Hardness: Subgraph Isomorphisms via LGSs

We will now show the NP-hardness of the emptiness problem for LGSs by reducing the emptiness problem for subgraph isomorphism to it. The subgraph isomorphism problem requires to check whether a subgraph of a graph (host) and isomorphic to a second graph (query) exists. Application of this problem can be found in network analysis [15], bioinformatics and chemoinformatics [1, 4].

Definition 10. *Let $H = (\Sigma, V_H, E_H)$ and $Q = (\Sigma, V_Q, E_Q)$ be two graphs called host and query respectively. There exists a subgraph of H isomorphic to Q whenever there exists a pair of injections $\phi : V_Q \hookrightarrow V_H$ and $\eta : E_Q \hookrightarrow E_H$ s.t. $\sigma(e) = \sigma \circ \eta(e)$, $\phi \circ s(e) = s \circ \eta(e)$, and $\phi \circ t(e) = t \circ \eta(e)$ for each $e \in E_Q$.*

The subgraph isomorphism problem, as well as the emptiness problem associated to it, is shown to be NP-complete by Cook [6]. Its complexity and its importance makes it one of the most studied problem and multiple algorithmic solutions where derived for it [4, 7, 27]. We will now show that the emptiness problem for subgraph isomorphism can be solved using LGSs.

Proposition 5. *Let $H = (\Sigma, V_H, E_H)$ and $Q = (\Sigma, V_Q, E_Q)$ be a host and a query for subgraph isomorphism respectively. Moreover, let*

$$G = \bigoplus_{v \in V_Q} v_{\{\exists!\} \cup \{\emptyset | \text{out}(v) = \emptyset\}} \oplus \left(\bigotimes_{e \in E_Q} \left(s(e) \xrightarrow{\sigma(e)} t(e) \right) \right)$$

Then, there exists a subgraph of H isomorphic to Q iff there exists a LGS of G in H , i.e. $\mathbb{S}^{G \rightarrow H} \neq \emptyset$.

Proof. From the definition of G , its must, unique and exclusive sets, as well as its choice function, are $\mathcal{M} = \mathcal{U} = \mathcal{E} = V_Q$ and $\mathcal{C} = \lambda x. \{\text{out}(x)\}$ respectively. Suppose $\phi : V_Q \hookrightarrow V_H$ and $\eta : E_Q \hookrightarrow E_H$ be two injections as in Definition 10. Then the graph $S = (\Sigma, V^{G \rightarrow H}, E^{G \rightarrow H})$ where $V^{G \rightarrow H} \triangleq \{(u, u') \mid u' = \phi(u)\}$ and $E^{G \rightarrow H} \triangleq \{((u, u'), a, (v, v')) \mid (u', a, v') = \eta((u, a, v))\}$ form a LGS for G . Indeed, it satisfy Conditions LGS1 to LGS3, since ϕ is an injection. Moreover, since $\eta : E_Q \hookrightarrow E_H$ is also an injection and for each edge $e \in E_Q$ it holds that $\sigma(e) = \sigma \circ \eta(e)$, $\phi \circ s(e) = s \circ \eta(e)$ and $\phi \circ t(e) = t \circ \eta(e)$, S must be such that for each $(u, u') \in V^{G \rightarrow H}$ and for each $(u, a, v) \in \text{out}(u)$ there exists v' such that $(v, v') \in V^{G \rightarrow H}$ and $((u, u'), a, (v, v')) \in E^{G \rightarrow H}$. It follows that S is a subgraph of $G \times H$ and Condition LGS4 is satisfied, since $\mathcal{C}(u) = \{\text{out}(u)\}$. Moreover the satisfaction of Condition LGS5 follows from Proposition 1. S is therefore a LGS of G in H . Conversely, suppose that there is a LGS $S = (\Sigma, V^{G \rightarrow H}, E^{G \rightarrow H})$. Let ϕ s.t. $\phi(u) = u' \iff (u, u') \in V^{G \rightarrow H}$ and η s.t. $\eta((u, a, v)) = (u', a, v') \iff ((u, u'), a, (v, v')) \in E^{G \rightarrow H}$. Since $\mathcal{M} = \mathcal{U} = \mathcal{E} = V_Q$ and S is a LGS, it holds that ϕ is an injection defined on the domain V_Q . Moreover η is also an injection, since $\mathcal{C} = \lambda x. \{\text{out}(x)\}$ and S satisfies Condition LGS4, and together with the hypothesis that S is a subgraph of $G \times H$ it must also hold that for each edge $e \in E_Q$ $\sigma(e) = \sigma \circ \eta(e)$, $\phi \circ s(e) = s \circ \eta(e)$ and $\phi \circ t(e) = t \circ \eta(e)$. There exists therefore a subgraph of H isomorphic to Q . \square



Fig. 3. A possible query for subgraph isomorphism (on the left) and its translation to a guest for LGSs (on the right).

Note how the translation from subgraph isomorphism's queries to guest for LGSs defined in Proposition 5 is *structure-preserving*. Indeed, an example of this can be seen in Fig. 3. This property is important since it makes defining LGSs' guests to solve the subgraph isomorphism problem as intuitive as the respective queries for it. This is also the case for other notions commonly used in the graphs' pattern matching community. Moreover, since the translated guest is as intuitive as the original query, this property strengthens the idea of using guests and LGSs to represent and compute hybrid queries w.r.t. these notions.

From Propositions 4 and 5 it follows that:

Theorem 1. *The emptiness problem for LGSs is NP-complete.*

5 Graph Simulations Are Loose Graph Simulations

Graph simulations are particular relations between graphs that are extensively applied in several fields [8, 10]. The *graph simulation problem* requires to check whether a portion of a graph (host) *simulates* another graph (query).

Definition 11. A graph simulation of $Q = (\Sigma, V_Q, E_Q)$ (herein query) in $H = (\Sigma, V_H, E_H)$ (herein host) is a relation $\mathcal{R} \subseteq V_Q \times V_H$ such that:

- for each node $u \in V_Q$ there exists a node $v \in V_H$ such that $(u, v) \in \mathcal{R}$;
- for each pair $(u, v) \in \mathcal{R}$ and for each edge $e \in \text{out}(u)$ there exists an edge $e' \in \text{out}(v)$ such that $\sigma(e) = \sigma(e')$ and $(t(e), t(e')) \in \mathcal{R}$.

Graph simulation existence can be decided in polynomial time [3, 13]. Their emptiness problem can be reduced to the emptiness problem for loose ones.

Proposition 6. Let $H = (\Sigma, V_H, E_H)$ and $Q = (\Sigma, V_Q, E_Q)$ be a host and a query for graph simulation respectively. Moreover, let

$$G = \bigoplus_{v \in V_Q} v_{\{\exists\} \cup \{\emptyset \mid \text{out}(v) = \emptyset\}} \oplus \bigotimes_{e \in E_Q} s(e) \xrightarrow{\sigma(e)} t(e)$$

Then, there is a graph simulation of Q in H iff $\mathbb{S}^{G \rightarrow H} \neq \emptyset$.

Proof. From definition of G , its must, unique, exclusive sets and its choice function are $\mathcal{M} = V_Q$, $\mathcal{U} = \mathcal{E} = \emptyset$ and $\mathcal{C} = \lambda x. \{\text{out}(x)\}$ respectively. Let \mathcal{R} be a graph simulations. The graph $S = (\Sigma, V^{G \rightarrow H}, E^{G \rightarrow H})$ where $V^{G \rightarrow H} = \mathcal{R}$ and $E^{G \rightarrow H} = \{((u, u'), a, (v, v')) \mid (u, u'), (v, v') \in R, (u, a, v) \in E_Q, (u', a, v') \in E_H\}$ is a loose graph simulations for G . $\mathcal{U} = \mathcal{E} = \emptyset$ makes Conditions LGS2 and LGS3 always true, whereas the first condition of Definition 11, that requires all vertices of V_Q to appear in the first projection of \mathcal{R} , makes Conditions LGS1 satisfied. The second condition of Definition 11 requires that, given a pair $(u, v) \in R$, every edge of $\text{out}(u)$ is associated with one edge of $\text{out}(v)$ with the same label and with targets paired in \mathcal{R} . Condition LGS4 is therefore satisfied. Lastly, the satisfaction of Condition LGS5 follows from Proposition 1. S is therefore a loose graph simulation of G in H . Conversely, suppose there exists a LGS $S = (\Sigma, V^{G \rightarrow H}, E^{G \rightarrow H})$. Then $V^{G \rightarrow H}$ is a graph simulation. The definition of must set $\mathcal{M} = V_Q$ ensures that each vertex of V_Q must appear in the first projection of $V^{G \rightarrow H}$: the first condition of Definition 11 is satisfied. Moreover, the definition of the choice function $\mathcal{C} = \lambda x. \{\text{out}(x)\}$ and Condition LGS4 implies that for each $(u, u') \in V^{G \rightarrow H}$ and for all $(u, a, v) \in \text{out}(u)$ there exists v' such that $((u, u'), a, (v, v')) \in E^{G \rightarrow H}$ and, since S is a subgraph of $G \times H$, $(v, v') \in V^{G \rightarrow H}$. Thus, the second condition of Definition 11 holds and $V^{G \rightarrow H}$ is a graph simulation. \square

Example 4. Figure 4 shows a query for graph simulations and the equivalent guest for loose graph simulations. As already seen in Sect. 4.1, the translation preserve the structure of the graph.



Fig. 4. A possible query for *graph simulation* (on the left) and its translation in a guest for loose graph simulations (on the right).

6 Regular Languages Pattern Matching

Regular languages defines finite sequences of characters (called *words* or *strings*) from a finite alphabet Σ [14]. Although widely used in text pattern matching, they are also used in graph pattern matching [2, 20]. In this section we will restrict ourselves to ϵ -free regular languages, i.e. regular languages without the empty word ϵ [29]. This restriction is quite common, since the empty word is matched by any text or graph and therefore it does not represent a meaningful pattern.

Definition 12. Let Σ be an alphabet. \emptyset is a ϵ -free regular language. For each $a \in \Sigma$, $\{a\}$ is a ϵ -free regular language. If A and B are ϵ -free regular language, so are $A \cdot B \triangleq \{vw \mid v \in A \wedge w \in B\}$, $A \mid B \triangleq A \cup B$, and $A^+ \triangleq \bigcup_{n \in \mathbb{N}} A^{n+1}$.

In [29] it is shown that every regular language without the *empty* string ϵ can be expressed with the operations defined for ϵ -free regular languages. We will now introduce the pattern matching problem for non-empty ϵ -free regular languages. In the following let $H = (\Sigma, V_H, E_H)$ and \mathcal{L} be respectively a host and a ϵ -free regular language such that $\mathcal{L} \neq \emptyset$.

Definition 13. The *emptiness problem for regular language pattern matching (RLPM)* consist in checking if there is a path $\rho \in \mathbb{P}_H$ such that $\sigma(\rho) \in \mathcal{L}$.

To solve this problem using LGSs we will use the equivalence between regular languages and non-deterministic finite automata [26].

Definition 14. An *NFA* is a tuple, $N = (\Sigma, Q, \Delta, q_0, F)$ consisting of an alphabet Σ , a finite set of states Q , an initial state q_0 , a set of accepting (or final) states $F \subseteq Q$ and a transition function $\Delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$. Let $w = a_0, a_1, \dots, a_n$ be a word in Σ^* . The NFA N accepts w if there is a sequence of states r_0, r_1, \dots, r_{n+1} in Q such that $r_0 = q_0$, $r_{i+1} \in \Delta(r_i, a_i)$ for $i = 0, \dots, n$, and $r_{n+1} \in F$. With $\mathcal{L}(N)$ we denote the set of words accepted by N , i.e. its accepted language.

Remark 1. Any non-empty regular language without ϵ can be translated to a non-deterministic finite automaton (NFA) with one initial state (say q'_0), one final state (say f) and s.t. $\text{in}(q'_0) = \emptyset$ and $\text{out}(f) = \emptyset$. Indeed, for $N = (\Sigma, Q, \Delta, q_0, F)$ any NFA s.t. $\mathcal{L}(N) \neq \emptyset$ and $\epsilon \notin \mathcal{L}(N)$ define $N' = (\Sigma, Q \cup \{q'_0, f\}, \Delta', q'_0, \{f\})$ where:

- for all $a \in \Sigma$, $\Delta'(q'_0, a) \triangleq \Delta(q_0, a)$ and $\Delta'(f, a) = \emptyset$;
- for all $q \in Q$ and $a \in \Sigma$, $\Delta'(q, a) \triangleq \Delta(q, a) \cup \{f \mid F \cap \Delta(q, a) \neq \emptyset\}$.

By construction $\mathcal{L}(N) = \mathcal{L}(N')$, $\text{in}(q'_0) = \emptyset$, and $\text{out}(f) = \emptyset$.

Proposition 7. *Let $N = (Q, \Sigma, \Delta, q_0, \{f\})$ be a NFA where the initial state q_0 does not have any incoming transitions and the only final state f does not have any outgoing ones. Let $H = (\Sigma, V_H, E_H)$ be a host. Let*

$$G = q_0\{\exists\} \oplus f\{\exists, \emptyset\} \oplus \bigoplus_{q \in Q, a \in \Sigma, q' \in \Delta(q, a)} \left(q \xrightarrow{a} q' \right)$$

Then, there exists a path $\rho \in \mathbb{P}_H$ in H s.t. $\sigma(\rho)$ is accepted by N iff there exists a loose graph simulation of G in H , i.e. $\mathbb{S}^{G \rightarrow H} \neq \emptyset$.

Proof. It follows from definition of acceptance that if there is $(e_0, \dots, e_n) \in \mathbb{P}_H$ such that $\sigma(\rho)$ is accepted by N then, there is a sequence

$$(p_0, s(e_0)) \xrightarrow{\sigma(e_0)} (p_1, s(e_1)) \xrightarrow{\sigma(e_1)} \dots \xrightarrow{\sigma(e_{n-1})} (p_n, s(e_n)) \xrightarrow{\sigma(e_n)} (p_{n+1}, t(e_n))$$

such that $p_0 = q_0$ and $p_{n+1} = f$; for all $i \in \{1, \dots, n\}$ $t(e_{i-1}) = s(e_i)$; for all $i \in \{0, \dots, n\}$ $p_{i+1} \in \Delta(p_i)$. Regard the sequence as a graph, say S , then $S \in \mathbb{S}^{G \rightarrow H}$ since S is a subgraph of $G \times H$ and G is constructed from N by preserving its transition relation Δ . Conditions LGS1 to LGS3 hold since $p_0 = q_0$, $p_n = f$ and $\mathcal{U} = \mathcal{E} = \emptyset$. Conditions LGS4 holds since $\{(p_i, \sigma(e_i), p_{i+1})\} \in \mathcal{C}(p_i)$ for any $i \in \{0, \dots, n\}$ by construction. Conditions LGS5 holds since projecting the graph to its first component yields a path from q_0 to f . Representing G requires space polynomial in the size of N . Conversely, if there is $S \in \mathbb{S}^{G \rightarrow H}$ then LGS5 ensures that there is a path $\rho = (e_0, \dots, e_n)$ in it such that $\pi_1 \circ s(\rho) = q_0$ and $\pi_1 \circ t(\rho) = f$. It follows from definition of E that the path ρ is coherent with Δ , i.e. $\forall i \in \{0, \dots, n\}$ $\pi_1 \circ t(e_i) \in \Delta \circ \pi_1 \circ s(e_i)$. Thus, the sequence of labels $\sigma(\pi_2(\rho))$ in the second projection of ρ ($(\pi_2 \circ s(e_0), \sigma(e_0), \pi_2 \circ t(e_0)), \dots, (\pi_2 \circ s(e_n), \sigma(e_n), \pi_2 \circ t(e_n)))$), is such that $\sigma(\pi_2(\rho))$ is accepted by N . \square

Example 5. Figure 5 shows a NFA and a guest identifying the same language. These two objects have the same structure (states/nodes and transition/edges).

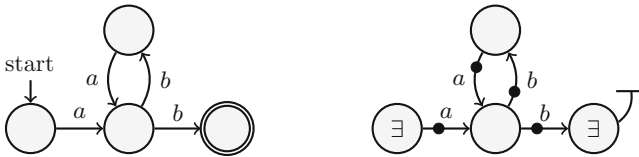


Fig. 5. A query for *regular languages* represented as an NFA (left) and as a LGS guest (on the right). The accepted language is $(ab)^+$.

7 Subgraph Isomorphism with Regular Path Expressions

Many approaches found in literature define hybrid notions of similarities, “merging” classical ones such as GS, SGI and RLPM [2,9]. These and similar merges are naturally handled by the modular definition of LGS guests. As an example, we discuss *subgraph isomorphism with regular languages* (RL-SGI) [2].

Definition 15. *Let Σ be a finite alphabet. A graph decorated with regular languages (over Σ) is a tuple $(\Sigma, V, E, \mathcal{L})$ consisting of a set V of nodes, a set $E \subseteq V \times V$ of edges and a labelling function $\mathcal{L} : E \rightarrow RE_\Sigma$ decorating each edge with a non empty ϵ -free regular language over Σ .*

Definition 16 (RL-SGI). *Let $H=(\Sigma, V_H, E_H)$ be a host and $Q=(\Sigma, V_Q, E_Q, \mathcal{L})$ a graph decorated with regular languages. We say that there is a regular-language subgraph isomorphism of Q into H iff there is a pair of injections $\phi : V_Q \hookrightarrow V_H$ and $\eta : E_Q \hookrightarrow \mathbb{P}_H$ s.t. for each $e \in E_Q$ $\phi \circ s(e) = s \circ \eta(e)$, $\phi \circ t(e) = t \circ \eta(e)$, and $\sigma \circ \eta(e) \in \mathcal{L}(e)$. Vertexes of paths in $\eta(E_Q)$ cannot appear in $\phi(V_Q)$ except for their source and target, i.e.: $\forall (e_0, \dots, e_n) \in \eta(E_Q) \forall i \in \{1, \dots, n\} s(e_i) \notin \phi(V_Q)$.*

RL-SGI can be seen as a hybrid notion between subgraph isomorphism and RLPM. We will now show how to solve this problem with loose graph simulations by defining a proper translation from its queries to guests.

Proposition 8. *Let $Q = (\Sigma, V_Q, E_Q, \mathcal{L})$ be a query for RL-SGI. Let*

$$G = \bigoplus_{v \in V_Q} v_{\{\exists!i\}} \oplus \bigotimes_{e \in E_Q} G_e[q_e/s(e)][f_e/t(e)]$$

such that G_e is the translation of the automaton $N_e = (\Sigma, V_e, \delta_e, q_e, \{f_e\})$ for $\mathcal{L}(e)$, as per Proposition 7 and where q_e and f_e are merged if $s(e) = t(e)$. For each host $H = (V_H, E_H)$ there exists a RL-SGI of Q into H iff $\mathbb{S}^{G \rightarrow H} \neq \emptyset$.

Proof. It follows from definition of G that: (i) V_Q is a subset of the vertices of V_G and $\mathcal{M} = \mathcal{U} = \mathcal{E} = V_Q$; (ii) for any $v \in V_Q$, any $\gamma \in \mathcal{C}(v)$, and any $e \in \text{out}(v)$ of Q , there is exactly one edge in γ that is induced by a transition in N_e . Similarly to the proof of Proposition 5, Conditions LGS1 to LGS3 together with the first property ensure that each LGS over G corresponds to an injection w.r.t V_Q . It follows from the second property, Proposition 7, Conditions LGS4 and LGS5 that every LGS over G contains, for each $e \in E_Q$ a path whose labels, starting and ending nodes lie in $\mathcal{L}(e)$ and $V_Q \times V_H$, whereas all other vertices are in $(V_G \setminus V_Q) \times V_H$. Then, $\mathbb{S}^{G \rightarrow H} \neq \emptyset$ iff there are RL-SGIs of Q into H . \square

Example 6. Figures 6 and 7 show a query for RL-SGI and its translation as a LGS guest. As illustrated by Proposition 8 and Fig. 7, translations are obtained modularly: following Sects. 4.1 and 6, the first step is to represent nodes and edges of a RL-SGI query in the guests for the SGI and RLPM queries, respectively; the second is to compose them via the guest algebra.

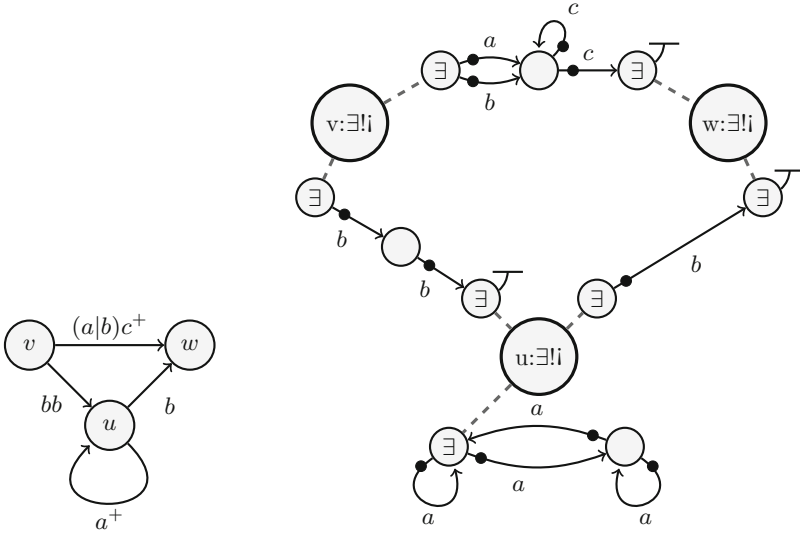


Fig. 6. A RE-SGISO query (left) and simple guests required to encode it (right). Vertices with the same name are highlighted by dashed edges between them.

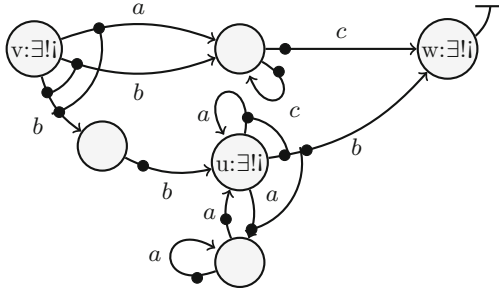


Fig. 7. A guest obtained via *multiplication* and *addition* operator from the guest in Fig. 6 (right) and equivalent to the RE-SGISO query in Fig. 6 (left).

8 A Polynomial Fragment of LGSs

RLPM and GS are two well-known problems for graph pattern matching and they both admit polynomial time algorithms. Since the emptiness problem for LGSs is NP-complete, we are interested in studying fragments of LGSs that are solvable in polynomial time yet expressive enough to capture the RLPM and GS problems. The class of simulation problems for guests whose unique and exclusive sets are empty enjoys this property.

Fix $G = (\Sigma_G, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ and $H = (\Sigma_H, V_H, E_H)$. If \mathcal{U} and \mathcal{E} are empty then, LGSs for G and H are closes under unions hence the union $\bigcup \mathbb{S}^{G \rightarrow H}$ of all LGSs correspond to the greatest LGS. Observe that greatest LGSs may not exist in the general case.

Proposition 9. *Let G be a guest such that $\mathcal{U} = \mathcal{E} = \emptyset$. Then $\bigcup \mathbb{S}^{G \rightarrow H}$ is a LGS.*

Figure 8 shows an algorithm for computing the greatest LGS provided that \mathcal{U} and \mathcal{E} are empty. The algorithm runs in polynomial time and can be readily adapted to compute the greatest LGSs included in a given subgraph of $G \times H$. It follows that the emptiness problem admits a polynomial procedure.

Data: A host H and a guest G s.t. $\mathcal{U} = \mathcal{E} = \emptyset$
Result: $\bigcup \mathbb{S}^{G \rightarrow H}$ if it exists, otherwise *false*.

```

1   $(\Sigma, V_S, E_S) \leftarrow G \times H;$ 
2  do
3     $(\Sigma, V_{S'}, E_{S'}) \leftarrow (\Sigma, V_S, E_S);$ 
4    foreach  $(u, v) \in V_{S'}$  do
5      foreach  $((u, v), a, (u', v')) \in \text{out}((u, v))$  do
6        if  $\nexists \gamma \in \mathcal{C}(u)$  s.t.  $(u, a, u') \in \gamma$  and  $\forall (u, b, u'') \in \gamma$ 
           $\exists (v, b, v'') \in \text{out}(v)$   $((u, v), b, (u'', v'')) \in \text{out}((u, v))$  then
7           $E_{S'} \leftarrow E_{S'} \setminus \{(u, v), a, (u', v')\};$ 
8        if  $(\text{out}((u, v)) = \emptyset$  and  $\emptyset \notin \mathcal{C}(u)$  or  $(\exists m \in \mathcal{M}$  s.t.  $\mathbb{P}_G(u, m) \neq \emptyset$  and
           $\forall v' \in V_H$   $\mathbb{P}_{(\Sigma, V_{S'}, E_{S'})}((u, v), (m, v')) = \emptyset)$  then
9           $E_{S'} \leftarrow E_{S'} \setminus (\text{out}((u, v)) \cup \text{in}((u, v)));$ 
10          $V_{S'} \leftarrow V_{S'} \setminus \{(u, v)\};$ 
11 while  $V_S \neq V_{S'}$  or  $E_S \neq E_{S'}$ ;
12 if  $\forall m \in \mathcal{M} \exists v \in V_H$  s.t.  $(m, v) \in V_S$  then return  $(\Sigma, V_S, E_S)$ ;
13 else return false;
```

Fig. 8. Algorithm for computing the greatest loose graph simulation.

Theorem 2. *Let H be a host and G be a guest such that $\mathcal{U} = \mathcal{E} = \emptyset$. Then, the maximal LGS exists and is computed in polynomial time.*

Proof. The algorithm in Fig. 8 starts by computing $G \times H$ and saving it to (Σ, V_S, E_S) (Line 1). Afterwards, the *do-while* loop (Lines 2–11) proceeds removing nodes and edges of (Σ, V_S, E_S) that do not satisfy Conditions LGS4 and LGS5. Lastly (Lines 12–15), Condition LGS1 is checked and, if satisfied, (Σ, V_S, E_S) is returned, otherwise there is no greatest LGS and the algorithm terminates returning *false*. The algorithm runs in polynomial time, since Conditions LGS1, LGS4 and LGS5 can be checked in polynomial time (Proposition 4)

and the loop will be performed at most $|V_S| + |E_S|$ times. Conditions at Lines 6 and 8 check that edges and nodes satisfy Conditions LGS4 and LGS5. If any of these does not hold, the temporary copy of (Σ, V_S, E_S) , i.e. $(\Sigma, V_{S'}, E_{S'})$, is updated removing an edge or a vertex. Thus, $V_S \neq V_{S'}$ or $E_S \neq E_{S'}$ iff (Σ, V_S, E_S) does not satisfy Conditions LGS4 and LGS5. After the *do-while* loop, (Σ, V_S, E_S) is a (possibly empty) relation that satisfies Conditions LGS4 and LGS5. Thus it remains only to check Condition LGS1 and this is done at Line 15: if the check fails there is no greatest LGSs otherwise it is the graph (Σ, V_S, E_S) returned by the algorithm. Assume otherwise that there is a LGS (Σ, V_M, E_M) s.t. $V_S \subset V_M$ or $E_S \subset E_M$. Then in (Σ, V_M, E_M) there is a node or an edge that satisfies LGS4 and LGS5 and is in $G \times H \setminus (\Sigma, V_S, E_S)$. Since it satisfies LGS4 and LGS5 it cannot be removed by the loop hence it is in (Σ, V_S, E_S) — a contradiction. \square

9 Conclusions and Future Work

In this paper we have introduced *loose graph simulations*, which are relations between graphs that can be used to check structural properties of labelled hosts. LGSs' guests can be represented using a simple graphical notation, but also compositionally by means of an algebra which is sound and complete. We have shown formally that computing LGSs is an NP-complete problem, where the NP-hardness is obtained via a reduction of subgraph isomorphism to them. Moreover, we have shown that many other classical notions of graph pattern matching are naturally subsumed by LGSs. Therefore, LGSs offer a simple common ground between multiple well-known notions of graph pattern matching supporting a modular approach to these notions as well as to the development of common techniques.

An algorithm for computing LGSs in a decentralised fashion and inspired to the “distributed amalgamation” strategy is introduced in [16]. Roughly speaking, the host graph is distributed over processes; each process uses its partial view of the host to compute partial solutions to exchange with its peers. Distributed amalgamation guarantees each solution is eventually found by at least one process.

The same strategy is at the core of distributed algorithms for solving problems such as *bigraphical embeddings* and the distributed execution of bigraphical rewriting systems [17, 19, 22]. Bigraphs [12, 21, 23] have been proved to be quite effective for modelling, designing and prototyping distributed systems, such as *multi-agent systems* [18]. This similarity and the ability of LGS to subsume several graph problems suggests to investigate graph rewriting systems where redex occurrences are defined in terms of LGSs.

Another topic for further investigation is how to systematically minimise guests or combine sets of guests into single instances, while preserving the semantics of LGSs. Moreover, following what already done in Sect. 8, the complexity of various fragments of LGSs still needs to be addressed, *eg.* defining a fragment that is *fixed-parameter tractable*. Results in these directions would have a positive practical impact on applications based on LGSs.

Acknowledgements. We thank the anonymous reviewers and the participants to the GCM'17 workshop for their comments. We thank Andrea Corradini for his insightful observations on a preliminary version of this work and for proposing the name “loose graph simulations”.

References

1. Apostolakis, J., Körner, R., Marialke, J.: Embedded subgraph isomorphism and its applications in cheminformatics and metabolomics. In: 1st German Conference in Cheminformatics (2005)
2. Barceló, P., Libkin, L., Reutter, J.L.: Querying regular graph patterns. *J. ACM* **61**(1), 8:1–8:54 (2014)
3. Bloom, B., Paige, R.: Transformational design and implementation of a new efficient solution to the ready simulation problem. *J. SCP* **24**(3), 189–220 (1995)
4. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D.E., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.* **14**(S-7), S13 (2013)
5. Chakrabarti, D., Faloutsos, C.: Graph mining: laws, generators, and algorithms. *ACM Comput. Surv.* **38**, 2 (2006)
6. Cook, S.A.: The complexity of theorem-proving procedures. In: *STOC*, pp. 151–158. ACM (1971)
7. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* **26**(10), 1367–1372 (2004)
8. Fan, W.: Graph pattern matching revised for social network analysis. In: *ICDT*, pp. 8–21. ACM (2012)
9. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y.: Adding regular expressions to graph reachability and pattern queries. *Front. Comput. Sci.* **6**(3), 313–338 (2012)
10. Fan, W., Wang, X., Wu, Y., Deng, D.: Distributed graph simulation: impossibility and possibility. *PVLDB* **7**(12), 1083–1094 (2014)
11. Floyd, R.W.: Algorithm 97: shortest path. *Commun. ACM* **5**(6), 345 (1962)
12. Grohmann, D., Miculan, M.: Directed bigraphs. In: *Proceedings of MFPS. Electronic Notes in Theoretical Computer Science*, vol. 173, pp. 121–137. Elsevier (2007)
13. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: *FOCS*, pp. 453–462. IEEE Computer Society (1995)
14. Hopcroft, J.E., Motwani, R., Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation - International Edition*, 2 edn. Addison-Wesley, Boston (2003)
15. Lischka, J., Karl, H.: A virtual network mapping algorithm based on subgraph isomorphism detection. In: *VISA*, pp. 81–88. ACM (2009)
16. Mansutti, A.: *Le simulazioni lasche: definizione, applicazioni e computazione distribuita*. Master's thesis, University of Udine (2016)
17. Mansutti, A., Miculan, M., Peressotti, M.: Distributed execution of bigraphical reactive systems. *ECEASST* **71**, 1–21 (2014)
18. Mansutti, A., Miculan, M., Peressotti, M.: Multi-agent systems design and prototyping with bigraphical reactive systems. In: Magoutis, K., Pietzuch, P. (eds.) *DAIS 2014. LNCS*, vol. 8460, pp. 201–208. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43352-2_16

19. Mansutti, A., Miculan, M., Peressotti, M.: Towards distributed bigraphical reactive systems. In: Echahed, R., Habel, A., Mosbah, M. (eds.) Proceedings of GCM (2014)
20. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. *SIAM J. Comput.* **24**(6), 1235–1258 (1995)
21. Miculan, M., Peressotti, M.: Bigraphs reloaded: a presheaf presentation. Technical report UDMI/01/2013, University of Udine (2013)
22. Miculan, M., Peressotti, M.: A CSP implementation of the bigraph embedding problem. CoRR, abs/1412.1042 (2014)
23. Milner, R.: The Space and Motion of Communicating Agents. Cambridge University Press, Cambridge (2009)
24. Pevzner, P.: Computational Molecular Biology - an Algorithmic Approach. MIT Press, Cambridge (2000)
25. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1. World Scientific, Singapore (1997)
26. Thompson, K.: Regular expression search algorithm. *Commun. ACM* **11**, 419–422 (1968)
27. Ullmann, J.R.: An algorithm for subgraph isomorphism. *J. ACM* **23**, 31–42 (1976)
28. Yan, X., Han, J.: gSpan: graph-based substructure pattern mining. In: ICDM, pp. 721–724. IEEE Computer Society (2002)
29. Ziadi, D.: Regular expression for a language without empty word. *Theor. Comput. Sci.* **163**(1&2), 309–315 (1996)

Grand Challenges in Modeling (GRAND)

Models, More Models, and Then a Lot More

Önder Babur¹(✉), Loek Cleophas¹, Mark van den Brand¹,
Bedir Tekinerdogan², and Mehmet Aksit³

¹ Eindhoven University of Technology, Eindhoven, The Netherlands
{O.Babur,L.G.W.A.Cleophas,M.G.J.v.d.Brand}@tue.nl

² Wageningen University & Research, Wageningen, The Netherlands
Bedir.Tekinerdogan@wur.nl

³ University of Twente, Enschede, The Netherlands
M.Aksit@utwente.nl

Abstract. With increased adoption of Model-Driven Engineering, the number of related artefacts in use, such as models, metamodels and transformations, greatly increases. To confirm this, we present quantitative evidence from both academia — in terms of repositories and datasets — and industry — in terms of large domain-specific language ecosystems. To be able to tackle this dimension of scalability in MDE, we propose to treat the artefacts as data, and apply various techniques — ranging from information retrieval to machine learning — to analyse and manage those artefacts in a holistic, scalable and efficient way.

Keywords: Model-Driven Engineering · Scalability
Model analytics · Data mining · Machine learning

1 Introduction

Model-Driven Engineering (MDE) promotes the use of models, metamodels and model transformations as first-class citizens to tackle the complexity of software systems. As MDE is applied to larger problems, the complexity, size and variety of those artefacts increase. With respect to model size and complexity, for instance, the aspect of scalability has been pointed out by Kolovos et al. [18]. Regarding this aspect, a good amount of research has been done for handling a small number of (possibly very big and complex) models, e.g. in terms of comparison, merging, splitting, persistence or transformation. However, scalability with respect to model variety and multiplicity (i.e. dealing with a large number of possibly heterogeneous models) has so far remained mostly under the radar.

In this paper, we advocate this aspect of scalability as a potentially big challenge for broader MDE adoption. We highlight evidence and concerns which cross-cut the dichotomies of industry vs. academia and of open source vs. commercial software. We thus show that scalability proves to be an issue overall. Furthermore, we mention several related domains and disciplines as inspiration for tackling scalability, with pointers to some related work. Yet we note the

inherent differences of MDE artefacts (models in particular), compared to common types of data such as natural language text and source code. This might render it difficult to directly apply techniques from other domains to MDE.

2 The Expanding Universe of MDE

The aforementioned scalability issue emerges partly due to some recent developments in the MDE community. Firstly, there have been efforts to initiate public repositories to store and manage large numbers of models and related artefacts [5, 23]. Further efforts include mining public repositories for MDE-related items from GitHub, e.g. Eclipse-based MDE technologies [17] and UML models [14] (the Lindholmen dataset). In the latter, the number of UML models can go up to more than 90k. The sheer number of models inevitably calls for techniques for searching, preprocessing (e.g. filtering), analysing and visualising the data in a holistic and efficient manner.

Mini Study: Ecore Metamodels in GitHub. Kolovos et al. present a study on the use of MDE technologies in GitHub [17]. Among a rich set of empirical results, they report the number of search results for Ecore metamodels in GitHub (as of early 2015) to be ~ 15 k, and show a rapidly increasing trend in the number of commits on MDE-related files. We were triggered by the fact that the same exercise of searching Ecore files, with the query reported in the paper (<https://github.com/search?q=EClass+extension:ecore&type=Code>) yields (as of September 2017) more than 67k results; a fourfold increase. Here we demonstrate a mini case study on the Ecore metamodel files in GitHub over time. We slightly relaxed the query by replacing the search term *EClass* with *ecore*, and mined all the files together with the relevant commits on them. Figure 1 depicts a strong upward trend for (a) the number of commits per year on Ecore files, and (b) the number of newly created Ecore files per year. The sharp increase in 2017 can be partly attributed to recently emerging metamodel repositories and datasets on GitHub; nevertheless as the end effect there are increasingly more Ecore metamodels in GitHub.

MDE in the Industry. Even within a single industry or organisation, a similar situation emerges with increased adoption of MDE. We have been collaborating with high tech companies in the Netherlands. One of those companies maintains a set of MDE-based domain-specific language (DSL) ecosystems. Just a single one of those ecosystems currently contains dozens of metamodels, thousands of models and tens of thousands LOC of transformations. With the complete revision history, the total number of artefacts goes up to tens of thousands. Another company, which applies MDE in six different projects, reports a similar collection of thousands of artefacts based on various technologies (e.g. different transformation languages). Similar stories in terms of scale hold for our other industrial partners, with growing heterogeneous sets of artefacts involving multiple domains. Note that for systems with implicit or explicit (e.g. as a Software Product Line) variability, *variants* can be considered another amplifying factor besides *versions* for the total number of MDE artefacts to manage.

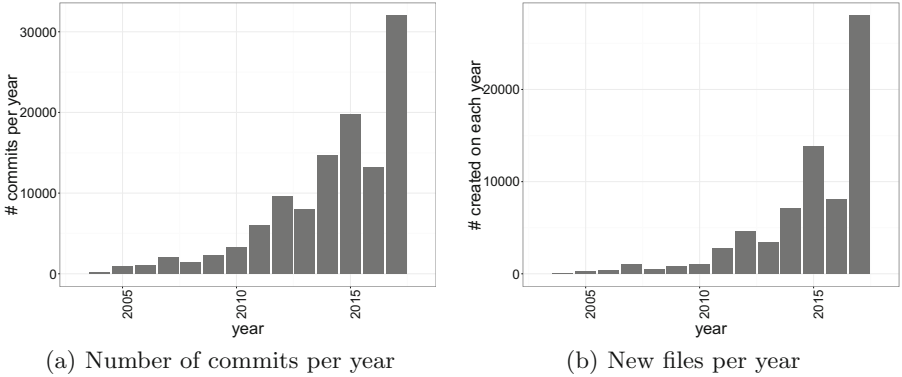


Fig. 1. GitHub results on Ecore metamodels.

Along with conventional forward engineering approaches, we observe an increasing trend in our partners with legacy software: automated migration into model-driven/-based engineering using process mining and model learning. All the presented facts let us confirm the statement by Brambilla et al. [8] and Whittle et al. [24] that MDE adoption in (at least some parts of) the industry grows quite rapidly, and we conclude that tackling scalability will be increasingly important in the future.

3 Treating MDE Artefacts as Data

Based on the observations above, we advocate a perspective where MDE artefacts are treated holistically as data, processed and analysed with various scalable and efficient techniques, possibly inspired by related domains. Tackling large volumes of artefacts has been commonplace in other domains, such as text mining for natural language text [15], and repository mining for source code [16]. While we might not be able to apply those techniques as-is on MDE-related artefacts, the general workflow remains as a rough guideline with several steps of data collection, cleaning, integration and transformation, feature engineering and selection, modelling (e.g. as statistical models, neural networks), and finally deployment, exploration and visualisation.

To exemplify the different nature of MDE data and hence the different requirements on its analysis, take the problem of clone detection. Clone detection on source code is already several steps away from text mining, as code clone detection usually involves steps such as a language-specific parsing of the code into abstract syntax trees (AST), normalisation of identifiers, and structural transformations [21]. Model clone detection, on the other hand, possess further challenges. To cite Deissenboeck et al., “*Algorithms from code clone detection are only of minor interest for model clone detection, as they usually work on either a linear text or token stream or on the tree structured AST of the code, which both are not transferable to general directed graphs.*” [11]. Furthermore,

Störrle points out inherent differences of models compared to code, including CASE tool integration and tool-specific representations, internal identifiers and different layouts with no semantic implications, abstract vs. (possibly multiple) concrete syntaxes, etc. [22]. The case of clone detection reinforces our argument that techniques from related domains such as data mining and repository mining might not be directly translatable to the MDE domain.

4 Relevant Domains for Model Analytics

Despite the different nature of models as exemplified above, we can get inspired by the techniques from other disciplines and try to adapt them for the problems in MDE. As a preliminary overview, in this section we list and discuss several such domains. While there is related MDE research on some of the items on the list, we believe a conscious and integrated mindset would mitigate the challenges for scalable MDE.

Descriptive Statistics. Several MDE researchers have already performed empirical studies on MDE artefacts with a statistical mindset. For instance, Kolovos et al. assesses the use of Eclipse technologies in GitHub, giving related trend analyses [17]. Mengerink et al. present an automated analysis framework on version control systems with similar capabilities [19]. Di Rocco et al. perform a correlation analysis on metrics for various MDE artefacts [12]. Descriptive statistics could in the most general sense be exploited to gain insights over large numbers of MDE artefacts in terms of general characteristics, patterns, outliers, statistical distributions, dependence, etc.

Information Retrieval. Techniques from information retrieval (IR) can facilitate indexing, searching and retrieving of models, and thus their management and reuse. The adoption of IR techniques on source code dates back to the early 2000s, and within the MDE community there has been some recent effort in this direction (e.g. by Bislimovska et al. [7]). Further IR-based techniques can be found in [2, 4] involving repository management and model searching scenarios.

Natural Language Processing. Accurate Natural Language Processing (NLP) is needed to handle realistic models with noisy text content, compound words, and synonymy/polysemy. In our experience, it is very problematic to blindly use NLP tools on models, e.g. just WordNet synonym checking without proper part-of-speech tagging and word sense disambiguation. More research is needed to find the right chain of NLP tools applicable for models (in contrast to source code and documentation), and reporting accuracies and disagreements between tools (along the lines of the recent report in [20] for repository mining). Note that NLP offers further advanced tools, such as language modelling, which are still to be investigated for MDE.

Data Mining. Following the perspective of approaching MDE artefacts as data, we need scalable techniques to extract relevant units of information from models (*features* in data mining jargon), and to discover patterns including domain clusters, outliers/noise and clones. Several example applications can be found in [1, 2, 4]. To analyse, explore and eventually make sense of the large datasets in MDE (e.g. the Lindholmen dataset), we can investigate what can be borrowed from comparable approaches in data mining for structured (graph) data.

Machine Learning. The increasing availability of large amounts of MDE data can be exploited, via machine learning, to automatically infer certain qualities and functions. There has been a thrust of research in this direction for source code (e.g. for fault prediction [10]), and it would be noteworthy to investigate the emerging needs of the MDE communities and feasibility of such learning techniques for MDE. The approach in [3] for learning model transformations by examples is one of the few pieces of such work in MDE.

Visualization. We propose visualization and visual analytics techniques to inspect a whole dataset of artefacts (e.g. cluster visualizations in [4], in contrast with visualizing a single big model in [18]) using various features such as metrics and cross-artefact relationships. The goals could range from exploring a repository to analysing an MDE ecosystem holistically and even studying the (co-)evolution of MDE artefacts.

Distributed/Parallel Computing. With the growing amount of data to be processed, employing distributed and parallel algorithms in MDE is very relevant. There are conceptually related approaches in MDE worthwhile investigating, e.g. distributed model transformations for very large models [6, 9] or model-driven data analytics [13]. Yet we wish to draw attention here to performing computationally heavy data mining or machine learning tasks for large MDE datasets in an efficient way.

We propose this non-exhaustive list as a preliminary exploitation guideline to help tackling scalability in MDE. Although the aforementioned domains themselves are quite mature on their own, it should be investigated to what extent results and approaches can be transferred into the MDE technical space.

5 Conclusion

We observe a rapid increase in the size of the MDE universe, both in open source and industry, which leads to scalability issues yet to be addressed by the community. To overcome this new and relatively overlooked challenge, we propose a holistic research perspective with several components, ranging from information retrieval to machine learning. We believe that approaches towards this direction already matter, but will increasingly be more important for the successful widespread use of MDE.

Acknowledgments. This work is supported by the 4TU.NIRICT Research Community Funding on Model Management and Analytics in the Netherlands.

References

1. Babur, Ö.: Statistical analysis of large sets of models. In: 31th IEEE/ACM International Conference on Automated Software Engineering (ASE 2016), Singapore, 3–7 September 2016 (2016)
2. Babur, Ö., Cleophas, L., van den Brand, M.: Hierarchical clustering of metamodels for comparative analysis and visualization. In: 2016 Proceedings of the 12th European Conference on Modelling Foundations and Applications, pp. 2–18 (2016)
3. Baki, I., Sahraoui, H.A.: Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Trans. Softw. Eng. Methodol.* **25**(3), 20:1–20:37 (2016). <https://doi.org/10.1145/2904904>
4. Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Automated clustering of metamodel repositories. In: Nurcan, S., Soffer, P., Bajec, M., Eder, J. (eds.) CAiSE 2016. LNCS, vol. 9694, pp. 342–358. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39696-5_21
5. Basciani, F., Di Rocco, J., Di Ruscio, D., Di Salle, A., Iovino, L., Pierantonio, A.: MDEFForge: an extensible web-based modeling platform. In: Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud Co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, CloudMDE@MoDELS 2014, Valencia, Spain, 30 September 2014, pp. 66–75 (2014). <http://ceur-ws.org/Vol-1242/paper10.pdf>
6. Benellallam, 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. ACM (2015)
7. Bislmovska, B., Bozzon, A., Brambilla, M., Fraternali, P.: Textual and content-based search in repositories of web application models. *TWEB* **8**(2), 11:1–11:47 (2014). <https://doi.org/10.1145/2579991>
8. Brambilla, M., Cabot, J., Wimmer, M.: *Model-Driven Software Engineering in Practice*, 1st edn. Morgan & Claypool Publishers, San Rafael (2012)
9. Burgueño, L., Wimmer, M., Vallecillo, A.: *Towards distributed model transformations with LinTra* (2016)
10. Catal, C., Diri, B.: A systematic review of software fault prediction studies. *Expert Syst. Appl.* **36**(4), 7346–7354 (2009)
11. Deissenboeck, F., Hummel, B., Juergens, E., Pfaehler, M., Schaetz, B.: Model clone detection in practice. In: Proceedings of the 4th International Workshop on Software Clones, pp. 57–64. ACM (2010)
12. 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, New York (2014). <http://doi.acm.org/10.1145/2593770.2593774>
13. Hartmann, T., Moawad, A., Fouquet, F., Nain, G., Klein, J., Traon, Y.L., Jezequel, J.M.: Model-driven analytics: connecting data, domain knowledge, and learning. arXiv preprint [arXiv:1704.01320](https://arxiv.org/abs/1704.01320) (2017)
14. Hebig, R., Ho-Quang, T., Chaudron, M.R.V., Robles, G., Fernández, M.A.: The quest for open source projects that use UML: mining GitHub. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, 2–7 October 2016, pp. 173–183 (2016). <http://dl.acm.org/citation.cfm?id=2976778>

15. Hotho, A., Nürnberger, A., Paass, G.: A brief survey of text mining. *LDV Forum* **20**(1), 19–62 (2005). http://www.jlcl.org/2005_Heft1/19-62_HothoNuernbergerPaass.pdf
16. Kagdi, H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.* **19**(2), 77–131 (2007). <https://doi.org/10.1002/smr.344>
17. 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: *OSS4MDE@ MoDELS*, pp. 20–29 (2015)
18. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, 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, BigMDE 2013*, pp. 2:1–2:10. ACM, New York (2013). <http://doi.acm.org/10.1145/2487766.2487768>
19. Mengerink, J.G., Serebrenik, A., Schiffelers, R.R., van den Brand, M.G.: Automated analyses of model-driven artifacts. *IWSM Mensura* (2017, to appear)
20. Omran, F.N.A.A., Treude, C.: Choosing an NLP library for analyzing software documentation: a systematic literature review and a series of experiments. In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, 20–28 May 2017*, pp. 187–197 (2017). <https://doi.org/10.1109/MSR.2017.42>
21. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Sci. Comput. Program.* **74**(7), 470–495 (2009). <http://www.sciencedirect.com/science/article/pii/S0167642309000367>
22. Störrle, H.: Towards clone detection in UML domain models. *Softw. Syst. Model.* **12**(2), 307–329 (2013)
23. Störrle, H., Hebig, R., Knapp, A.: An index for software engineering models. In: *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) Co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, 28 September–3 October 2014*, pp. 36–40 (2014). <http://ceur-ws.org/Vol-1258/poster8.pdf>
24. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. *IEEE Softw.* **31**(3), 79–85 (2014)

On the Need for Temporal Model Repositories

Robert Bill¹, Alexandra Mazak¹, Manuel Wimmer¹(✉),
and Birgit Vogel-Heuser²

¹ CDL-MINT, TU Wien, Vienna, Austria
{bill,mazak,wimmer}@big.tuwien.ac.at

² Technical University of Munich, Munich, Germany
vogel-heuser@ais.mw.tum.de

Abstract. Current model repositories often rely on existing versioning systems or standard database technologies. These approaches are sufficient for hosting different versions of models. However, the time dimension is often not explicitly represented and accessible. A more explicit presentation of time is needed in several use cases going beyond the classical system design phase support of models such as in simulation and runtime environments.

In this paper, we discuss the need for introducing temporal model repositories and their prospective benefits. In particular, we outline several challenges which immediately arise when moving towards temporal model repositories, which are: storage, consistency, access, manipulation, and visualization of temporal models.

1 Introduction

Model repositories are a crucial infrastructure for applying model-driven engineering (MDE) in practical settings [10]. Current and past research works concerning model repositories comprise mainly two areas: (*i*) concurrent modeling using a central repository to coordinate the editing of models [2], and (*ii*) scalability in storing and retrieving models [18].

In this challenge paper, we are demonstrating why temporal model repositories are needed to cope with emerging use cases in the general field of MDE which require models to be evolutionary artifacts leading to the new notion of “liquid models” [19]. Furthermore, such repositories are highly needed in application fields where models have to be used throughout the complete system life-cycle such as in production systems engineering [26].

1.1 Why Temporal Model Repositories are Needed

Previous research has been focusing on storing models in model repositories such as SVN and Git using XMI serializations [3] as well as in database technologies such as relational databases, graph databases, or tuple stores [5]. Traditionally, each model version of an evolving model is stored as self-contained model instance. These differently stored model versions allow to reason about

evolution concerns. However, the temporal aspect is not explicitly targeted on the model element level. In order to reason about questions considering specific model elements, the different versions have first to be aligned and compared before one can actually reason about temporal aspects such as model element changes. Moreover, storing full model states for each version is not efficient. Just consider changing one value between two model versions. This would result in mostly two identical models which have to be stored. This clearly shows that historical model information is currently not well supported by existing model repositories.

1.2 Existing Work on Temporal Artifacts

There are several approaches which consider the explicit support of a temporal dimension. In particular, there is work on temporal databases, model versioning and behavioral model verification. Initial work has been done in the area of temporal relational databases [25]. This work has been continued in the area of temporal data warehouses [15]. Additionally, these works have resulted in the explicit support of temporal SQL in many existing relational database systems. In the field of programming languages work exists on temporal objects¹. There, the goal is to store the historical states of an object in addition to the current state. Finally, there exists interesting work in the area of CAD engineering tools and accompanying datastores. Especially, multi-version object-oriented databases allow for revisions to model evolution in time and variants to model parallel ways of development [12, 23].

Model behavior verification is a typical use case for temporal artifacts. To verify the behavior of MDE artifacts, model checking is one of the most used techniques. Reasons for that might include that the state space used in model checking, which includes states and transitions, can be easily interpreted as including different model versions as states and model changes as transitions. This closely resembles a revision graph used in model versioning systems.

Firstly, there are verification languages that extend existing languages like OCL by classical model checking languages such as CTL* or the μ -calculus (e.g., [21]). Secondly, dedicated languages using patterns have been introduced to reduce the mental load to specify temporal properties, e.g., [9, 16] for OCL. Beside employing new constraint language extensions for verification, the consistency between two or more behavioral UML models might be checked (e.g., [24]) as well. Simple safety properties may be formulated by reusing existing model query formalisms. Model checking implementations typically are either state-based (e.g., [17]) or translate the model checking problem to another formalism (e.g., [14]). The behavior itself may be defined by graph transformations, behavioral diagrams or plain pre-conditions and post-conditions. Another research line is the development of domain-specific property languages which allow to reason about temporal properties of systems [20].

¹ <https://martinfowler.com/eaDev/timeNarrative.html>.

1.3 Structure of this Paper

First, we present a motivating example in the context of systems engineering which requires temporal information about an evolving system. Then, we present several challenges which arise by switching from single-version models to multi-version models for representing evolutions of a model. Finally, we conclude the paper by discussing next steps.

2 Motivating Example

Consider a system engineering example based on a simple block modeling language to model machines and sensors (cf. Fig. 1a). A sensor can be attached to any machine and this may change over time as measurements have to be collected for each machine in certain periods of time. Of course, the model may just reflect the current state (Sensor 1 is connected to Machine 3 in T_i). However, we have context dependent information such as measurement errors and down times of the sensor which depend on the particular machine the sensor is located (Sensor 1 was connected to Machine 2 in T_{i-1} and to Machine 1 in T_{i-2}). In order to support this scenario, we do not only need the two traditional dimensions (*i*) representing machines and sensors as blocks and (*ii*) their feature values, but add time as third dimension to store feature value history.

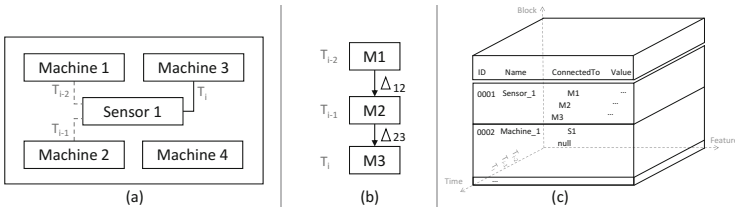


Fig. 1. Dealing with temporal models: (a) concrete example, (b) traditional repository view, (c) temporal repository view

Figure 1b shows the current state-of-the-art of storing evolving models in model repositories. Since, we cannot represent different versions of a model within one model, all the model versions have to be stored separately. Figure 1c shows a more condensed representation by introducing an explicit time dimension which allows to represent several versions of a model by providing different values with time stamps. Please note that it is feasible to construct Fig. 1c from the information provided by Fig. 1b. The benefit of having the representation provided by Fig. 1c is the direct representation of and the access to temporal information, which will be demonstrated in the following section.

3 Temporal Model Repositories Challenges

In this section, we outline the main challenges encountered by moving towards temporal model repositories demonstrated by different use cases. In particular, these challenges encompass (i) model storage, (ii) model consistency, (iii) model access, (iv) model manipulation, and (v) model visualization. While these topics might impose challenges for large single models as well, they drastically increase when incorporating time.

3.1 Model Storage

Figure 2 illustrates different levels of granularity for model storage. We use this figure to outline potential strategies to cope with models evolving in time by certain examples. Many model simulations as well as models@runtime [7] approaches store each snapshot separately leading to high memory requirements and potentially suboptimal performance. Thus, use cases like conducting change analysis on large models cannot be performed. However, they provide simple and fast access to each snapshot. For example, without further meta-information, which might need to be calculated, it is not possible to even detect that the value of the sensor decreased and the machine m1 was deleted (see Fig. 2a).

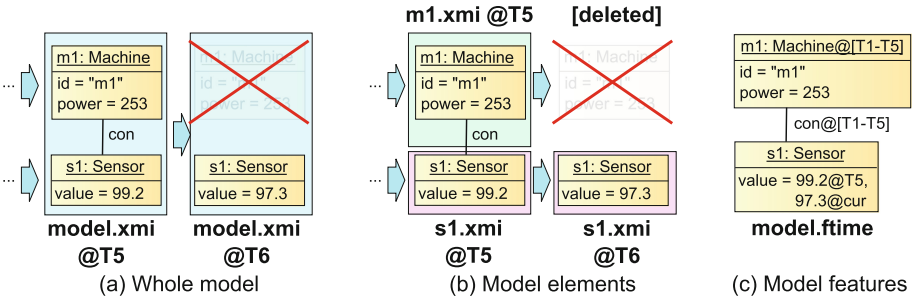


Fig. 2. Model storage granularity levels

Metamodeling frameworks such as the Eclipse Modeling Framework (EMF) would allow to store each model element in a separate file. By this, matching processes are simplified and object deletions are immediately detected. For example, deleting m1 results in a deletion of the corresponding file m1.xmi (see Fig. 2b). However, such an approach results in a potentially unwanted scattering of the model in many files and thus, model loading times are increasing. Approaches based on storing models in single or multiple xmi files are also not able to describe the continuous change of models. By reducing the granularity of time-dependency from model level to model element level, or even feature level, the required memory may be reduced, especially when performing many small changes on models.

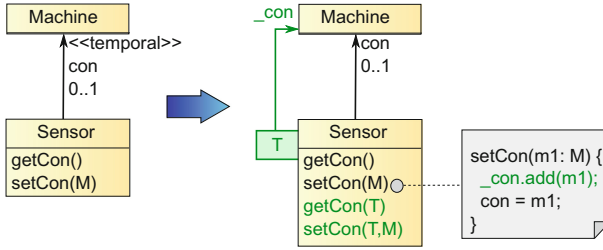


Fig. 3. Metamodel augmentation by temporal property pattern

Figure 2c shows how feature-level-granularity helps describing time-dependant models. Instead of having multiple files, a time-dependant model may consist of just one file where each element and feature value has a certain timespan. Therefore, small model edits are just small model extensions. Also, feature values may be described as time-dependent functions in discrete or continuous time. Continuous functions may be derived from approximations or interpolations of a certain set of discrete time measurements values or a known temporal behavior. However, no matter whether models are stored in XMI or temporal databases, means are needed to reduce the overhead of the storage strategies discussed above (Fig. 2) when accessing a single model. For object-oriented languages, there is the temporal property pattern² which might be useful for models too.

Figure 3 shows how a metamodel may be enriched to support temporal features by providing additional getter and setter methods to access historical information. These methods contain a time parameter which allow us to (i) access historic, potentially interpolated, values, and (ii) set values at a specific historic point in time. Values might be stored in a sorted map with time as index. If a value set is then the most recent one, the current value accessible in *con* is updated too (see Fig. 3). Additionally, we assume that setting a value with the conventional set method sets the current value. In this respect, a promising approach to introduce temporal storage capabilities to models has been presented in [11]. This approach leads to an interesting research question in how this approach may be lifted to the metamodel level.

3.2 Model Consistency

Time also requires new types of model constraints. We have to differentiate between general semantic constraints which are equal for all models and new kinds of model-specific temporal constraints. The general constraints typically arise from syntactic errors in conventional models (e.g., having a reference to a deleted model element) which become semantic errors in time-aware models. The constraints shown in Table 1 are model-specific. The simplest kind of expressions

² <https://martinfowler.com/eaDev/TemporalProperty.html>.

Table 1. Various types of temporal constraints with examples

Exp. type	Property	Constraint in extended OCL
Logical	Ids never change	<code>id = id@init</code>
Statistical	The relative standard deviation of the power consumption never exceeds 1%	<code>let p = power@always in p->stdDev() < p->avg()*0.01</code>
Valid changes	Whenever a machine name changes, the machine is physically transferred to a different factory at the same time	<code>name@next->contains(name) = factory@next-> contains(factory)</code>
Valid traces	When a machine fails, it will be either working in at most three days or not at all	<code>Always Globally not working implies ((Finally[3] working) or (Globally not working))</code>

are logical expressions resulting from method pre- and postconditions. In the case that models are only changed by calling operations, the constraint that *Ids never change* (see Table 1, Logical) could be expressed by requiring that ids of every object remain unchanged after the operation call. In production systems engineering, one might be interested in potential error sources. For example, power spikes might reduce the lifespan of a machine and thus one could be interested in constraints which ensure some quality properties of a virtual factory (see Table 1, Statistical). In this context, one might be interested that employees don't do unreasonable things like changing machine names without cause (see Table 1, Valid changes). Or, that repair contracts which guarantee that a machine works again after at most three days should be fulfilled (see Table 1, Valid traces).

3.3 Model Access

Textual query languages, like OCL, are often used to aggregate model information, whereas graph pattern based query languages are typically used to search for specific content. There is a number of languages to (i) retrieve statistical information and (ii) verify the behavior of models [6, 20]. However, it is still an open question how these languages may be integrated with current model repositories for querying temporal models. Figure 4a shows a potential temporal query pattern to detect unusually high sensor measurement values. In this case, a sensor which should exist in timepoint *T1* has values for at least five points in time while being connected to the machine *m1* which are higher than ever observed by this sensor before. Figure 4b shows a potential answer to this query, where the time point *T1* was matched to 3. No specific timepoint for *m1* in the pattern was given, so 7 was chosen by the engine for the result object *om1*.

The main difference to conventional model query outcomes is that the resulting model includes time meta-information for each element.

The query execution efficiency is paramount for temporal models as the models do not only grow in size, but also in the number of versions. In particular, incremental evaluation seems beneficial as model changes occur which often affect small parts of the model. Also, for model analysis and simulation, intermediate models might be interesting only regarding to certain queries. Thus, means for automatically storing model fragments relevant for the evaluation of a time-based query have to be developed.

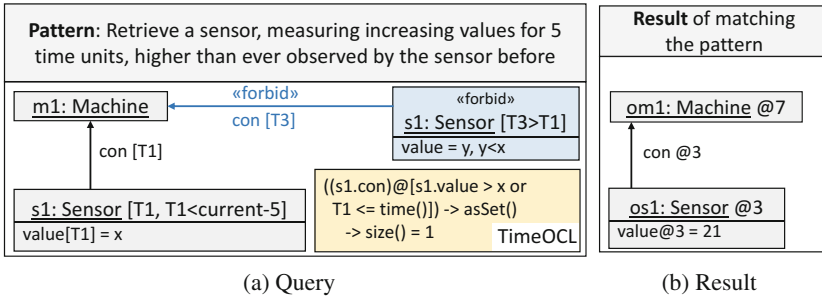


Fig. 4. Time-based model pattern and answer

3.4 Model Manipulation

Models may be manipulated with model transformations and/or a model API. Model queries can be extended to model transformations by adding actions. Furthermore, they may need an additional parameter for the current time. Figure 5 shows an example of a transformation which connects a sensor to a machine. This machine must never have been connected to a sensor that currently exists. Since all transformations are applied to change the current model, actions do not necessarily need to have a time parameter. A time-conscious model API should be close to the original model API, but also allow to easily travel in time for an object. Several semantic issues have to be resolved, e.g., the meaning of adding any object in $T2$ to a feature value of an object in $T1$. This could be interpreted in many ways, e.g., (i) doing nothing if timestamps are different, (ii) connecting the object only in $T1$, or (iii) connecting the object in the whole period between $T1$ and $T2$. Time-based model transformations could be highly useful for simulation purposes [22] and may lead to benefits of representing execution traces [8] in a very condensed form.

3.5 Model Visualization

For some use cases, models will have to incorporate their history in their concrete syntax, e.g., links may be displayed stronger if they were active for most of the

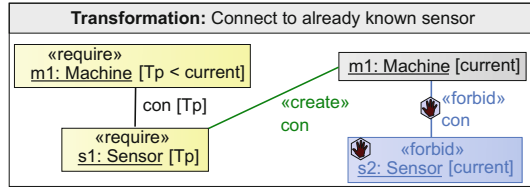


Fig. 5. History-driven model transformation

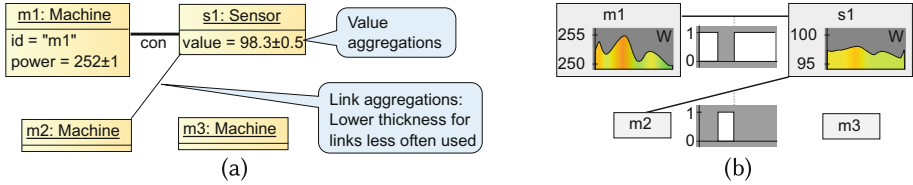


Fig. 6. Time-oriented model visualisation: (a) abstract syntax and (b) concrete syntax

time, values may be replaced by statistical information, and so on. Figure 6 shows a possible realization of such a syntax. Simple object diagram extensions as depicted might combine the familiarity of object diagrams with the display of additional information. However, displaying the complete history of a model in a single diagram puts some pressure on layouting methods. For example, in order to encompass more objects which might not always be rendered depending on the selected timespan. For some languages additional annotations, profiles, or extensions may help. There is a variety of data visualization techniques available, which may be used to visualize the development of single features, small model parts, or the whole model [1]. Such visualization methods may also be useful to augment concrete graphical syntaxes of models as shown in the right part of Fig. 6. In particular, temporal model query languages should provide a potential source for model visualization.

3.6 Further Challenges

Not only models have a dynamic nature, also metamodels may change over time (e.g., see multi-level modeling [4]). Furthermore, there may exist more than “just” one successor for a model version, which requires support for branching. Having temporal model repositories at hand, the extension to deductive temporal model repositories would be beneficial to learn from model evolution. Last but not least, there exists no unique model of time. Due to the different dynamics of components, the distinction between different (coexisting) “notions of time” is required [13]. Consider the two following distinctions of time for a single device (e.g., a sensor): (i) “time” implicitly defined as sequence of logs resulting from the device’s operation (i.e., external time) and (ii) “time” defined and measured by the device’s clock, whose measure of time may be recorded like other program variables (i.e., internal time) [13].

4 Conclusions

In this paper, many interesting research challenges are posed in the young research area of temporal models and temporal model repositories. We have identified the following steps where work is needed to meet the demanding for a common terminology and an exact formulation of the research goals in this area. This also requires to work bottom-up to better understand the concrete requirements in use cases relying on temporal information as well as top-down by studying the literature provided by other research communities in related fields such as databases and programming languages. We are looking forward to investigate these open issues together with the MDE research community.

Acknowledgments. This work has been funded by the Austrian Federal Ministry of Science, Research and Economy (BMWFV) and the National Foundation for Research, Technology and Development (CDG).

References

1. Aigner, W., Miksch, S., Schumann, H., Tominski, C.: Visualization of Time-Oriented Data: Human-Computer Interaction Series. Springer, Heidelberg (2011). <https://doi.org/10.1007/978-0-85729-079-3>
2. Altmanninger, K., Brosch, P., Langer, P., Seidl, M., Wiel, K., Wimmer, M.: Why model versioning research is needed!? an experience report. In: Proceedings of MoDSE-MCCM Workshop, pp. 1–12 (2009)
3. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. *IJWIS* **5**(3), 271–304 (2009)
4. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: Gogolla, M., Kobryn, C. (eds.) *UML 2001*. LNCS, vol. 2185, pp. 19–33. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45441-1_3
5. Barmpis, K., Kolovos, D.S.: Comparative analysis of data persistence technologies for large-scale models. In: Proceedings of Extreme Modeling Workshop, pp. 33–38 (2012)
6. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: Model checking of CTL-extended OCL specifications. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) *SLE 2014*. LNCS, vol. 8706, pp. 221–240. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_13
7. Blair, G., Bencomo, N., France, R.B.: Models@run.time. *Computer* **42**(10), 22–27 (2009)
8. Bousse, E., Mayerhofer, T., Combemale, B., Baudry, B.: A generative approach to define rich domain-specific trace metamodels. In: Taentzer, G., Bordeleau, F. (eds.) *ECMFA 2015*. LNCS, vol. 9153, pp. 45–61. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21151-0_4
9. Bradfield, J., Filipe, J.K., Stevens, P.: Enriching OCL using observational Mu-Calculus. In: Kutsche, R.-D., Weber, H. (eds.) *FASE 2002*. LNCS, vol. 2306, pp. 203–217. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45923-5_14
10. Brambilla, M., Cabot, J., Wimmer, M.: *Model-Driven Software Engineering in Practice: Synthesis Lectures on Software Engineering*, 2nd edn. Morgan & Claypool Publishers, San Rafael (2017)

11. Cabot, J., Olivé, A., Teniente, E.: Representing temporal information in UML. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 44–59. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45221-8_5
12. Cellary, W., Vossen, G., Jomier, G.: Multiversion object constellations: a new approach to support a designer’s database work. *Eng. Comput.* **10**(4), 230–244 (1994)
13. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling time in computing: a taxonomy and a comparative survey. *ACM Comput. Surv.* **42**(2), 1–59 (2010)
14. Gogolla, M., Hilken, F., Doan, K., Desai, N.: Checking UML and OCL model behavior with filmstripping and classifying terms. In: Proceedings of TAP, pp. 119–128 (2017)
15. Golfarelli, M., Rizzi, S.: Temporal data warehousing: approaches and techniques. In: Integrations of Data Warehousing, Data Mining and Database Technologies, pp. 1–18 (2011)
16. Kanso, B., Taha, S.: Temporal constraint support for OCL. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 83–103. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36089-3_6
17. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006). https://doi.org/10.1007/11691617_19
18. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of BigMDE, pp. 2:1–2:10 (2013)
19. Mazak, A., Wimmer, M.: Towards liquid models: an evolutionary modeling approach. In: Proceedings of CBI, pp. 104–112 (2016)
20. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., Wimmer, M.: ProMoBox: a framework for generating domain-specific property languages. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 1–20. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_1
21. Mullins, J., Oarga, R.: Model checking of extended OCL constraints on UML models in SOCLE. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 59–75. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72952-5_4
22. Rivera, J.E., Romero, J.R., Vallecillo, A.: Behavior, time and viewpoint consistency: three challenges for MDE. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 60–65. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01648-6_7
23. Rykowski, J., Cellary, W.: Using multiversion object-oriented databases in CAD/CIM systems. In: Wagner, R.R., Thoma, H. (eds.) DEXA 1996. LNCS, vol. 1134, pp. 1–10. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0034665>
24. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. *Electr. Notes Theor. Comput. Sci.* **55**(3), 357–369 (2001)
25. Tansel, A.U., Clifford, J., Gadia, S., Jajodia, S., Segev, A., Snodgrass, R. (eds.): Temporal Databases: Theory, Design, and Implementation (1993)
26. Vogel-Heuser, B., Fay, A., Schaefer, I., Tichy, M.: Evolution of software in automated production systems: challenges and research directions. *JSS* **110**, 54–84 (2015)

On the Need for Artifact Models in Model-Driven Systems Engineering Projects

Arvid Butting, Timo Greifenberg^(✉), Bernhard Rumpe,
and Andreas Wortmann

Software Engineering, RWTH Aachen University, Aachen, Germany
greifenberg@se-rwth.de
<http://www.se-rwth.de>

Abstract. Model-driven development has shown to facilitate systems engineering. It employs automated transformation of heterogeneous models into source code artifacts for software products, their testing, and deployment. To this effect, model-driven processes comprise several activities, including parsing, model checking, generating, compiling, testing, and packaging. During this, a multitude of artifacts of different kinds are involved that are related to each other in various ways. The complexity and number of these relations aggravates development, maintenance, and evolution of model-driven systems engineering (MDSE). For future MDSE challenges, such as the development of collaborative cyber-physical systems for automated driving or Industry 4.0, the understanding of these relations must scale with the participating domains, stakeholders, and modeling techniques. We motivate the need for understanding these relations between artifacts of MDSE processes, sketch a vision of formalizing these using artifact models, and present challenges towards it.

1 Motivation

The complexity of future interconnected cyber-physical systems, such as the smart future, fleets of automated cars, or smart grids poses grand challenges to software engineering. These challenges partly arise from the number of domains, stakeholders, and modeling techniques required to successfully deploy such systems. Model-driven engineering has shown to alleviate this, but introduces the challenge of managing the multitude of different artifacts, such as configuration, models, templates, transformations, and their relations as contributed by the experts of different domains. Considering, for instance, software engineering for the factory of the future [10], successful deployment of a virtual factory [9] requires integration of modeling techniques to describe the factory's geometry, production processes and their optimization, software architecture, production systems with their interaction, manufacturing knowledge, and, ultimately,

This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS16043P. The responsibility for the content of this publication is with the authors.

general-purpose programming language artifacts. The artifacts contributed by the respective domain experts are required in different stages of the development process and exhibit various forms of relations, such as design temporal dependencies, run-time dependencies, or creational dependencies (*e.g.*, a model and the code generated from it).

Moreover, how artifacts interrelate not only depends on their nature, but also on the context they are used in and the tools they are used with. For instance, software architecture models may be used for communication and documentation, model checking, transformation into source code, or simulation of the system part under development. In these contexts, the relations required to understand and process such an artifact may change: whereas the pure architecture model might be sufficient for communicating its structural properties, transformation into source code relates it to transformation artifacts and to the artifacts produced by this transformation.

This paper extends previous work [5] in greater detail for a better understanding on how explicating these artifacts and their relations facilitates traceability of artifacts, change impact analysis [1], and interoperability of software tools all of which are crucial to successful model-driven engineering of the future's systems of systems.

2 Modeling Artifact Relations

Typical MDSE projects require a multitude of different artifacts addressing the different domains' concerns (cf. Fig. 1). Managing the complexity of these artifacts requires understanding their relations, which entails understanding the relations between their languages as well as between the development tools producing and processing artifacts. We envision a MDSE future in which these relations are made explicit and machine-processable using modeling techniques. To this end, we desire reifying this information as first-level modeling concern in form of an explicit *artifact model* defined by the lead architect of the overall MDSE project. Such a model precisely specifies the kinds of artifacts, tools, languages, and relations present in the MDSE project and thus enables representing the MDSE project in a structured way.

Such an artifact model should be capable to describe all different situations in terms of present artifacts and relations that could arise during its lifetime. The current situation of the project can be inspected by automatically extracting artifact data from the project according to the artifact models' entities and relations. This data corresponds to the artifact model ontologically, *i.e.*, represents an instance of it at a specific point in time. Analysts or specific software tools can employ this data to produce an overview of the current state, reporting issues, and identifying optimization potentials. Ultimately, this aims at enabling a more efficient development process.

To this end, the artifact model comprises, among others, the organization of artifacts in the file system, the configuration of the tool chain, the trace of the last tool chain execution as well as static knowledge relations between artifacts

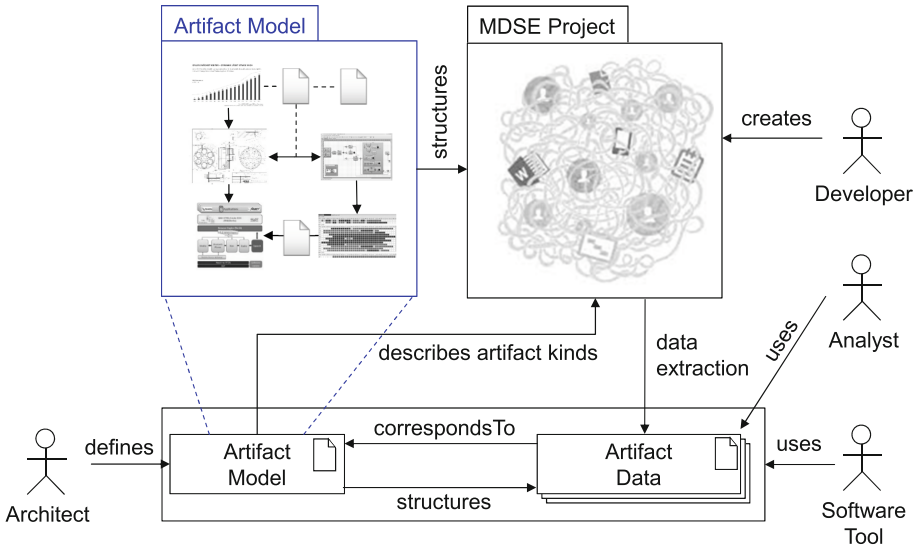


Fig. 1. An artifact model structures the different kinds of artifacts within MDSE projects. Corresponding artifact data enables analyses of the project state by analysts and software tools.

leading to an architectural view including input models, artifacts forming specific tools or the target product, artifacts managed by the tools, output artifacts, and handcrafted artifacts.

This model depends on the technologies and tools used to develop the target product. Hence, it must be tailored specifically to each MDSE project. Globally, parts of such a model could be reused from similar projects (which might be achieved employing language engineering and composition methods on the artifact modeling language). For instance, model parts describing the interfaces of tools could be reusable as well as the types of specific artifacts and their relations might be applicable to multiple projects. Nevertheless, we assume each project will require manual artifact modeling to adjust existing structures. Ultimately, creating such an artifact model would

- ease communication, specification, and documentation of artifact, tool, and language dependencies,
- enable automated dependency analysis between artifacts and tools,
- support change impact analysis in terms of artifact tool, or language changes,
- support checking compliance of tools and proposing artifact, tool, and relation adaptations to 'glue' tool chains,
- facilitate an integrated view on the usage of tools in concrete scenarios,
- enable data-driven decision making, and
- enable computation of metrics and project reports to reveal optimization potentials within the tool chain and the overall MDSE process.

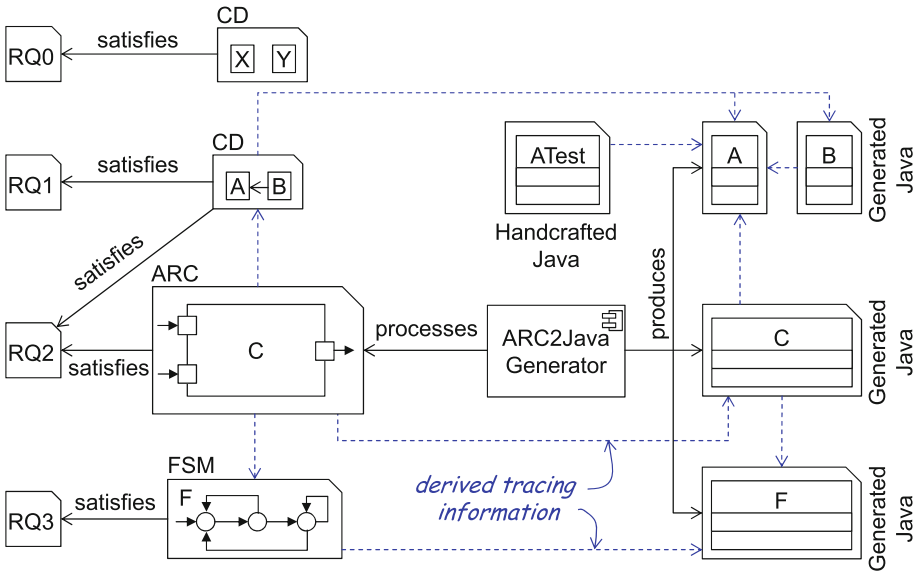


Fig. 2. Overview of explicit and implicit relationships between elements in an MDSE process, where associations colored black are explicitly specified and dashed, blue associations are implicitly defined within the underlying languages of the respective artifact, or derived by taking into account creational dependencies across different stages of a process. (Color figure online)

3 Example

Consider a company developing a software system using MDSE methodology in a multi-stage process. A common challenge in MDSE processes is to trace the impact of changes within an artifact to related artifacts across multiple stages in the process, and to detect implicit dependencies of different artifacts. An excerpt of artifacts used in an exemplary process in context of a single employed tool is depicted in Fig. 2. First, the company specifies requirements (*cf.* RQ0, . . . , RQ3 in Fig. 2) that the functionality of the developed software should satisfy. In the next phase, models that reflect the specified requirements are implemented by modeling experts of the company. Each requirement defines assumptions that are satisfied by one or more modeling artifacts. The modeling artifacts conform to different modeling languages such as, *e.g.*, class diagrams, an architecture description language, and a finite state machine language as depicted in Fig. 2.

The connections between requirements and the modeling artifacts are defined manually. Then, the `ARC2JavaGenerator` tool is employed to transform the given heterogeneous models into Java code. A common difficulty, especially in large software projects, is to understand the exact mapping between each input and output artifact(s) of a tool execution. This mapping is usually encapsulated within the tool. Further, modeling languages typically allow several different

kinds of relationships between models and understanding these requires knowledge about the languages.

With an explicit artifact model as envisioned, the relationships between models and generated Java classes can be derived. Extracting derived information, from various sources, such as, *e.g.*, import statements in models, as well as the relationships between generated artifacts and the input models, greatly supports developers in analyzing dependencies in MDSE projects. For instance, the relations derived in Fig. 2 (denoted by dashed, blue arrows) could facilitate tracing and impact analysis as follows:

- (1) To evaluate, which generated artifacts are needed to satisfy a specific requirements, this information can be derived taking into account the relations between models and requirements, and the relations between a model and the generated artifacts.
- (2) To evaluate, which test case verify which of the requirements, the method in (1) can be used and, in addition, the relations between test and generated code must be regarded.
- (3) To evaluate whether there exist unused model files, which lead to unnecessary complexity for the MDSE project. Those models can be identified by the mapping between input and output artifacts. If there is no mapping for an input artifact, this artifact is a candidate for removal.
- (4) To evaluate whether there exist unused source code files. In our example, the generated Java class B is unused, as there is no derived association to this class from the Java class C generated from the architecture model, which is the excerpt under investigation here. Identifying unused files that do not need to be tested, packaged, or deployed leads to a more efficient MDSE process.
- (5) To evaluate whether the transformation satisfies all requirements, the method in (1) can be used to determine all satisfied requirements. In the example, the class diagram containing the classes X and Y are not referenced from the architecture model, but the class diagram satisfies RQ0 and has no relation to parts of the generated code. This may indicate erroneous models.
- (6) To estimate costs for modifications, tracing can calculate, which artifacts are affected by the modification of a certain artifact. The granularity of the distinct types of relations investigated in an artifact model influences the quality of such a change impact analysis.

4 State of the Art

There are various approaches to support model-driven systems engineering. Popular system engineering tools, such as *PTC Integrity Modeler*¹, *Syndeia*², or *Cameo Systems Modeler*³ support modeling with SysML [8] and, hence, are able

¹ <https://www.ptc.com/en/model-based-systems-engineering/integrity-modeler>.

² <http://intercax.com/products/syndeia/>.

³ <https://www.nomagic.com/products/cameo-systems-modeler>.

to manage different kinds of development artifacts in the same tool. One important feature of those tools is tracing between the managed artifacts. Nevertheless, these tools cannot be used out of the box for our intended purpose: (1) SysML is a general purpose modeling language, *i.e.*, it can be used to describe a large variety of systems. However, as domain-specific modeling advocates tailoring the modeling techniques to the participating domains, using general-purpose modeling languages usually leads to a coarse interpretation of diagrams with respect to the domains. This lack of precision, prevents leveraging the potential of fully automated, integrated model processing. Consequently, additional formalization is need to ensure that the semantics of modeled artifacts and relations between them is unambiguous. This, however, is either not supported by such tools at all or very limited (*cf.* stereotypes). (2) As we are especially interested in the coherences of MDSE processes, there is a need to inspect the artifacts, relations, and processes of MDSE tools themselves. Considering, for instance, highly customizable code generators, it rarely is fully understood which of the artifacts are in use in the context of a given project at a specific time. Especially when parts of MDSE tools are automatically generated themselves, tracing of the overall build process in terms of its artifacts becomes more challenging. This challenge can also not be solved by the mentioned tools, as they usually do not take the development tools into account in such a way, but focus on the development artifacts instead. In MDSE, proper model management is crucial when working with large collections of models. In [3] the notion of megamodels was introduced, which still subject to ongoing research [12, 13]. Under the assumption that everything is model [2] one could argue that the artifact models proposed in this work are megamodels too. As we require formal encoding of models and their relations, there is a difference between megamodels and the proposed artifact models from our viewpoint. The elements of megamodels represent models and the links represent relationships between models [12]. The proposed artifact models focus on the model-driven build process including a whitebox view of the model-driven development tools. For instance, model elements of artifact models can also represent tools, artifacts the tools consists of, generated or handcrafted code files of the target system or configuration files. Links are then relationships between any of those elements. Nevertheless, there are commonalities with megamodels as models and their relationships can also become an important part in artifact models and the corresponding project data.

5 Challenges of Artifact Modeling

There are few approaches towards such an artifact model. The author of [4] focus on the integration of tools and the specification of tool chains and transformations between artifacts. Thus, artifacts managed within different tools are related to each other. The authors of [11] focus on an artifact-oriented way to describe a model-based requirements engineering process. Both approaches consider the requirement and design phases of MDSE projects only, but do not take code generation phases or implementation phases into account. Also, the tools

themselves are not considered in the presented models. The authors of [6], contributed the idea of providing project data to analysts and software tools, but do not combine this idea with an explicit artifact model. Hence, there are still open challenges, which have to be overcome towards efficient and sophisticated artifact modeling.

Methodology. The definition of a methodology on how to create artifact models tailored to the needs of a particular MDSE project. This includes:

- defining the scope of the MDSE project where artifact modeling can help taming the complexity,
- the development and selection of suitable modeling languages, tools and guidelines,
- the creation of model libraries providing reusable concepts common for system engineering projects, and
- development of reusable algorithms based on artifact models providing valuable analysis for common problems of system engineering projects.

Tools. Defining mechanisms, tools, and infrastructure supporting extraction and understanding of artifact data, including

- visualization capabilities, such as those proposed in [7],
- a methodology for integrating the different automated analysis tools to a given infrastructure,
- common interfaces for accessing artifact data, and the
- handling large amounts of artifact data efficiently.

Integration. Overcome modeling challenges, such as

- providing ways of defining and ensuring compliance between related software tools, such as editors, generators, or transformations, and
- integrating process data and historical data into such an artifact model to enable comprehending the state and changes of artifacts and their relations over time.

6 Conclusion

Model-driven development facilitates systems engineering. However, to this end it introduces new challenges, out of which taming the complexity of participating artifacts and their relations is a very important one. We argue that investigating and reifying these using an artifact model and corresponding tooling is crucial to the successful deployment of future systems of systems. The ultimate goal would be, that architects can model their project, including the tools, the MDSE process and the target system's architecture with all relevant relations with minimal effort. The corresponding data should be extracted automatically and enable overviewing of the project's current state. This enables making data-driven decisions regarding tool landscape, processes, and architectures such that

the future MDSE projects can be run successfully. In this paper we presented a small example clarifying the problem. Nevertheless, in complex scenarios with multi-level generation processes and where models of different engineering discipline are related to describe the target product, the benefit of our approach will unfold to full extend. To guide this, particular challenges of artifact modeling future research should investigate were highlighted.

References

1. Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)
2. Bézivin, J.: On the unification power of models. *Softw. Syst. Model.* **4**(2), 171–188 (2005)
3. 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, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2004)
4. Braun, P.: Metamodellbasierte Kopplung von Werkzeugen in der Softwareentwicklung. Logos, Berlin (2004)
5. Butting, A., Greifenberg, T., Rumpe, B., Wortmann, A.: Taming the complexity of model-driven systems engineering projects. In: Cabot, J., Paige, R., Pierantonio, A. (eds.) Part of the Grand Challenges in Modeling (GRAND 2017) Workshop (2017). <http://www.edusymp.org/Grand2017/>
6. Czerwonka, J., Nagappan, N., Schulte, W., Murphy, B.: CODEMINE: building a software development data analytics platform at Microsoft. *IEEE Softw.* **30**(4), 64–71 (2013)
7. Greifenberg, T., Look, M., Rumpe, B.: Visualizing MDD projects. In: Software Engineering Conference (SE 2017). LNI, pp. 101–104. Bonner Köllen Verlag (2017)
8. Object Management Group: OMG Systems Modeling Language (OMG SysML), May 2017. <http://www.omg.org/spec/SysML/1.5/>
9. Jain, S., Lechevalier, D.: Standards based generation of a virtual factory model. In: Proceedings of the 2016 Winter Simulation Conference, pp. 2762–2773. IEEE Press (2016)
10. Khan, A., Turowski, K.: A survey of current challenges in manufacturing industry and preparation for industry 4.0. In: Abraham, A., Kovalev, S., Tarassov, V., Snaštel, V. (eds.) Proceedings of the First International Scientific Conference “Intelligent Information Technologies for Industry” (IITI’ 16). AISC, vol. 450, pp. 15–26. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33609-1_2
11. Méndez Fernández, D., Penzenstadler, B.: Artefact-based requirements engineering: the AMDiRE approach. *Requir. Eng.* **20**(4), 405–434 (2015)
12. Salay, R., Kokaly, S., Chechik, M., Maibaum, T.: Heterogeneous megamodel slicing for model evolution. In: ME@ MODELS, pp. 50–59 (2016)
13. Simmonds, J., Perovich, D., Bastarrica, M.C., Silvestre, L.: A megamodel for software process line modeling and evolution. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 406–415. IEEE (2015)

Cognifying Model-Driven Software Engineering

Jordi Cabot^{1,2}(✉), Robert Clarisó², Marco Brambilla³, and Sébastien Gérard⁴

¹ ICREA, Barcelona, Spain

`jordi.cabot@icrea.cat`

² Universitat Oberta de Catalunya, Barcelona, Spain

`rclariso@uoc.edu`

³ Politecnico di Milano, Milan, Italy

`marco.brambilla@polimi.it`

⁴ CEA List, Palaiseau, France

`sebastien.gerard@cea.fr`

Abstract. The limited adoption of Model-Driven Software Engineering (MDSE) is due to a variety of social and technical factors, which can be summarized in one: its (real or perceived) benefits do not outweigh its costs. In this vision paper we argue that the *cognification* of MDSE has the potential to reverse this situation. Cognification is the application of knowledge (inferred from large volumes of information, artificial intelligence or collective intelligence) to boost the performance and impact of a process. We discuss the opportunities and challenges of cognifying MDSE tasks and we describe some potential scenarios where cognification can bring quantifiable and perceivable advantages. And conversely, we also discuss how MDSE techniques themselves can help in the improvement of AI, Machine learning, bot generation and other cognification techniques.

Keywords: Model · Machine learning · Bot · Model-driven · AI

1 Introduction

It is hard to imagine anything that would “change everything” as much as cheap, powerful, ubiquitous intelligence and exploitation of knowledge. Even a very tiny amount of useful intelligence embedded into an existing process boosts its effectiveness to a whole other level [18]. This process is known as *cognification*.

Cognification can be defined as the application of knowledge to boost the performance and impact of a process. It does not restrict itself to what we typically refer to as artificial intelligence, with Deep Learning as its latest and hottest technique. Cognification includes as well the combination of past and current human intelligence (i.e. all current humans online and the actions they do). Under this definition, collective intelligence and crowdsourcing [30] are valid cognification tools. Cognification does not imply either the existence of a super AI but the availability of highly specialized intelligences that can be plugged in depending on the needs of the problem to be solved.

Several initiatives aim to cognify specific tasks in Software Engineering, for instance, using machine learning (ML) for requirements prioritization [26], for estimating the development effort of a software system [31] or the productivity of individual practitioners [22] or to predict defect-prone software components [27]. This trend is also happening at the tool level, e.g., the recent Kite IDE¹ claims to “augment your coding environment with all the web’s programming knowledge”. MDSE should join this ongoing trend.

Moreover, we know the limited adoption of MDSE is due to a variety of social and technical factors [14] but we can summarize them all in one: its benefits do not outweigh its costs. We believe cognification could drastically improve the benefits and reduce the costs of adopting MDSE.

In this paper we discuss the opportunities that cognification can bring to the MDSE field, in terms of possible tools, process steps, and usage scenarios and their empowerment through cognification. We also discuss how MDSE itself can be applied to AI and knowledge-aware technology development, and we conclude with some challenges and risks that this roadmap may encompass.

2 Opportunities in the Cognification of MDSE

All MDSE tasks and tools are good candidates to be cognified, since they typically involve plenty of knowledge-intensive activities. Here, we comment on a few examples where the performance or quality of experience can be especially boosted thanks to cognification.

2.1 Modeling Bots

Cognification can enable **modeling bots** playing the role of virtual modeling assistant. Based on previous models on the same domain available online or on a comparison between your current model and general ontologies, the bot could suggest missing properties on the model, recommend best practices or warn that some model aspects differ from the way they are typically modeled.

Preliminary work in this direction considers recommendations during the construction of a model to reduce the effort made by designers. [29] proposes a recommender that suggests which model element should be the target of a newly created reference. Similarly, [21] studies the notion of an “*auto-complete*” that is able to infer the intended pattern that is being designed.

In some contexts, models are created by domain experts by using a Domain-Specific Language (DSL). This creation process includes manual activities like auto-completing a partial model or introducing small fixes to ensure it satisfies the integrity constraints of the domain. *Proactive modeling* [24] aims to automate these manual activities. Bots could even interact with domain experts in a more friendly environment (e.g. social networks, [25] would be a first attempt for modeling in twitter) to facilitate the knowledge acquisition process.

¹ <https://kite.com/>.

2.2 Model Inferencers

A **model inferencer** is able to deduce a common schema behind a set of unstructured data (logs, tweets, forum messages,...). This model would be a useful “lense” to interpret the data. A good example of a model inferences is JSON Discoverer [17]. Given a (set of) JSON documents, it analyzes the JSON data and generates for you a class diagram showing graphically the implicit JSON schema of your documents plus an object diagram representing their data.

In a related context, *process mining* [1] aims to achieve a similar goal with respect to procedural information describing a process, action or activity.

2.3 Smart Code Generators

A cognified **code generator** would be able to learn the style and best practices of a particular organization and mimic them in the code it outputs. By learning from good code samples, the generator would be able to imitate the company’s best practices and style and maximize its chances to be accepted as a new “developer” for the company.

This line of thought is close to existing works on example-based generation of modeling artefacts. For instance, [19], where a search-based approach is used to derive model-to-model transformations from sample input and output models.

2.4 Real-Time Model Reviewers

Machine learning has already been used in the context of verification and validation to tune verification algorithms by selecting the best choice for heuristics [12, 15]. However, here the aim would be constructing a **real-time model reviewer** able to give continuous feedback on the quality of the model from the point of view of consistency and correctness.

Such tool would resort to different verification and validation (V&V) techniques, using information about previous analysis to predict the complexity of the analysis [16] and select the most suitable method to verify a particular model. This problem is known in the A.I. field as the *algorithm selection problem* [20].

In addition to selecting the most adequate tool, previous experiences can be useful by highlighting the most frequent faults or the constructs that are most likely to produce problems. This type of information can be used to guide the search process, e.g. which kind of properties should be checked for a particular model. Some preliminary works addressed the use of semantic reasoning for validation and property checking for models: for instance [9], propose a simple tool able to address issues that cannot be identified by traditional type checking tools.

Another approach to model quality would be to *evolve* or *adapt* models rather than detect and correct faults. For instance, the notion of *liquid models* [23], which considers that design-time models should not be immutable artifacts, but that they should evolve to take into account potential deviations occurring at run-time. Cognification could be used as a means to study and explore candidate models during this evolution process.

2.5 Advanced Self-morphing and Collaborative Modeling Tools

Automatic learning approaches can also be used within **self-morphing modeling tool**, able to adapt its interface, functions, behaviour (and even the expressiveness of the language offered) to the expertise of the tool user and to the context of the modeling problem addressed. This challenge is related to the problem of plastic user interfaces, i.e. interfaces able to keep its usability under changing uses and circumstances (responsive websites can be regarded as a limited example of a plastic UI). While there has been some work on using MDE to generate plastic UIs [28], building a plastic MDE IDE remains an open challenge.

The tool should not just adapt to different user profiles but also effectively support the collaboration of those users. Indeed, cognification also comprises exploitation of human and collective intelligence. As an example, we could leverage crowdsourcing techniques [6] to devise the best domain-specific modeling language to be used by the tool in order to optimize the communication process between modeling experts and domain experts. This has been so far applied to the problem of agreeing on the concrete syntax notation of domain-specific languages, as described in [4].

2.6 Semantic Reasoning Platforms, Explainability and Storification

At the purpose of making models as self-explanatory as possible, semantics-based techniques can be applied to the concepts in the model to enable automatic explanation of models. This would require a **semantic reasoning platform** able to map modeled concepts to existing ontologies and provide definitions (similar to Kindle WordWise²), references, relations to relevant concepts, services and contextual information; and conversely also generate new knowledge through inference.

This would also enable automatic generation of thesaurus, data dictionaries, and even explanatory text out of models. Integrating MDE with resources such as ontologies, semantic processors, NLP tools, rule-based inference engines, and alike will be crucial in this setting.

2.7 Scalable Model and Data Fusion Engine

Big data architectures and data streaming platform enable the construction of **data fusion engines** that are able to perform semantic integration and impact analysis of design-time models with runtime data such as software usage logs, user profiles and so on. A typical use case is the integration of Web application logs, which are widespread in Web servers, with user interaction models of Web applications. An example is the work [2,3] which integrates with a scalable approach the real-time big data stream coming from large-scale Web sites with IFML models [7]. This close interaction between design and runtime models is also the focus of the MegaMart2 EU project³.

² <https://www.amazon.com/gp/feature.html?ie=UTF8&docId=1002989731>.

³ <https://megamart2-ecsel.eu/>.

3 Model-Driven Engineering of AI and Knowledge-Aware Software

Besides the advantages that cognification can bring to MDSE, the reverse is also true: MDSE techniques can be used to improve knowledge-aware technologies. As in any other domain, the use of models (and its abstraction power) can help simplify, interoperate and unify a fragmented technological space, as it is the case right now in AI, with plenty of competing and partially overlapping libraries and components for all kinds of knowledge processing tasks.

So far, this line of work remains largely unexplored. A few exceptions are [13], aiming to integrate machine learning results in domain modeling, [10], aligning MDE and ontologies, and works oriented to model-driven development of semantic Web based applications [8] and semantic web services [5].

4 Conclusions and Challenges

As we described in this paper, a lot of possible scenarios and tools in MDSE can benefit from the application of cognification techniques in broad sense. Even if only a few of them become a reality in the short-term, they have the potential to drastically change the way MDSE is used and perceived in the software community.

Nevertheless, some risks need to be addressed. The main challenge is that most of the above scenarios imply the availability of lots of training data in order to get high-quality learning results. In MDSE, this means a *curated catalog* of good and bad models, meta-models, transformations, code, and so on [11]. While some model repositories are available (e.g. REMODD⁴ or MDEForge⁵) the number and diversity of the modeling artefacts they contain is still limited. Improving this situation is a strong requirement towards the cognification of MDE.

References

1. van der Aalst, W.: Process mining manifesto. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM 2011. LNBP, vol. 99, pp. 169–194. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28108-2_19
2. Bernaschina, C., Brambilla, M., Koka, T., Mauri, A., Umuhoza, E.: Integrating modeling languages and web logs for enhanced user behavior analytics. In: WWW 2017, pp. 171–175 (2017)
3. Bernaschina, C., Brambilla, M., Mauri, A., Umuhoza, E.: A big data analysis framework for model-based web user behavior analytics. In: Cabot, J., De Virgilio, R., Torlone, R. (eds.) ICWE 2017. LNCS, vol. 10360, pp. 98–114. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60131-1_6

⁴ <http://www.remodd.org/>.

⁵ <http://www.mdeforge.org/>.

4. Brambilla, M., Cabot, J., Izquierdo, J.L.C., Mauri, A.: Better call the crowd: using crowdsourcing to shape the notation of domain-specific languages. In: Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017). ACM, New York (2017)
5. Brambilla, M., Ceri, S., Celino, I., Cerizza, D., Valle, E.D., Facca, F.M., Turati, A., Tziviskou, C.: Experiences in the design of semantic services using web engineering methods and tools. *J. Data Semant. (JoDS)* **11**, 1–31 (2008)
6. Brambilla, M., Ceri, S., Della Valle, E., Volonterio, R., Acero Salazar, F.: Extracting emerging knowledge from social media. In: WWW 2017, pp. 795–804 (2017)
7. Brambilla, M., Fraternali, P.: Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML. Morgan Kaufmann, Burlington (2014)
8. Brambilla, M., Tziviskou, C.: Modeling ontology-driven personalization of web contents. In: 8th International Conference on Web Engineering, ICWE 2008, New York, USA, pp. 247–260 (2008)
9. Brambilla, M., Tziviskou, C.: An online platform for semantic validation of UML models. In: Gaedke, M., Grossniklaus, M., Díaz, O. (eds.) ICWE 2009. LNCS, vol. 5648, pp. 477–480. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02818-2_42
10. Gasevic, D., Djuric, D., Devedzic, V.: Model Driven Engineering and Ontology Development, 2nd edn. Springer, Heidelberg (2009). <https://doi.org/10.1007/978-3-642-00282-3>
11. Gogolla, M., Cabot, J.: Continuing a benchmark for UML and OCL design and analysis tools. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 289–302. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_22
12. Haim, S., Walsh, T.: Restart strategy selection using machine learning techniques. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 312–325. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_30
13. Hartmann, T., Moawad, A., Fouquet, F., Le Traon, Y.: The next evolution of MDE: a seamless integration of machine learning into domain modeling. *Softw. Syst. Model.* (2017)
14. Hutchinson, J.E., Whittle, J., Rouncefield, M.: Model-driven engineering practices in industry: social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Program.* **89**, 144–161 (2014). <https://doi.org/10.1016/j.scico.2013.03.017>
15. Hutter, F., Babic, D., Hoos, H.H., Hu, A.J.: Boosting verification by automatic tuning of decision procedures. In: Proceedings of the Formal Methods in Computer Aided Design, FMCAD 2007, pp. 27–34. IEEE Computer Society, Washington, DC (2007)
16. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K.: Algorithm runtime prediction: methods & evaluation. *Artif. Intell.* **206**, 79–111 (2014)
17. Izquierdo, J.L.C., Cabot, J.: JSONDiscoverer: visualizing the schema lurking behind JSON documents. *Knowl.-Based Syst.* **103**, 52–55 (2016). <https://doi.org/10.1016/j.knosys.2016.03.020>
18. Kelly, K.: The Inevitable: Understanding the 12 Technological Forces that Will Shape our Future. Viking, New York (2016)
19. Kessentini, M., Sahraoui, H.A., Boukadoum, M., Benomar, O.: Search-based model transformation by example. *Softw. Syst. Model.* **11**(2), 209–226 (2012). <https://doi.org/10.1007/s10270-010-0175-7>

20. Kotthoff, L., Gent, I.P., Miguel, I.: An evaluation of machine learning in algorithm selection for search problems. *AI Commun.* **25**(3), 257–270 (2012)
21. Kuschke, T., Mäder, P., Rempel, P.: Recommending auto-completions for software modeling activities. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) *MODELS 2013*. LNCS, vol. 8107, pp. 170–186. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_11
22. Lopez-Martin, C., Chavoya, A., Meda-Campaa, M.E.: A machine learning technique for predicting the productivity of practitioners from individually developed software projects. In: *SPND 2014*, pp. 1–6 (2014)
23. Mazak, A., Wimmer, M.: Towards liquid models: an evolutionary modeling approach. In: *2016 IEEE 18th Conference on Business Informatics (CBI)*, vol. 1, pp. 104–112. IEEE (2016)
24. Pati, T., Feiock, D.C., Hill, J.H.: Proactive modeling: auto-generating models from their semantics and constraints. In: *Proceedings of the 2012 Workshop on Domain-Specific Modeling, DSM 2012*, pp. 7–12. ACM, New York (2012)
25. Pérez-Soler, S., Guerra, E., de Lara, J., Jurado, F.: The rise of the (modelling) bots: towards assisted modelling via social networks. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, 30 October–03 November 2017*, pp. 723–728 (2017). <http://dl.acm.org/citation.cfm?id=3155652>
26. Perini, A., Susi, A., Avesani, P.: A machine learning approach to software requirements prioritization. *IEEE Trans. Softw. Eng.* **39**(4), 445–461 (2013)
27. Shepperd, M., Bowes, D., Hall, T.: Researcher bias: the use of machine learning in software defect prediction. *IEEE Trans. Softw. Eng.* **40**(6), 603–616 (2014)
28. Sottet, J.-S., Ganneau, V., Calvary, G., Coutaz, J., Demeure, A., Favre, J.-M., Demumieux, R.: Model-driven adaptation for plastic user interfaces. In: Baranauskas, C., Palanque, P., Abascal, J., Barbosa, S.D.J. (eds.) *INTERACT 2007*. LNCS, vol. 4662, pp. 397–410. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74796-3_38
29. Steimann, F., Ulke, B.: Generic model assist. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) *MODELS 2013*. LNCS, vol. 8107, pp. 18–34. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_2
30. Stol, K.J., Fitzgerald, B.: Two’s company, three’s a crowd: a case study of crowd-sourcing software development. In: *ICSE 2014*, pp. 187–198. ACM (2014)
31. Wen, J., Li, S., Lin, Z., Hu, Y., Huang, C.: Systematic literature review of machine learning based software development effort estimation models. *Inf. Softw. Technol.* **54**(1), 41–59 (2012)

Non-human Modelers: Challenges and Roadmap for Reusable Self-explanation

Antonio Garcia-Dominguez^(✉) and Nelly Bencomo

School of Engineering and Applied Science, Aston University, Birmingham, UK
{a.garcia-dominguez,n.bencomo}@aston.ac.uk

Abstract. Increasingly, software acts as a “non-human modeler” (NHM), managing a model according to high-level goals rather than a predefined script. To foster adoption, we argue that we should treat these NHMs as members of the development team. In our GrandMDE talk, we discussed the importance of three areas: effective communication (self-explanation and problem-oriented configuration), selection, and process integration. In this extended version of the talk, we will expand on the self-explanation area, describing its background in more depth and outlining a research roadmap based on a basic case study.

1 Introduction

There is increased interest in tools managing models by themselves according to goals, rather than following a predefined script. A case study for the 2016 Transformation Tool Contest [8] on the class responsibility assignment problem showed how traditional model-to-model languages had to be orchestrated with higher-level components that guided rule applications (e.g. genetic algorithms, simulated annealing, reachability graphs or greedy application of a heuristic). Two solutions were based on reusable model optimisation frameworks (MOMoT, MDEOptimiser). These tools were evaluating options in a model much like a human would: these could be considered as “non-human modellers” (NHMs).

Self-adaptive systems based on the models@run.time approach are another good example of tools that manage models on their own. The models@run.time approach advocates using models to provide a formal description of how the system should behave. In a self-adaptive system, there is a feedback loop that guides the evolution of the model based on its current performance and the environment. Some examples of self-adaptive models@run.time include smart grid outage management [4] or ambient assisted living [9].

These NHMs are software entities which participate in a modeling team. As team members, we need to be able to influence their behaviour and understand why they took specific decisions within the models: this may be particularly complex when talking to domain experts rather than MDE specialistics. The NHMs also need to find their place in our processes and in the lifetime of the system that is being developed: they could be used once early in development, invoked once per development sprint, or brought in as another member of a

(possibly live) collaborative modeling tool. These challenges point to several interesting lines of research, which were raised at the GrandMDE workshop.

This extended version of the original proposal will focus on the discussion developed during the GrandMDE workshop around the self-explanation area, in which strong links with existing ideas from traceability and provenance were identified. After introducing this expanded background, a motivational case study will introduce a general roadmap for our envisioned approach. This roadmap will be grounded around existing standards and industrial-strength tools where possible.

2 Discussed Topics

Based on the previous discussion, these are some of the lines of work that we considered relevant to integrating non-human modellers (NHMs) as team members:

- Accessible configuration: existing tools have wildly different approaches to fine tune their behaviour, and domain experts find it increasingly harder to figure out which knobs to turn. It should be possible to abstract over specific approaches and provide users with a problem-centered description of any parameters, much as SPEM describes software processes abstractly.
- Self-explanation: the approaches currently available for this capability in self-adaptive systems are *ad hoc* and costly to develop, making them very rare in practice. There is no common approach for model optimisation either. This line of work would start by allowing changes to a model to be annotated with “why” a change was made: which reasoning process was followed, what inputs were used, and which other alternatives were evaluated. This would be followed up with producing accessible descriptions of this stored information.
- Selection: the community would benefit from building a coherent toolbox of options for NHMs and guiding practitioners on how to pick the right one for a particular problem, much like how the Weka tool brought together many data mining approaches into a common framework¹.
- Process integration: depending on the task, the NHM will need to be integrated into the day-to-day work of the team. Beyond one-off uses, NHMs could be part of a continuous integration loop (reacting to commits on a model), or participate in a shared modelling environment (perhaps with the ability to chat with users about observed issues).

Among these topics in our talk at the GrandMDE’17 workshop, self-explanation attracted most of the questions afterwards: attendees requested a concrete motivational example, and there were several discussions on how this challenge tied to existing work in the areas of traceability and provenance. The rest of this work will focus on answering those questions and setting out an initial roadmap for advancing the state of the art in self-explanation of NHMs.

¹ <http://www.cs.waikato.ac.nz/ml/weka/>.

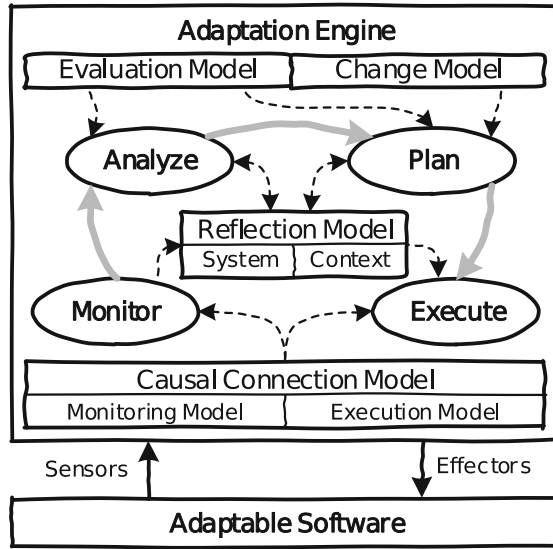


Fig. 1. MAPE feedback loop using different runtime model types [19]

3 Background for Reusable Self-explanation

Adaptive software relies on feedback loops between *sensors* that bring information from the environment, and *effectors* that change it. When it is model-based (i.e. following a *models@run.time* approach), these sensors and effectors will be exposed through *runtime models*. Giese et al. [10] defines runtime models as models that:

- Are designed to be used during the runtime of a system,
- are encoded to enable its processing,
- are casually connected to the system (changes in the model propagate to the system and/or the other way around).

Wätzoldt and Giese [19] presented a useful categorisation of runtime models around the MAPE (Monitor, Analyze, Plan, and Execute) loop [13], shown in Fig. 1. Within this loop, we can observe the following types of models within the adaptation engine:

- *Reflection models* reflect concerns about the running system (*system models*) and its context (*context models*).
- *Evaluation models* contain the specification of the system (*requirement models*) and any assumptions made (*assumption models*). These define the solution space of the system.
- *Change models* describe specific solutions for the space defined by the evaluation models. *Variability models* explicitly describe the available options (much like a feature model), while *modification models* only indicate what changes would be made on the reflection model.

- *Causal connection models* link the system to the reflection models. *Monitoring models* indicate how the reflection models should be updated from the system and the context, whereas *execution models* specify how to operate the system based on the information in those reflection models.

In this case, typically the evaluation and causal connection models would be produced by the developers of the adaptive system, and the reflection and change models would be managed by the adaptation engine itself. Here, the adaptation engine would be the NHM.

Self-explanation would therefore entail telling users why the adaptation engine made a specific change in those reflection and change models. This change could come from multiple reasons: changed requirements or assumptions in the evaluation models, different solution alternatives in the change models, or new data coming through the monitoring model. Alternatively, a user could simply want to see a snapshot of the reflection models at a certain moment, and request the reason why a specific value or entity was present.

Solving this problem requires combining ideas from multiple areas. So far, we have identified three: traceability, model versioning and provenance. The following sections will provide more background on each of these topics and how they relate to self-explanation for these adaptation engines.

3.1 Traceability

Linking system requirements to its various design and implementation artifacts has been a concern for a long time. Gotel and Finkelstein defined requirements traceability back in 1994 [11] as “the ability to describe and follow the life of a requirement, in both a forwards and backwards direction”. Much work has been done since then in terms of defining guidelines for traceability, creating and maintaining trace links, and querying those trace links after the fact [5].

It is hard to create and maintain traceability links between manually created artefacts. However, the automation brought by MDE has made it possible to generate links between models in most of the popular model-to-model transformation languages (e.g. ATL [12] or ETL [15]). These traceability links are typically quite simple, with references to the various ends of the link (source and target elements) and the transformation rule that was applied, as mentioned by Matragkas et al. [16].

On the one hand, self-adaptive systems are automated just like model-to-model transformation engines, and therefore it should be possible to include trace creation into their processes. On the other hand, self-adaptive systems need to operate in an uncertain environment - their decisions usually require more complex reasoning and are dependent on the specific context at the time. Generic source-target links would be unable to capture this information: a more advanced information model is required. It appears that a richer case-specific traceability metamodel would be ideal for this situation, as recommended by Paige et al. [18].

3.2 Model Versioning

Whereas traceability is about following the life of a requirement or seeing where a piece of code really come from, versioning is about keeping track of how a specific artifact evolved over time. Version control systems (VCSs) have been a staple of software engineering for a very long time, and current state-of-the-art systems such as Git² make it possible to have developers collaborate across the globe.

Models also evolve over time, and it is possible to reuse a standard text-based VCS for it. However, text-based VCSs do not provide explicit support for comparing models across versions or merging versions developed in parallel by different developers. This has motivated the creation of specific version control systems for models: *model repositories*. EMFStore [14] and CDO³ are mature examples. EMFStore is file-based and tracks modifications across versions of a model, grouping them into *change packages* which can be annotated with a commit message. CDO implements a pluggable storage solution (by default, an embedded H2 relational database) and provides support for branching and merging. Current efforts are focused on creating repositories that scale better with the number of collaborators (e.g. with better model merging using design space exploration [6]), or with the size of the models (e.g. Morsa [17]).

Self-explanation needs to keep track of the history of the various runtime models, and could benefit from these model repository. However, current systems only keep unstructured descriptions (in plain text) of each revision that a model goes through. It would be very hard to achieve self-explanation from these commit messages: we would rather have *commit models* that are machine readable. These commit models would have to relate the old and the new versions with external models, perhaps under their own versioning schemes. Supporting these cross-version relationships may require a good deal of research and technical work as well.

3.3 Provenance

Buneman et al. defined data provenance (also known as “lineage” or “pedigree”) as the description of the origins of a piece of data and the process by which it arrived at a database [3]. This was further divided into the “why provenance” (table rows from which a certain result row was produced) and the “where provenance” (table cells from which a certain result cell was produced).

Since then, provenance has been slowly extended to cover more and more types of information systems, and has taken special importance with the advent of “big data”. Commercial vendors such as Pentaho now include data lineage capabilities in their own Extract-Transform-Load tools⁴.

Various efforts have been made to standardise the exchange of provenance information. In 2013, the Provenance Working Group of the World Wide Web

² <http://git-scm.com>.

³ <http://projects.eclipse.org/projects/modeling.emf.cdo>.

⁴ https://help.pentaho.com/Documentation/6.0/OL0/0Y0/Data_Lineage.

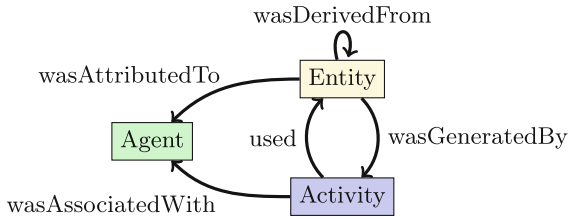


Fig. 2. High-level overview of the PROV data model [1]

Consortium (W3C) produced the PROV family of documents, which “defines a model, corresponding serializations and other supporting definitions to enable the inter-operable interchange of provenance information in heterogeneous environments such as the Web”. Provenance is further generalised to knowing the origins of any digital object: provenance records indicate how entities were generated by activities undertaken by various agents [1]. This high level view of the PROV data model is shown on Fig. 2. PROV includes further provisions for specifying *roles* taken by agents and entities in an activity, and how activities may follow *plans* across time.

This view of provenance can be seen as a richer, more detailed view of traceability that not only requires following artefacts that are produced from one another, but also tracking carefully what was done and by whom. In fact, the PROV data model could be used as a starting point for the formal notation of the machine-readable “commit messages” suggested in Sect. 3.2.

4 Example Scenario

After introducing the various topics related to self-explanation for NHMs, this section will propose a concrete scenario where a reusable infrastructure for self-explanation of models@run.time approaches would be useful. It is from the service monitoring and management domain. Specifically, it is about the use of self-adaptation for the management of clusters of Hawk servers.

4.1 Scenario Description

Hawk is a heterogeneous model indexing framework [2], originally developed as part of the MONDO project. It monitors a set of locations containing collections of file-based models, and mirrors them into graph databases for faster and more efficient querying. Hawk can be deployed as an Eclipse plugin, a Java library, or a standalone server. The server version of Hawk exposes its capabilities through a web service API implemented through Apache Thrift. Prior studies have evaluated how a single Hawk server can scale with an increasing number of clients [7], with competitive or better results than other alternatives (CDO and Mogwai).

Despite these positive results, Hawk servers still have an important limitation: at the moment, there is no support for aggregating multiple servers into

a cluster with higher availability and higher scalability. Hawk can use the high-availability configurations of some backends (e.g. the OrientDB multi-master mode), but this will not improve the availability of the Hawk API itself, which will still act as a single point of failure.

The only way to solve this is to have entire Hawk servers cooperate with others: they should distribute their load evenly and monitor the availability of their peers. Periodic re-indexing tasks should also be coordinated to ensure that at least one server in the cluster will always remain available for querying, and out-of-date servers should be forced to update before becoming available again.

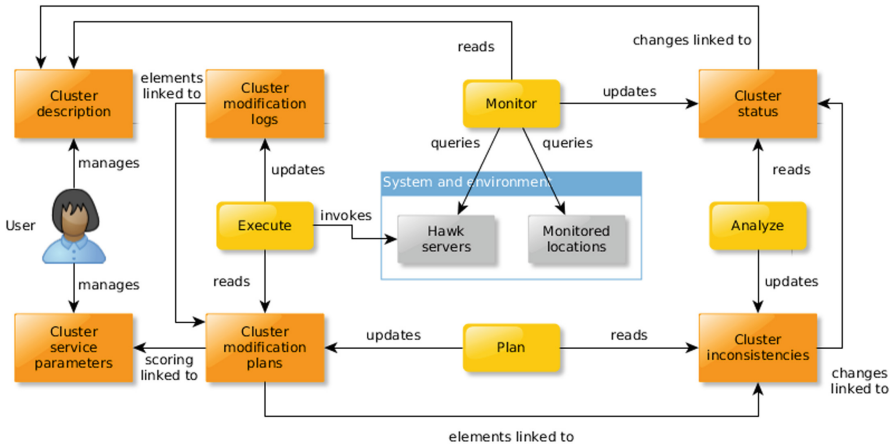


Fig. 3. Case study: self-adaptive Hawk clusters

This results in a self-adaptive system like the one shown in Fig. 3, where the adaptation layers wraps over the collection of Hawk servers (the “system”) and the monitored model storage locations (the “context”):

- The “User” of the cluster manages two models: the *description model* of the intended contents of the cluster (locations to be indexed, and servers to be managed), and the *non-functional service parameters model* of the cluster (desired tradeoff between availability and freshness).
- The “Monitor” step of the MAPE loop uses the description as a monitoring model and queries the various servers and locations. The obtained information is used to update the *cluster status model*, a reflection model where each change is annotated with metadata (e.g. query timestamp, response time, observed errors) and linked back to the element of the cluster description model that caused it.
- The “Analyze” step takes the cluster status model and revises the *cluster inconsistencies* model, removing previously observed inconsistencies that no longer hold (linking them back to the evidence) and adding new inconsistencies (again, linked to the evidence) where servers are outdated or unavailable.

The cluster inconsistencies model would be a change model, as it reflects the problem to be solved in the “Plan” step.

- The “Plan” step produces a set of *cluster modification plan models* with different alternatives to solve the inconsistencies. These alternatives are different sequences of invocations of the Hawk management API. Their elements may solve (partially or fully) certain inconsistencies, and each plan will need to be scored according to the cluster service parameters. These could be considered as execution models.
- Finally, the “Execute” step will run the highest scoring modification plan and record the results of those invocations in a *modification log model* (yet another reflection model). Each log entry needs to be justified from the original elements of the selected modification plan.

4.2 Approach for Reusable Self-explanation

Each of these MAPE steps is acting as its own NHM, taking in the latest versions of independently evolving models and combining them with external information to drive the evolution of its own runtime models. Allowing users to develop trust by gaining an understanding of the workings of the adaptation engine is important for the wider adoption of Hawk clustering. It may seem that tracking all these adaptations would require purpose-specific models and infrastructure, but on closer examination, there are already many elements in the state of the art that can be reused.

The PROV Data Model is a promising start for a reusable metamodel for self-explanation of changes in runtime models (Sect. 3.3). However, there are multiple ways in which PROV could be used:

1. PROV explains each element in each version of the runtime models. The activities are the MAPE steps, the agent is the adaptation engine, and the entities are the source pieces of evidence for the element. It is not exactly clear how would the reasoning for a particular element being there would be linked.
2. PROV explains each change applied to each version of the runtime models. The activities are the changes themselves, the agent is the MAPE step, and the entities are the source pieces of evidence for the element, as well as the reasoning for that change (or group of changes). Our initial estimation is that this approach would provide more fine-grained information for later self-explanation.

Regardless of our selection, PROV lacks specific provisions for model management activities, or pointing to certain model elements within a specific version of a model. These would need to be defined and developed.

Tracking the history of the models themselves can already be done through the model repositories mentioned in Sect. 3.2, but it is unclear how to store the above PROV descriptions and link them back to the various revisions of the models. The PROV descriptions could be kept as complementary models in the

repository, or they could be kept as *commit models* which replace the usual textual commit messages. Using complementary models could be easier to reuse across model repositories, but some conventions would still be needed to link the tracked model to the PROV information. Using commit models could be more natural if the model repository already tracked model changes in its commits (like EMFStore): links could be directly established to those elements. This is another area that merits further research.

Finally, once the PROV records have been created and stored, the next part would be presenting them in a manner that is approachable but still reusable across applications. Conceptually, what we want is (i) being able to track the state of the MAPE loop at a certain point of time, and more importantly (ii) answering how a model element or value in it came to be. While (i) should be readily available through sensible use of model versioning, we are not aware of a reusable interface for querying the information in (ii). Conceptually, it would be a more powerful version of the capability of most version control systems to know when was a certain line of text touched, and by whom.

5 Research Roadmap

Summarising the discussion from the previous section, we can establish an initial roadmap for the creation of a first prototype of the envisioned reusable self-explanation framework for self-adaptive systems following the MAPE loop:

1. First of all, the creation of *basic self-adaptive prototypes* that achieve the intended functionality except for the self-explanation capabilities.
Deriving the self-explanation framework from working software (bottom up) should provide a more realistic implementation, and will give us more experience in the implementation of self-adaptive systems according to the models@run.time paradigm. Hawk will benefit earlier from the horizontal scalability as well.
2. Next, the prototype would be extended with *simulation capabilities* for the system and environment, in order to create new situations for testing adaptability more quickly. This would make it possible to see exactly how the system adapted to a predefined situation, and check if the self-explanation capabilities meet our expectations.
Ideally, because of the models@run.time approach, this should be a matter of mocking the answers from the system - the rest of the approach should remain as is.
3. Being able to run the self-adaptive system in real and simulated environments, the next part of the work would be *comparing* the two approaches to extending PROV that were observed. Initially, it would be a matter of recording the information in both ways (element-first or change-first) and comparing their level of detail, and the relative ease of capture and querying.
4. This would be followed by the *integration* of the PROV records into the history of the runtime models, whether as side-by-side models or as the previously mentioned *commit models*. The comparison would also need to take into

account practical details such as scalability over time and model complexity, ease of use and reusability across different model repositories.

5. Finally, the *visualisation* of the PROV records could be treated in multiple ways. The envisioned goal is for a side-by-side view of the selected model element and its history, where the user may be able to pull additional PROV-encoded information on demand, at least over a single iteration of the full MAPE loop. The links identified between the models in Fig. 3 would enable this, unless proven otherwise during the design of the PROV extensions.

6 Conclusion

This paper started from a general proposal for the thorough consideration of goal- or requirement-based non-human entities managing models (the so-called “non-human modellers”) as additional members of a modelling team that we must talk to, understand, pick and integrate into our processes. The “reusable self-explanation” part took the most questions during the event, and for that reason we expanded on some of the background behind these ideas and described a scenario from the service monitoring domain in which it would be useful.

The discussion has touched upon the fact that most of the ingredients already exist: traceability and provenance have been around for a long time, and model versioning is a common practice in industrial MDE environments, with mature purpose-specific software to do it. However, our understanding is that their specific combination for reusable self-explanation has yet to be achieved, and for that purpose we have set out a bottom-up roadmap which starts with the development of a testbed and continues with the extension, storage and reusable visualisation of a dialect of the PROV specification.

References

1. PROV Model Primer. W3C Working Group Note, World Wide Web Consortium, April 2013. <https://www.w3.org/TR/2013/NOTE-prov-primer-20130430/#intuitive-overview-of-prov>
2. Barmpis, K., Kolovos, D.: Hawk: towards a scalable model indexing architecture. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary. ACM (2013). <http://dl.acm.org/citation.cfm?id=2487771>
3. Buneman, P., Khanna, S., Tan, W.-C.: Why and where: a characterization of data provenance. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 316–330. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44503-X_20. <http://dl.acm.org/citation.cfm?id=645504.656274>
4. Burger, E., Mittelbach, V., Koziolok, A.: View-based and model-driven outage management for the smart grid. In: Proceedings of the 11th International Workshop on Models@run.time, Saint Malo, France, pp. 1–8. CEUR-WS.org, October 2016. http://ceur-ws.org/Vol-1742/MRT16_paper_1.pdf
5. Cleland-Huang, J., Gotel, O.C.Z., Huffman Hayes, J., Mäder, P., Zisman, A.: Software traceability: trends and future directions. In: Proceedings of the on Future of Software Engineering, FOSE 2014, pp. 55–69. ACM, New York (2014). <http://doi.acm.org/10.1145/2593882.2593891>

6. Debreceeni, C., Ráth, I., Varró, D., De Carlos, X., Mendiáldua, X., Trujillo, S.: Automated model merge by design space exploration. In: Stevens, P., Wasowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 104–121. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_7
7. Garcia-Dominguez, A., Barmpis, K., Kolovos, D.S., Wei, R., Paige, R.F.: Stress-testing remote model querying APIs for relational and graph-based stores. *Softw. Syst. Model.* 1–29 (2017). <https://link.springer.com/article/10.1007/s10270-017-0606-9>
8. Garcia-Dominguez, A., Krikava, F., Rose, L.M. (eds.): Proceedings of the 9th Transformation Tool Contest, CEUR Workshop Proceedings, vol. 1758, December 2016. <http://ceur-ws.org/Vol-1758/>. ISSN 1613–0073
9. Garcia Paucar, L.H., Bencomo, N.: Runtime models based on dynamic decision networks: enhancing the decision-making in the domain of ambient assisted living applications. In: Proceedings of the 11th International Workshop on Models@run.time, Saint Malo, France, pp. 9–17. CEUR-WS.org, October 2016. <http://eprints.aston.ac.uk/29790/>
10. Giese, H., Bencomo, N., Pasquale, L., Ramirez, A.J., Inverardi, P., Wätzoldt, S., Clarke, S.: Living with uncertainty in the age of runtime models. In: Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.) Models@run.time. LNCS, vol. 8378, pp. 47–100. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_3
11. Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Proceedings of IEEE International Conference on Requirements Engineering, pp. 94–101, April 1994
12. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2006, pp. 719–720. ACM, New York (2006). <http://doi.acm.org/10.1145/1176617.1176691>
13. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* **36**(1), 41–50 (2003). <http://ieeexplore.ieee.org/abstract/document/1160055/>
14. Koegel, M., Helming, J.: EMFStore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 2, pp. 307–308. ACM (2010)
15. 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
16. Matragkas, N.D., Kolovos, D.S., Paige, R.F., Zolotas, A.: A traceability-driven approach to model transformation testing. In: Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, 29 September 2013 (2013). http://ceur-ws.org/Vol-1077/amt13_submission_7.pdf
17. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: A repository for scalable model management. *Softw. Syst. Model.* **14**(1), 219–239 (2015). <https://link.springer.com/article/10.1007/s10270-013-0326-8>
18. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous identification and encoding of trace-links in model-driven engineering. *Softw. Syst. Model.* **10**(4), 469–487 (2011). <http://link.springer.com/10.1007/s10270-010-0158-8>
19. Wätzoldt, S., Giese, H.: Classifying distributed self-* systems based on runtime models and their coupling. In: Proceedings of the 9th International Workshop on Models at run.time, pp. 11–20 (2014). http://st.inf.tu-dresden.de/MRT14/papers/mrt14_submission_3.pdf

Some Narrow and Broad Challenges in MDD

Martin Gogolla^(✉), Frank Hilken, and Andreas Kästner

University of Bremen, Bremen, Germany
{gogolla, fhilken, andreask}@informatik.uni-bremen.de

Abstract. This contribution describes a number of challenges in the context of Model-Driven Development for systems and software. The context of the work are formal descriptions in terms of UML and OCL. One focus point is on making such formal models more approachable to standard developers.

1 Introduction

Model-driven development (MDD) is regarded today as a promising approach to system and software design, based on the idea that expressive, abstract models and not concrete code is in the focus of the deployment process. The terms ‘narrow’ and ‘broad challenges’ refer to our view that for bringing the MDD vision into practice, short term and long term goals have to be considered.

As shown in Fig. 1, in our view on the development process we distinguish between development artifacts (in the left lane) and property artifacts for quality assurance (in the right lane). Our work concentrates on formal UML and OCL models and offers on the basis of the design tool USE [5, 7] various quality assurance approaches based on testing [11], validation [6] and verification [8] for structural and behavioral [3, 10] aspects.

2 Challenges for Model and Transformation Properties

Inspired by [1] this contribution is designed to formulate some ideas for possible research in Model-Driven Development (MDD). Figure 1 shows our view for structuring and arranging challenges in MDD. The left lane sketches a (traditional) waterfall process (with feedback) using *core development artifacts* (e.g., models, code): Starting from a high-level, descriptive model various model transformations lead to an efficiently realized program of the initial model. The right lane emphasizes the role of *property artifacts* (e.g., validation scenarios, proofs, tests) that quality check the core development artifacts. The middle lane puts emphasis on involving human developers stressing the need for *human-oriented techniques* in the development process.

As there are several kinds of models (e.g., descriptive or prescriptive ones), different properties will be of interest. For example: (a) global properties valid in the complete model or local properties for model parts must be separated; (b) invariants and contracts have to be checked against implementations.

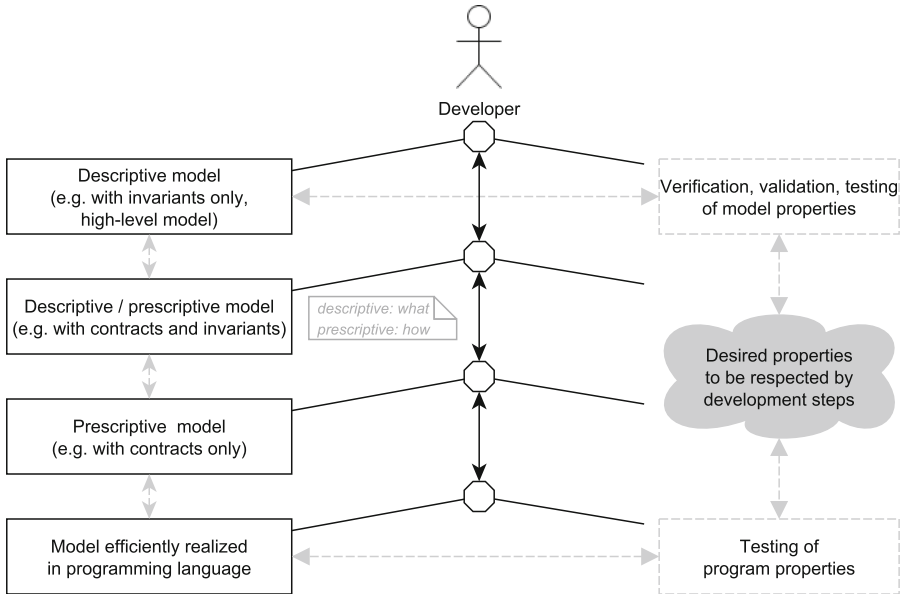


Fig. 1. Context for MDD challenges.

This implies finding and fixing appropriate techniques in the development process. For example: (a) global invariants have to be transformed into local contracts; (b) conceptual modeling features (as e.g., associations) must be turned into programming language like features (e.g., class fields); (c) platform-independent features must be specialized into platform-dependent features; (d) large models must be split into manageable small model slices; (e) generally, descriptive high-level features must be transformed into efficient low-level features.

We suggest the handling of models and model transformations with lightweight model finders and model provers. Different proving machineries have been suggested and are already employed for checking model qualities, e.g., relational logic, rewriting, description logics, logic programming, SAT, or SMT. These different approaches have all their own advantages, allow to inspect different model qualities and can coexist in the MDD world. A general strategy for the formulation of properties (of models and transformations) in an approach-independent way is however still missing. For example, we have previously developed rather particular model-to-model transformations in order to map (a) behavioral properties to efficient structural properties (filmstripping) [12], (b) multi-level models into two-level models [4], (c) complex model features into simpler ones (e.g., transformation of composition and aggregation into class diagrams with constraints) [13], or (d) linear temporal logic into UML and OCL for validation and verification purposes [10].

A continuously high priority in the field of model finders is to increase the performance in order to keep up with the ever increasing complexity of systems and their models. New techniques for the validation and verification of (partial) models are still coming up regularly [2, 16]. However, none of the existing approaches to date has a full coverage of the modeling elements and it is difficult to find – or choose – the right verification engine. A major part of the problem is the lack of benchmarks for these verification engines to compare feature sets and performance of existing tools. Without common criteria that can be compared it is difficult to judge the effectiveness of the approaches in comparison to each other. Such benchmark needs to be flexible enough to account for tools that can only handle specific validations or only support a restricted set of modeling elements.

Further challenges are located at the more detailed levels. They include the handling of (a) arithmetic or non-classical logics, (b) more data collections like tuples, (c) advanced behavioral features like state machines.

Finally, there exists no tool that can handle most types of modeling paradigms to be considered suitable for everyday use in most situations. The solution to multiple modeling problems is often scattered among multiple approaches with different tools. There exists no integration between these tools and the paradigms have to be covered individually, meaning that every paradigm has to be solved with a completely different approach with ever changing details, i.e. required artifacts and their usage.

3 Challenges for the Development Process

Formal techniques as the ones advocated by us are usually machine-oriented, not human-oriented descriptions. We see a need to make formal techniques more approachable to everyday developers and to allow for a development style that mixes formal and informal techniques.

Taking Bran Selic’s slogan “Objects before classes” seriously, we want to offer the option to start modeling with objects and to create classes based on objects. Object diagrams are less abstract than class diagrams, they represent a specific moment of time in a system. When the whole system is not known during the start of the development process, it might be easier to model such a specific moment. A class is an abstract concept, objects are more familiar, they can represent a real entity and are easier to grasp. When the goal is to model a whole system, a single object diagram will probably not be enough. But it can be used as a starting point to make educated guesses and transform it into a first version of a class diagram. Similarly, developing formal behavioral models from behavioral scenarios remains a challenging task.

To explore the possibilities of starting modeling with objects, we developed a plugin for the design tool USE. As shown in Fig. 2, it is possible to create objects without corresponding classes and links without corresponding associations, an approach that shows many similarities to partial models [15]. Most parts can be missing, to allow for more freedom in creating the object diagram. In the

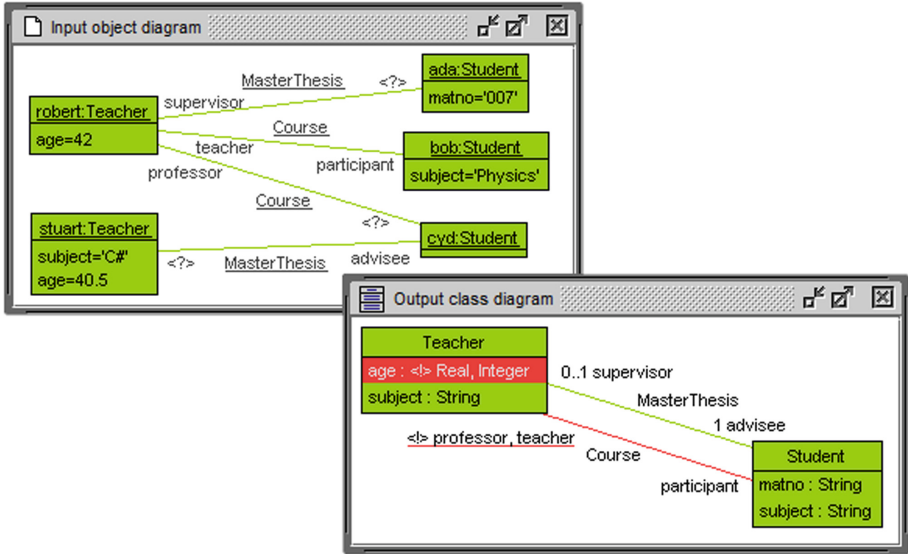


Fig. 2. Transformation example: object diagram to class diagram (Color figure online)

given input example, some role names are missing. The second diagram then shows the result of the transformation from objects to classes. Looking at the *MasterThesis* association, the links in the object diagram are merged and a completely labeled association is the result. The *Course* association however, results in a conflict, because different role names are used for the same end in the source diagram. Now that it is highlighted in the output diagram, it can be fixed in the input diagram as a next step in the iterative modeling process. Another conflict is shown in the *age* attribute of the *Teacher* class. The plugin detects different types of the attribute, which again gets highlighted. The highlighting is done using an informal notation, utilizing color and symbols. With the help of the color, the problems can be easily found, even in bigger diagrams. The symbols then highlight the specific problem. In this case, the exclamation mark highlights a conflict and the question mark highlights missing information.

This prototypical version of the plugin can already be used to get acceptable results. However, it is planned to expand the functionality. The multiplicities for example are currently given directly based on the exemplary object diagram, which is of course wrong most of the time. Instead, it is planned to make the iterative creation process more interactive and allow the user to directly input the wanted multiplicities. User input should be stored for all further iterations and only be replaced by new user input. Another future task includes the order of attributes in the classes. A concept has to be developed, how the order of the object-attributes can somehow be preserved, even though different objects might have different orders. Another limitation of the current version is the handling of

attribute types. Right now, only four different types are allowed, this needs to be expanded. Also up to further discussion is the conflict between the specific types Real and Integer, like in Fig. 2. It might be better to merge the types instead. Further ideas for extensions include the implementation of generalization, higher order associations, composition and aggregation.

Apart from considering structural aspects in taking object diagrams as the basis for the design of class diagrams, the same principle ‘from concrete to universal descriptions’ can be considered for behavioral aspects. Instance-level sequence diagrams and object diagram sequences can be taken as the starting point for behavioral descriptions like pre- and postconditions, protocol state machines or operation implementations, as this has been done to a certain extent already in [9, 14].

4 Conclusion

This contribution has shortly discussed some narrow and broad challenges for model-driven development. Our focus is and will be on formal system descriptions, however we believe that much work has to be done in order to make the many existing methods and tools approachable to software and system developers who do not have expertise on formal approaches.

References

1. Assmann, U., Bézivin, J., Paige, R., Rumpe, B., Schmidt, D. (eds.): Perspectives Workshop: Model Engineering of Complex Systems (MECS). Dagstuhl Seminar Proceedings 08331 (2008)
2. Dania, C., Clavel, M.: OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In: International Conference on Model Driven Engineering Languages and Systems, MODELS 2016, pp. 65–75. ACM (2016)
3. Doan, K.-H., Gogolla, M., Hilken, F.: Towards a developer-oriented process for verifying behavioral properties in UML and OCL models. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 207–220. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_15
4. Gogolla, M.: Experimenting with multi-level models in a two-level modeling tool. In: Atkinson, C., Grossmann, G., Kühne, T., de Lara, J. (eds.) Proceedings of 2nd International Workshop on Multi-Level Modelling (MULTI 2015), vol. 1505, pp. 3–11. CEUR Proceedings (2015). <http://ceur-ws.org/Vol-1505/>
5. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.* **69**, 27–34 (2007)
6. Gogolla, M., Hamann, L., Hilken, F., Kuhlmann, M., France, R.B.: From application models to filmstrip models: an approach to automatic validation of model dynamics. In: Fill, H., Karagiannis, D., Reimer, U. (eds.) Proceedings of Modellierung (MODELLIERUNG 2014), GI, LNI, vol. 225, pp. 273–288 (2014)
7. 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 Modellierung (MODELLIERUNG’2016), GI, LNI, vol. 254, pp. 203–218 (2016)

8. Gogolla, M., Hilken, F., Niemann, P., Wille, R.: Formulating model verification tasks prover-independently as UML diagrams. In: Anjorin, A., Espinoza, H. (eds.) ECMFA 2017. LNCS, vol. 10376, pp. 232–247. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_14
9. Grønmo, R., Møller-Pedersen, B.: From UML 2 sequence diagrams to state machines by graph transformation. *J. Object Technol.* **10**(8), 1–22 (2011)
10. Hilken, F., Gogolla, M.: Verifying linear temporal logic properties in UML/OCL class diagrams using filmstripping. In: Kitsos, P. (ed.) *Proceedings of Digital System Design (DSD 2016)*, pp. 708–713. IEEE (2016)
11. Hilken, F., Gogolla, M., Burgueno, L., Vallecillo, A.: Testing models and model transformations using classifying terms. *Softw. Syst. Model.* (2016). <https://doi.org/10.1007/s10270-016-0568-3>. Accessed 09 Dec 2016
12. Hilken, F., Hamann, L., Gogolla, M.: Transformation of UML and OCL models into filmstrip models. In: Di Ruscio, D., Varró, D. (eds.) *ICMT 2014*. LNCS, vol. 8568, pp. 170–185. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_13
13. Hilken, F., Niemann, P., Gogolla, M., Wille, R.: From UML/OCL to base models: transformation concepts for generic validation and verification. In: Kolovos, D., Wimmer, M. (eds.) *ICMT 2015*. LNCS, vol. 9152, pp. 149–165. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21155-8_12
14. Kaufmann, P., Kronegger, M., Pfandler, A., Seidl, M., Widl, M.: A SAT-based debugging tool for state machines and sequence diagrams. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) *SLE 2014*. LNCS, vol. 8706, pp. 21–40. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_2
15. Salay, R., Chechik, M., Famelis, M., Gorzny, J.: A methodology for verifying refinements of partial models. *J. Object Technol.* **14**(3), 3:1–31 (2015)
16. Semeráth, O., Varró, D.: Graph Constraint evaluation over partial models by constraint rewriting. In: Guerra, E., van den Brand, M. (eds.) *ICMT 2017*. LNCS, vol. 10374, pp. 138–154. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61473-1_10

Modelling by the People, for the People

Steven Kelly  

MetaCase, Jyväskylä, Finland
stevek@metacase.com

Abstract. Domain-specific modelling has moved the focus of model content from the implementation domain to the problem domain. However, much research still sees most modellers as programmers – a tacit assumption that is exemplified by the use of IDEs as modelling environments. Model-Driven Engineering research should instead be reaching out to make itself presentable to subject matter experts – as language creators, users, and research subjects. Every leap in developer numbers has been triggered by a new format, and we need another such leap now. Domain-specific modelling is ideally placed to step up and enable the creation of applications by people of all backgrounds.

Keywords: Domain-specific modeling · Productivity
Programmer demographics

1 Introduction

Domain-specific modelling has moved the focus of model content from the implementation domain to the problem domain [1]. However, much Model-Driven Engineering (MDE) research still sees most modellers as programmers – a tacit assumption that is exemplified by the use of IDEs as modelling environments, diagrams bearing a striking resemblance to UML, and structures straitjacketed into the hierarchical tree of XML. The false assumption seems to be “because I must build the language workbench in a programming IDE, experts must create their languages in that IDE, and modellers must model in that IDE”. MDE research is stuck in a programmer mindset and in IDEs with roots in the last millennium, when it should be reaching out to make itself presentable to subject matter experts, both as language creators and users.

This paper looks at the development of our industry over its whole history, considering the broad factors that are at play in its evolution. The problem of the focus on programmers is shown to be the lack of an infinitely expandable supply of programmers, and the inevitable effects on average programmer productivity caused by the natural ‘best first’ allocation of programmers from the high end of the ability scale.

2 Developments in Developer Demographics

Every leap in developer numbers has been triggered by a new format, either in languages (machine code to assembly language to third-generation languages (3GLs)) or devices (mainframes to PCs to mobile). ‘Language leaps’ came from research making

software development easier, so that more people are able to successfully create applications. ‘Device leaps’ are different: rather than increasing the supply and decreasing the cost, they increased the demand. Language leaps have increased developer numbers much more than devices leaps: an order of magnitude with the move to assembly or to 3GLs, as opposed to a ‘mere’ doubling alongside the PC revolution in the early 1980s, when the number of computers grew by a factor of over 25 (Fig. 1).

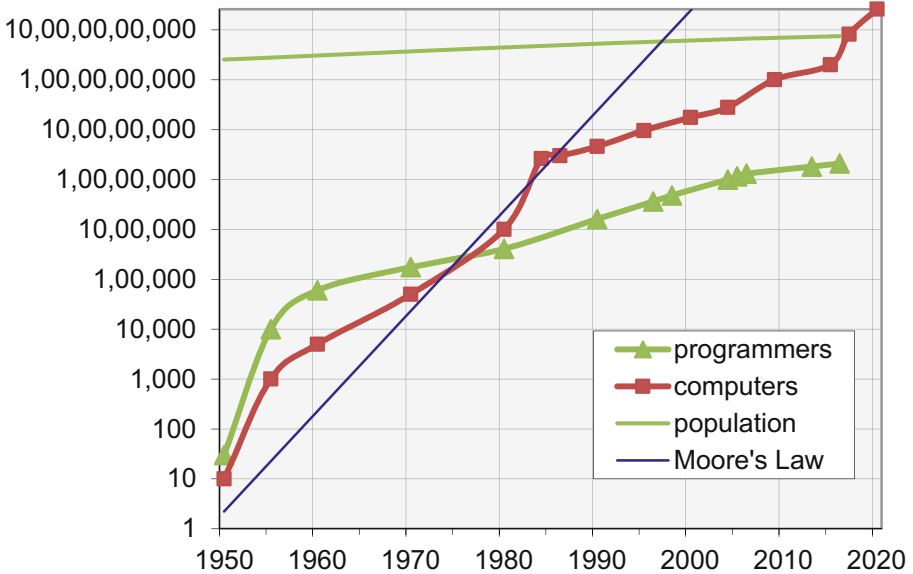


Fig. 1. Growth in numbers of programmers and computers globally

Interestingly, the increase in the number of developers in the language leaps seems to mirror the increase in productivity of a given average developer, e.g. roughly five times faster with 3GLs compared to assembly language. Since the early 3GLs, there has not been an appreciable leap in productivity, so it seems fair to assume that we have made only incremental progress in making software development accessible to people for whom it was previously too hard. Learning and working in JavaScript is not much easier than learning and working in COBOL. Contrast that with how much easier either is than working directly in machine code.

At the start of the 1980s, there were 313,000 programmers in the USA; a decade later, one million. (The US percentage of world programmers shrunk only modestly, from 77% to 62%, so this figure is useful globally too.) Let us assume a normal distribution for programming skill in the population: any other plausible distribution gives essentially the same results. For the sake of familiarity, and in the lack of better data, let us take the same mean of 100 and standard deviation of 15 as in IQ tests. Similarly, for simplicity let us take the US population to be constant at 240 million (only a few percent different at either end of that period). As programming was by no

means a new profession in 1980, let us further assume that the programmers then were the top people in terms of innate programming skill, and that the rest of the million added during that decade were the next best, rather than distributed more widely. Many assumptions, but we are not seeking an exact answer, just a general picture: What does the innate skill of programmers look like over time?

The programmers in 1970 ranged from percentile 99.933 to 100, so the median programmer was at percentile 99.966. By 1980, new programmers were being added at percentile 99.870. In 1990, the number of programmers in the US passed one million, so percentile 99.73. By 2000, the number of programmers in the US passed 3 million, so new programmers then were being added at percentile 98.75. On the IQ scale¹, these percentiles would correspond to scores of 151, 145, 142 and 134. For a programmer working since 1970, there has been a significant drop in the expectation of a team-mate's innate ability back then compared to a new programmer in 2000.

In fact, innate programming skill is made up of more than IQ. IQ tests, being a series of small problems, do little to measure ability to cope with complexity. That skill, increasingly important for today's programmers, is thus a somewhat independent variable, plausibly also on a normal distribution. Another attribute useful for programming is being able to put up with low-level detail (whether of problems, languages or tools). Some smart people like that, others hate it; again, it seems an independent variable. A fourth skill is being able to communicate with and work with people – a skill which seems almost to correlate inversely with the previous one, and which the stereotype of a programmer is generally seen as lacking. With some positive correlation and some negative correlation on these skills, we can perhaps take them as four independent normal distributions. What does innate programming skill look like, if it is the product of four normally distributed independent variables? Each additional normal variable sharpens the peak of the distribution, further increasing the drop in scores for each percentile change at the top tail of the distribution (Fig. 2).

For any given programmer, there is significant variability in development velocity, depending on the difficulty of the task. In particular, near the limit of the programmer's ability, velocity drops sharply, becoming zero at the limit. As the innate ability of new programmers has dropped, the proportion of tasks which those programmers can achieve progress with has dropped. When considered as part of a team, an individual's productivity contribution becomes negative around the limit of his ability.

In 2004, the number of programmers in the US peaked at 4 million, shrinking by 10% over the next decade before starting to grow again slowly, but still remaining below 4 million. Outsourcing had an effect, as did the broader economy, but both of those have now largely been overturned. As the level of programming skill available outside the current ranks of programmers decreased, with no easing of the complexity of tasks or improvements in the languages used, there came a point where a corollary to Brooks' Law [3] was reached: Adding new programmers to an industry would have a negative effect on production.

This cessation in growth of the number of programmers in the US has lasted over a decade. At the same time the US population has grown over 10%, and the demand for

¹ <https://www.iqcomparisonsite.com/iqtable.aspx>.

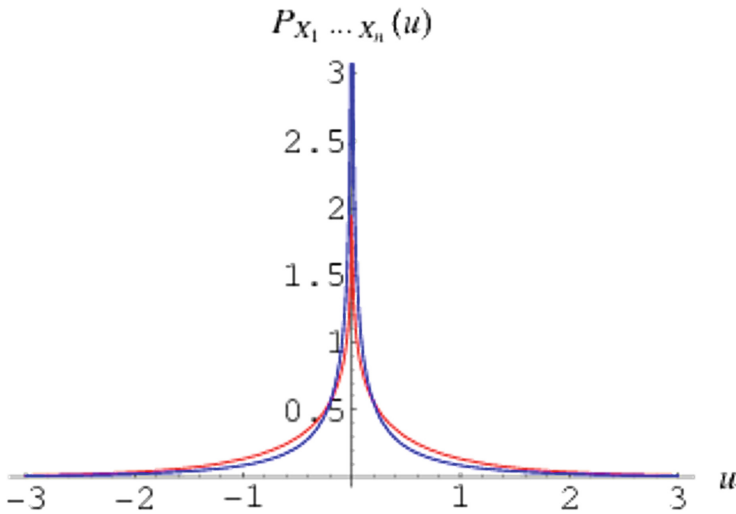


Fig. 2. Product of two (red) and three (blue) normally distributed variables [2] (Color figure online)

applications has grown significantly with the rise of the web, the smart phone and other devices. Part of the demand has been met by programmers from overseas. However, these have largely been meeting the needs of a US market rather than their own home markets, and as their home markets grow, and wages even out across the world, this temporary relief will inevitably dry up. When we hit our corollary to Brooks' Law globally, what can step in to save us?

3 Libraries, Frameworks and Languages

Components, libraries and frameworks are sometimes touted as offering a decisive advantage. A pre-existing component or library may save a company's time, but if there is enough need to make working without a library inefficient, and no such library yet exists, a company will simply assign a developer to write the library (explicitly or implicitly). Writing the application code that uses it will of course be largely the same, regardless of whether it was created in-house or available from elsewhere: one may be better-targeted, whereas the other may be more mature. A library thus offers a productivity gain to a company, but does not generally change the task of the programmers who use it, or their productivity on that task.

A framework can be described as 'a library with attitude': it does more than offer functions, letting you 'fill in the blanks' of its ready-built behaviour, and bounding and guiding you on the content of those blanks. With respect to the problem domain task and the architecture, a framework thus makes things easier for the software developer, increasing productivity. But for the majority of frameworks, the content entered 'in the blanks' is still largely 3GL code. If you are not up to the task of writing 3GL code, you

cannot successfully create an application even with the framework. A framework alone will not make application creation available to a larger audience.

A domain-specific language, on the other hand, helps both in increasing productivity and in reducing the demands on the developer. The ‘blanks’ are filled in with domain concepts, not code. By abstracting away from the implementation technology, a DSL also often makes it possible to generate the same application for different formats: web server and client, desktop app, and mobile app.

A graphical or projectional DSL in particular generally constrains you to only be able to create legal models, as opposed to the illegal source code you could write while trying to use a framework. Among programmers there are clearly a disproportionate number who prefer text over graphics, compared to the population at large; conversely, a graphical DSL is thus often a better choice when reaching out to current non-programmers.

4 Related Research

Over such a broad period of history, the amount of related research is huge. We will limit ourselves to either end of the period, to see what – if anything – has changed, and whether there are new directions on the rise.

In 1954, when most development was still in machine code and moving to assembly language, MIT hosted a session called “Future Development of Coding – Universal Code” [4] on the use of flow diagrams in coding. 16% of participants had “a higher level individual set up a flow diagram and persons at a lower level do the coding of each block”. Grace Hopper “remarked that flow diagrams offer a potent means of communication since they are not tied to the computer.” C. W. Adams “suggested that if the specifications are sufficiently rigid, the machine itself could handle the coding”: automatically generating full code from flow diagrams. G. E Reynold’s group built diagrams with “magnetized blocks that are easy to erase and to assemble.”

Model-Driven Engineering has achieved its best successes when based on Domain-Specific Modelling, rather than UML or text-based “models” [5]. A number of tools exist, with the most successful being distinct from programming IDE or abstracting away from it. Representational formats have coalesced around graphical diagrams, which have changed little over the last 60 years. A growing trend changes those ‘box and line’ diagrams, at least for beginners, to blocks that connect by interlocking [6, 9]. Modelling is also starting to become feasible on mobile platforms [7].

5 Conclusion

Good domain-specific modelling languages in non-IDE tooling have consistently shown productivity improvements of a factor of 5–10 [5]. Given the earlier link between productivity increase and increase in the number of people who can successfully develop software, there would seem to be a good possibility that a DSM approach could make application creation possible for an order of magnitude more people. Our own experience with MetaEdit+ [8] tends to confirm this: we have clients

where none of the modellers are programmers, clients where all are, and clients where there is a mix. The future could well reach the ideal of a balanced mix of good programmers and smart subject matter experts, all collaborating directly in the same set of models. MDE is ideally placed to step up and make possible the creation of applications of all formats, by people of all backgrounds.

Since the 1990s, much modelling research has focused on creating tools and languages, but with declining returns in terms of novelty, incremental benefit over current industrial use, and adoption. For tools, researchers need to throw off their “not invented here” attitudes, and reach out to investigate real industrial use of non-IDE based commercial domain-specific modelling tools, both language workbenches and fixed tools like Simulink and LabVIEW. For languages, rarely does a researcher have the domain knowledge and experience to create a good DSM language for a company. Rather, they should concentrate on being a catalyst, enabling a company to create its own language, studying that process, and incrementally improving it as they learn over several cases. In this area, more research is definitely needed and will be fruitful.

References

1. Kelly, S., Tolvanen, J.-P.: *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, Hoboken (2008)
2. Wolfram Research, Inc.: Normal Product Distribution. <http://mathworld.wolfram.com/NormalProductDistribution.html>
3. Brooks, F.: *The Mythical Man Month Essays on Software Engineering*, 2nd edn. Addison-Wesley, Reading (1995)
4. Adams, C., Hopper, G.: Future development of coding – universal code. In: Adams, C., Gill, S., Combelic, D. (eds.) *Digital Computers – Advanced Coding Techniques*, Summer Session 1954, pp. 80–81 (1954)
5. Tolvanen, J.-P., Kelly, S.: Model-driven development challenges and solutions: experiences with domain-specific modelling in industry. In: *Proceedings of MODELSWARD 2016, 4th International Conference on Model-Driven Engineering and Software Development*, pp. 711–719, February 2016
6. Bau, D., Gray, J., Kelleher, C., Sheldon, J., Turbak, F.: Learnable programming: blocks and beyond. *Commun. ACM* **60**, 72–80 (2017)
7. Vaquero-Melchor, D., Garmendia, A., Guerra, E., de Lara, J.: Towards enabling mobile domain-specific modelling. In: *ICSOFT*, pp. 117–122 (2016)
8. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+ a fully configurable multi-user and multi-tool CASE and CAME environment. In: Constantopoulos, P., Mylopoulos, J., Vassiliou, Y. (eds.) *CAiSE 1996*. LNCS, vol. 1080, pp. 1–21. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61292-0_1
9. Weintrop, D., Wilensky, U.: To block or not to block, that is the question: students’ perceptions of blocks-based programming. In: *Proceedings of the 14th International Conference on Interaction Design and Children*, New York, NY, USA, pp. 199–208 (2015)

From Building Systems Right to Building Right Systems

A Generic Architecture and Its Model Based Realization

Vinay Kulkarni^(✉) and Sreedhar Reddy

Tata Consultancy Services Research, Pune, India
{vinay.vkulkarni, sreedhar.reddy}@tcs.com

Abstract. Model driven engineering community has made considerable progress in the last decade and a half as regards developing software systems with enhanced productivity, quality and platform independence. However, in the increasingly dynamic world, enterprises are facing a different kind of challenge where the focus shifts from *how to build* to *what to build*. This paper proposes a shift in focus of model driven engineering community to meet these challenges. We outline an approach and a research agenda to realize the same.

Keywords: Modeling language engineering · Decision space models
Model mapping · Modeling and simulation

1 Introduction

Software systems are becoming ever pervasive. Systems are no longer islands and resemble more and more an ecosystem or system of systems. At the same time, they are subjected to a variety of change drivers such as globalization, regulatory changes, new business models, technology churn etc. Enterprises need to respond to these change drivers by suitably adapting business strategy, organisation structure, operating processes and business systems. Moreover, to stay competitive, these adaptations need to happen with minimal disruption, least cost, and shortest possible time. Considering the size and complexity of modern enterprise, this is time-, cost- and intellectually-intensive endeavour [1].

Since mid-90's, Model Driven Engineering (MDE) community has been working on this challenge but with software system development as its primary focus. Modeling languages such as UML, BPMN have been designed to model different concerns of a software system. Transformation mechanisms for automatically generating code from these models have also been developed. These advances have led to enhanced development productivity, quality and ease of keeping pace with technology advance [2]. Thus, it can be said that focus of MDE community has been on *how to develop a system right*.

Enterprise is typically viewed in terms of three planes namely Strategy, Process and System. The strategy plane defines the high level business goals and devises business strategies to achieve them. The process plane defines operational processes and

organisational structure to realize the business strategies. The systems plane supports execution of operational processes in as automated a manner as possible. As the enterprise operates in an increasingly dynamic environment, response to a change in the environment might require a change in any of these planes. Introduction of such a change can have ripple effects requiring changes at multiple places within as well as across planes. As a result, implementing a change is time-, cost- and effort-intensive activity [3]. On the other hand, the time window available for response is getting increasingly shorter. As a result, the cost of delayed and/or erroneous response can be prohibitive. Moreover, typically, decisions once taken up for implementation are prohibitively expensive to undo. Therefore, it should not come as a surprise that most of enterprise transformation initiatives hardly ever reach completion¹.

Thus, there is a clear need for an infrastructure to support decision making processes that provide a-priori certainty on the outcomes of decisions. This involves, at a minimum, the mechanisms to capture the necessary and sufficient information pertinent to the decisions in a formal manner. It should also provide for machinery to perform qualitative and/or quantitative analysis of the outcomes of decisions where the decision space is characterized by partial information and inherent uncertainty. The machinery should be capable of dealing with uncertainty and incomplete information. It should leverage domain knowledge to navigate the design space in an informed manner to help arrive at *right* systems. The machinery should also enable a mapping from system design space to system implementation space to ensure the systems are built *right*. It should also provide for a learning machinery to derive insights from the past to help arrive at and build *right* systems. This infrastructure should also be available at run time so as to support dynamic adaptation. Thus, the focus of MDE community should extend from *building systems right* to *building the right systems*.

Though this approach has wide applicability, we discuss it in enterprise context. Section 2 provides an overview of the proposed approach. Section 3 provides an outline of research agenda for realizing the approach. Section 4 outlines a few exemplars illustrating use cases for the proposed line of attack. Section 5 provides a summary.

2 Overview of Proposed Approach

Future enterprises are systems of systems with complex interactions operating in a dynamic environment. Given the structural and behavioral complexity, detailed understanding is possible only in localized contexts. At the same time, events occurring in one context influence the outcomes in other contexts. Lack of complete information coupled with inherent uncertainty make holistic analysis of systems intractable. As a result, decisions pertaining to system design and implementation are unlikely to be globally optimal. Non-availability of complete information and inherent uncertainty make traditional optimization approaches impractical. Therefore, simulation-based approach is the only recourse available for arriving at a “good enough” solution by

¹ <http://www.valueteam.biz/why-72-percent-of-all-business-transformation-projects-fail>.

navigating the design space [4]. However, considering the open nature of the problem space an exhaustive navigation of the design space is infeasible. Thus, intelligent navigation of design space guided by domain knowledge and learning from past experience is called for. Figure 1 provides a pictorial description of a possible line of attack that hinges on: (i) a layered view of enterprise, (ii) model-based machinery to help arrive at right systems, (iii) model-based machinery to help implement the system right, and (iv) a mechanism to map the two sets of models and a means to derive one from the other.

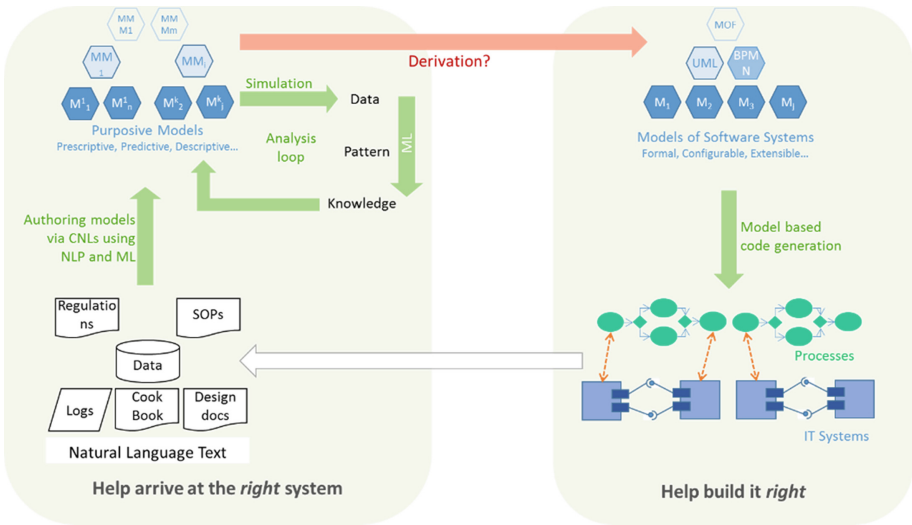


Fig. 1. Model based machinery to support future enterprises

The line of attack calls for availability of different kinds of models and modeling formalisms for prescriptive and predictive analyses. Much of the information to construct these models is available in enterprise but in a fragmented and heterogeneous form. This calls for a machinery to extract the relevant information from these fragmented sources and process it in the right context to discover/construct/integrate the right models. The right analysis machinery is required to play out what-if and if-what scenarios using these models. The results of analysis need to be interpreted and insights extracted to be fed back into the models. This is typically human-in-the-loop activity. The analysis and synthesis burden on human experts can be significantly reduced by having machinery that can extract patterns and knowledge from the data. Having arrived at the *right* system models, they need to be realized through right system implementations. Model based software development approaches can be used for this purpose [5–7]. There is a need for a bi-directional mapping between system models and implementation models. Such a mapping will enable derivation of implementation models from system models and will also help feed back the insights extracted from data produced by implemented systems into the system models. Thus, the proposed

modeling machinery not only helps at design time but also at run time facilitating a continuous adaptation loop. Arriving at this bidirectional mapping is an open challenge.

The models required to help arrive at the right systems need to address the *why*, *what*, *how* aspects and the mappings between them from the perspectives of the relevant stakeholders (i.e. *who*) [8]. The Why model specifies business goals and relationships between goals such as decomposition, dependency, subsumption, conflict etc. The What model specifies high level business strategies and associated trade-offs, and the Key Performance Indicators (KPI) in terms of which goal achievement can be ascertained. The How model specifies detailed operational processes and workflows involving the organisational structure and resources (i.e. Who model). Underpinning these four models there is a domain model (ontology) that serves as semantic reference. These models need to be simulatable independently as well as together so as to support what-if and if-what scenario playing. Thus, simulation machinery constitutes the necessary part for supporting design space navigation [9]. However, exhaustive navigation is not scalable. Therefore, we need mechanisms that help draw upon domain knowledge and past data to guide the navigation in the right direction [10]. It should be possible to capture the learnings from past navigations in the form of reusable knowledge.

The models required for helping implement the decision are already available in the current MDE infrastructure^{2,3,4}. It is possible to capture software system requirements formally so as to refine those further to a form from which efficient implementation can be generated.

In order to ensure consistent realization of system, we need to ensure traceability of three kinds: (i) across the models used for implementing the system, (ii) across the models used for arriving at the right system, and (iii) across the decision-space and implementation-space models.

Implementation-space is well-understood in the sense the structures to be used, their semantics, and their relationships can be readily discerned for a problem. MOF enables creation of purposive modeling languages and mappings between them, and a number of proven modeling and model transformation languages are already available. This machinery suffices to address all modeling needs of implementation-space.

On the other hand, decision-space is not so well understood. While there do exist some models with formal semantics [11], much of the available machinery comprises of informal high level notations lacking in formal semantics. Therefore, it is difficult to define the mappings between decision-space models. These deficiencies pose problems in designing the system right. For instance, it is not possible to carry out what-if and if-what analysis to determine the outcomes of the decisions. As decision spaces are related, the associated models need to be analysed in an integrated manner which is not possible for want of well-defined mappings. Thus, existing decision-space modeling machinery is inadequate to support design-space exploration in an effective and

² <http://www.omg.org/spec/UML/>.

³ <http://www.omg.org/spec/MOF/>.

⁴ <http://www.omg.org/spec/QVT/>.

efficient manner. It is not even sufficient to support traceability between the various decision-space models.

Due to the informal nature of decision-space models, it is difficult to maintain traceability between decision-space and implementation-space models. As a result, change impact analysis across the two spaces is not possible.

3 Research Agenda

We outline research required to come up with an integrated infrastructure that supports decision making processes to design right systems and implement them the right way. Specifically, the following investigations need to be carried out:

- In enterprise decision-space, there is a need to come up with formal models addressing all the relevant aspects that bear on decision making. For instance, it should be possible to specify goals and relationships such as [de]composition, dependency, conflicts etc. between them. A goal should be specifiable in terms of measurable quantities which in turn should be traceable to the system parameters as shown in Fig. 2. The Goal-Measure-Lever graphs need to be specified for each stakeholder and integrated together into a consistent whole. The decision-making endeavour can be viewed as “tweak a lever – observe the measures – check the goal” loop. Considering the size and complexity of modern enterprises, these graphs would typically be huge in size making manual construction and analysis quite impractical.

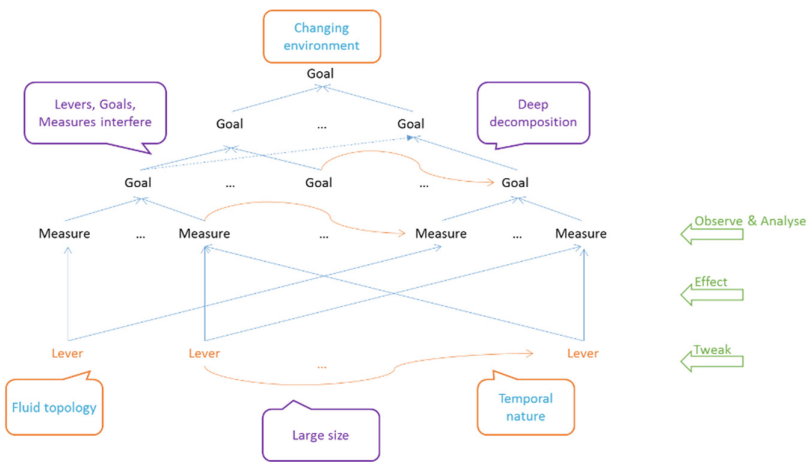


Fig. 2. Complex dynamic decision making

- The Goal-Measure-Lever graph of Fig. 2 should be specifiable using a simulatable language thus automating “tweak a lever – observe the measures – check the goal” loop. It should cater to what-if and if-what scenario playing. This machinery forms the primitive building blocks for navigating the design space.

- Exhaustive navigation of decision space can be prohibitively expensive. How can domain knowledge and past decision traces be exploited to help navigate the design space in a more efficient manner?
- To mimic real-world phenomena where the behaviour of one entity influences the behaviour of others, the different models constituting system specification need to be co-simulated. Since the underlying formalisms of these models could be quite different, supporting co-simulation is a challenge.
- Constructing decision-space models is time-, effort- and intellectually-intensive endeavour. Information to construct these models exists in multiple heterogeneous resources such as enterprise data, reports, literature etc. How to make use of this information to construct the desired models in as automated a manner as possible?
- How to capture the learnings from the usage of this decision-space machinery e.g. rules, cases, heuristics, predictive models etc. to enable reuse in future design cases?
- Typically, enterprises evolve in a siloed manner leading to information fragmentation over heterogeneous data sources. There is a need to provide enterprise wide unified view of this information through mappings over information models of these data sources. How to discover these mappings?
- How to make the modeling and model processing machinery easy-to-use for domain people? What kind of Domain Specific Languages (DSL) are required? How do these DSLs map to the underlying formal models?
- What is the right methodology to tie together the end-to-end modeling and model processing machinery spanning design and implementation spaces in a seamless manner?
- How can the proposed modeling machinery be integrated with runtime system so as to support dynamic adaptation as well?
- What is a prototypical problem space in enterprise where the proposed approach can be effectively illustrated?

The above research agenda calls for a multi-disciplinary effort spanning modeling language engineering, knowledge engineering, natural language processing, machine learning, software engineering, and management sciences. We have done initial explorations of some of the research space with promising leads emerging [9, 10].

4 Exemplars

We discuss a few use-cases for the approach outlined in this paper. Each of them illustrates a partial realization of a part of the vision outlined in this paper.

4.1 Digital Manufacturing

Manufacturing is expected to be more and more customer centric to the extent the product gets manufactured as per the specifications provided by customers. As a result, manufacturing supply chains need to be more dynamic wherein the production, scheduling and inventory control processes need to be significantly more responsive. For instance, the mobile handset industry has seen several disruptions in the last decade

– from feature phones to touchscreens to smartphones capable of high quality video experience. The ability to rapidly adapt to changing market demands was essential for survival⁵. A digital manufacturing organisation would model the production processes, supply chains, the market, the competition etc. and use these models individually and together to play out scenarios of interest so as to devise suitable interventions. This helps the digital manufacturing organisation to be more responsive with a degree of control on certainty of outcomes [12, 13].

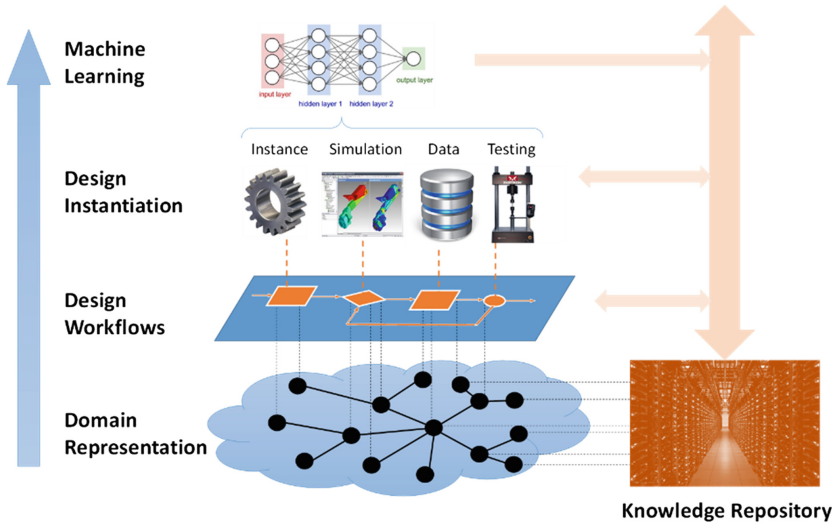


Fig. 3. ICME platform

4.2 Integrated Computational Materials Engineering (ICME)

ICME is a new paradigm for designing materials and manufacturing processes in an integrated manner [14]. It uses modeling and simulation to explore the design space [15, 16]. The approach also draws heavily on domain knowledge and machine learning to guide design space exploration as shown in Fig. 3. This has been validated for problems such as gear design, advanced high strength steel design etc. [17].

4.3 Model Driven Organisation (MDO)

MDO is a new paradigm for modeling all relevant aspects of an enterprise so as to support run-the-enterprise as well as change-the-enterprise processes [18]. It brings the ideas of Model Reference Adaptive Control to enterprise modeling [19] as shown in Fig. 4. It enables modeling of enterprise as a socio-technical system wherein the Why,

⁵ <https://www.wired.com/2012/04/5-reasons-why-nokia-lost-its-handset-sales-lead-and-got-downgraded-to-junk/>.

What, How and Who aspects get specified in a manner that is amenable to what-if and if-what scenario playing [9]. It provides means for storing history i.e. execution traces that can be mined for monitoring and sense-making [20] so that domain expert can take meaningful adaptation decisions [21]. This approach has been validated on industry-critical problems in laboratory setting [22].

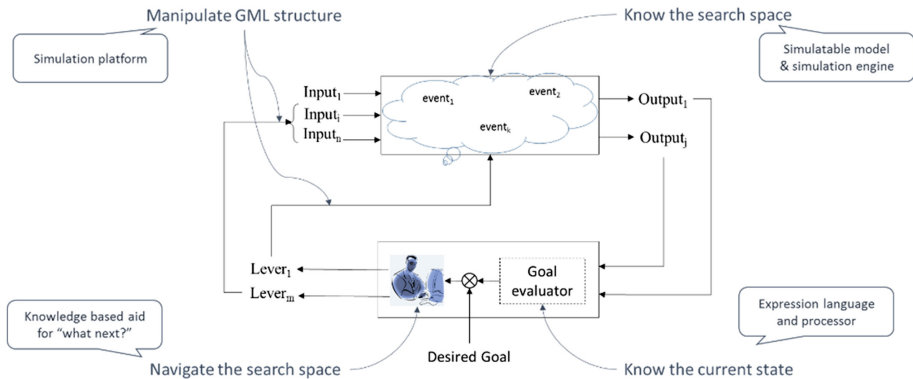


Fig. 4. Supporting adaptation under human supervision

5 Summary

The emerging digital forces of mobility, big data, social media, cloud computing and robotics are resulting in increasing pervasiveness of computing. Coupled with challenges emanating from increasing dynamism due to the connected world, enterprises are faced with a whole new set of challenges. This changing reality puts a new set of demands on software systems. The erstwhile paradigm of using software systems essentially to obtain mechanical advantage through automation of operational processes is no longer adequate. A new paradigm seems called for to meet these challenges. We have argued for a shift in focus of MDE community from *building systems right* to *building right systems*. We sketched a possible line of attack towards evolving such a new paradigm. We outlined a research agenda to support the proposed line of attack. We presented a set of teaser exemplars of enterprise systems of future that can be realized using the proposed line of attack.

References

1. Rouse, W.B., Baba, M.L.: Enterprise transformation. *Commun. ACM* **49**(7), 66–72 (2006)
2. Stahl, T., Volter, M.: *Model-Driven Software Development*. Wiley, Hoboken (2006)
3. Barjis, J.: Enterprise modeling and simulation within enterprise engineering. *J. Enterp. Transform.* **1**(3), 185–207 (2011)

4. Gosavi, A.: *Simulation-Based Optimization. Parametric Optimization Techniques and Reinforcement Learning*. Springer, Heidelberg (2003). <https://doi.org/10.1007/978-1-4899-7491-4>
5. Selic, B.: The pragmatics of model-driven development. *IEEE Softw.* **20**(5), 19–25 (2003)
6. Hailpern, B., Tarr, P.: Model-driven development: the good, the bad, and the ugly. *IBM Syst. J.* **45**(3), 451–461 (2006)
7. Kulkarni, V., Reddy, S.: Separation of concerns in model-driven development. *IEEE Softw.* **20**(5), 64–69 (2003)
8. Zachman, J.: The Zachman framework for enterprise architecture. *Zachman Int.* **79** (2002)
9. Barat, S., Kulkarni, V., Clark, T., Barn, B.: A simulation-based aid for organisational decision-making. In: *ICSOFT-PT 2016*, pp. 109–116 (2016)
10. Yeddula, R.R., Vale, S., Reddy, S., Malhotra, C.P., Gautham, B.P., Zagade, P.: A knowledge modeling framework for computational materials engineering. In: *SEKE*, pp. 197–202 (2016)
11. Meadows, D.H., Wright, D.: *Thinking in Systems: A Primer*. Chelsea Green Publishing, White River Junction (2008)
12. Cooper, K., Sardar, G., Tew, J.D., Wikum, E.: Reducing inventory cost for a medical device manufacturer using simulation. In: *Winter Simulation Conference 2013*, pp. 2109–2115 (2013)
13. Cooper, K., Wikum, E., Tew, J.D.: Evaluating cost-to-serve for a retail supply chain. In: *Winter Simulation Conference 2014*, pp. 1955–1964 (2014)
14. Allison, J., Backman, D., Christodoulou, L.: Integrated computational materials engineering: a new paradigm for the global materials profession. *JOM J. Miner. Met. Mater. Soc.* **58**(11), 25–27 (2006)
15. Gautham, B.P., Reddy, S., Das, P., Malhotra, C.: Facilitating ICME through platformization. In: Mason, P., et al. (eds.) *Proceedings of the 4th World Congress on Integrated Computational Materials Engineering (ICME 2017)*, pp. 93–102. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57864-4_9
16. Das, P., Yeddula, R.R., Reddy, S.: A model driven framework for integrated computational materials engineering. In: *ModSym+SAAS@ISEC 2016*, pp. 27–32 (2016)
17. Gautham, B.P., Kulkarni, N.H., Panchal, J.H., Allen, J.K., Mistree, F.: A method for the preliminary design of gears using a reduced number of American Gear Manufacturers Association (AGMA) correction factors. *Eng. Optim.* **49**(4), 565–582 (2017)
18. Clark, T., Kulkarni, V., Barn, B., France, R.B., Frank, U., Turk, D.: Towards the model driven organization. In: *HICSS 2014*, pp. 4817–4826 (2014)
19. Isermann, R., Matko, D., Lachmann, K.H.: *Adaptive Control Systems*. Prentice-Hall, Inc., Upper Saddle River (1992)
20. Clark, T., Kulkarni, V., Barat, S., Barn, B.: Actor monitors for adaptive behaviour. In: *ISEC 2017*, pp. 85–95 (2017)
21. Barat, S., Kulkarni, V., Clark, T., Barn, B.: A model based realisation of actor model to conceptualise an aid for complex dynamic decision-making. In: *MODELSWARD 2017*, pp. 605–616 (2017)
22. Barat, S., Kulkarni, V., Clark, T., Barn, B.: An actor-model based bottom-up simulation - an experiment on Indian demonetisation initiative. In: *Winter Simulation Conference, Las Vegas, USA, 3–6 December 2017* (2017)

The Tool Generation Challenge for Executable Domain-Specific Modeling Languages

Tanja Mayerhofer¹(✉) and Benoit Combemale²

¹ TU Wien, Vienna, Austria
mayerhofer@big.tuwien.ac.at

² University of Toulouse, Toulouse, France
benoit.combemale@irit.fr

Abstract. Executable domain-specific modeling languages (xDSMLs) have the potential of bringing major benefits to the development of complex software-intensive systems as they provide abstractions of complex system behaviors and allow for early analyses of that behavior. However, in order to be useful, xDSMLs have to be equipped with model analysis tools supporting domain engineers in comprehending, exploring, and analyzing modeled behaviors. Hand-crafting such tools in an ad hoc manner imposes significant efforts and costs on the development process and is, hence, mostly done for broadly adopted xDSML only. Executable metamodeling approaches seek to overcome this limitation by providing formalisms to define the execution semantics of xDSMLs in a systematic way building the basis for automatically generating model analysis tools. While significant advances towards achieving this vision have been achieved in recent years, there are still many challenges to be solved for generating out-of-the-box analysis support for xDSMLs. In this paper we revisit the tool generation challenge introduced by Bryant et al. [3] seven years ago reflecting on recent achievements and identifying open challenges.

1 Introduction

Model driven engineering (MDE) aims at reducing the accidental complexity associated with the development of complex software-intensive systems resulting from the large gap between concepts used by domain experts to express their problems and concepts provided by general-purpose programming languages to implement solutions. MDE addresses this problem through domain-specific modeling languages (DSML) enabling domain experts to overcome the gap between problem space and solution space through suitable abstractions and transformations [17].

Although there are many examples of the successful use of DSMLs to improve development productivity and quality [7, 17], it has been also recognized that the development of DSMLs is itself a challenging task. This resulted in the emergence of the software language engineering (SLE) discipline defined as the application

of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages [8].

SLE brought forward potential solutions for the development of DSMLs and supporting infrastructures that allow their effective use and deployment. Notable achievements in this area include the establishment and broad adoption of meta-modeling techniques for formalizing the abstract syntax of DSMLs [9], which lead to the development of generic and generative approaches providing model editing and static analysis facilities for DSMLs out-of-the-box once their metamodels are defined.

Executable DSMLs (xDSMLs) play an important role in the development of complex software-intensive systems as they support domain engineers in the specification of complex system behaviors and provide the basis for the performance of early analyses of such complex behaviors. The SLE community working on xDSMLs has made significant advances in the formalization of execution semantics yielding various approaches summarized in this paper under the term *executable metamodeling* [5]. But still, formalizing the execution semantics of xDSMLs is an inherently complex task requiring significant engineering effort. The prospective benefit of taking this effort consists in the automated generation of powerful execution infrastructures and analysis tools leading to reduced development costs for xDSMLs and increased quality of systems developed with xDSMLs.

Even though automating the development of execution infrastructures and analysis tools for xDSMLs has been the main driver behind research on executable metamodeling [3], it is today still a largely open research challenge. Much research is to be done for enabling the automated generation of different types of execution-based model analysis tools for xDSMLs taking into account the trade-off that has to be made between automation and customization towards the targeted xDSML.

In this paper, we revisit the tool generation challenge of xDSMLs brought forward by Bryant et al. [3] seven years ago critically reflecting on recent achievements towards addressing this challenge and pointing out open challenges.

The remainder of this paper is structured as follows. In Sect. 2 we formulate our vision on automating the development of xDSML tooling. Thereafter, in Sect. 3, we discuss open challenges towards achieving this vision. Finally, in Sect. 4 we conclude the paper with a summary.

2 The Vision of Automated Tool Development

In the current state of practice, model analysis tools for xDSMLs are still hand-crafted in an ad hoc manner leading to significant development costs. As a result, advanced model analysis tools are mostly only available for broadly adopted xDSML, such as UML [14] and Simulink [11].

As pointed out by Bryant et al. [3] in 2010, executable metamodeling has the potential to remedy this deficiency since formalizations of the execution semantics of xDSMLs establish the basis for automating the development of execution-based model analysis tools:

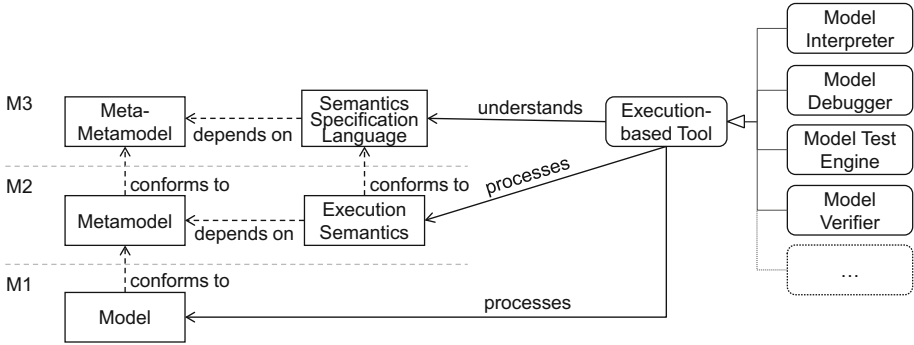


Fig. 1. Vision for automating the development of model execution tools

“However, many of these tools [model-based tools (e.g., editors, interpreters, debuggers, and simulators)] have a common semantic foundation centered around an underlying modeling language, which would make it possible to automate their development if the modeling language specification were formalized.” [3, p. 225]

Similarly as metamodeling languages laid the ground for automating the development of a variety tools that build upon the abstract syntax of a DSML, executable metamodeling approaches lay the ground for automating the development of powerful model execution infrastructures and execution-based model analysis tools. While this potential is the main driver of research on executable metamodeling, current approaches do not yet yield out-of-the-box analysis support for xDSMLs that is customized for the respective xDSML and ready-for-use by domain engineers. This significantly limits the value of executable metamodeling and xDSMLs in general.

Our vision for automating the development of execution-based modeling tools for xDSMLs is as follows:

Vision: Provide diverse execution-based analysis tools for xDSMLs out-of-the-box based on single formalizations of their execution semantics.

This vision is graphically illustrated in Fig. 1. Based on a single formalization of the execution semantics of an xDSML, various execution-based tools are obtained for the xDSML out-of-the-box that are ready-to-use by domain engineers. Such tools comprise, for instance, model interpreters, model debuggers, model test engines, and model verifiers. They rely on the single formalization of the execution semantics of the respective xDSMLs and can be used to analyze behavioral aspects of the modeled system.

3 Open Challenges Towards Tool Generation

The most significant advances towards the automated generation of analysis tools for xDSMLs have been achieved for the automated development of xDSML

debuggers. Existing approaches [1, 2, 4, 10, 15, 16, 19, 20] provide generic xDSML debuggers that utilize the execution semantics of xDSMLs for debugging models conforming to the respective xDSML. However, the possibilities to customize the generic debuggers to a specific xDSML are still quite limited and in most cases very specific to the employed executable metamodeling approach requiring adaptations of the execution semantics of the xDSML.

Furthermore, existing approaches that seek to automate the development of analysis tools for xDSMLs focus on providing support for one particular analysis technique only. Also, besides model debugging, hardly any other analysis technique has been investigated with respect to automating the development of tool support. Only few approaches exist that propose first solutions for automating the development of non-functional properties analysis tools [6], formal analysis tools [13, 18], and model testing tools [12]. In this regard, the ProMoBox framework proposed by Meyers et al. [12, 13] has to be explicitly highlighted as it proposes a solution for obtaining both model checkers and testing tools for xDSMLs.

To overcome the limitations of existing approaches and achieve the outlined vision of automating the development of model execution tools, several key challenges have to be addressed, which are discussed in more detail in the following.

(1) Identification and Formalization of Common Executability Concerns: The common foundations for performing execution-based model analyses are to be able to *interact* with an execution (e.g., provide input data to the model execution), *represent* an execution (e.g., represent changes in the execution state of a model), and *reason* about an execution (e.g., to reason about the correctness of a model's execution state evolution). To establish these foundations, it is necessary to make these concerns explicit in the definitions of xDSMLs, i.e., explicate them in an xDSML's execution semantics. However, in current executable metamodeling approaches, such information is mostly given only implicitly, deeply entangled in the definition of the execution semantics hindering its utilization for execution-based model analysis. For instance, while current approaches allow to explicitly define the initial input and final output of model executions, they do not allow to explicitly define possible interaction points with the model during execution, such as signals that can be sent to an event-driven model. Systematic engineering methods are required that support the explicit definition of interaction points and runtime information of executable models relevant for execution-based model analysis.

(2) Identification and Formalization of Analysis-Specific Executability Concerns: Besides foundational executability concerns common to execution-based model analysis, different kinds of analysis require information about additional analysis-specific executability concerns. For instance, to perform concurrency-aware analyses, the possible concurrency in the execution of models conforming to an xDSML has to be explicitly defined. xDSML engineering misses a systematic and comprehensive development methodology that supports language engineers in

identifying executability concerns relevant for a particular xDSML, and designing and implementing them in a modular and integrated way.

(3) *Automation of the Development of xDSML Analysis Tools:* Having all executability concerns of an xDSML readily implemented is of course only half of the way. The challenge of processing the defined executability concerns and providing different kinds of analysis tools well integrated with each other still remains. Integrating the provided analysis tools with each other is of major importance to reliably improve the quality of systems through the application of xDSMLs. For instance, it has to be investigated how model testing tools can be automatically integrated with model debuggers, such that faults detected with test cases can be efficiently localized.

(4) *Tool Customization:* The last challenge that we want to highlight is the need for customizing the generated tools for the respective xDSML. Customization towards the target xDSML and user is of uttermost importance to ensure the usefulness and effectiveness of the tools and is, hence, crucial for the realization of benefits from the application of xDSML. This customization has to be achieved on the interface between the derived tools and domain engineers, e.g., through languages provided to the domain engineers for defining analysis goals and interacting with the analysis tool. Significant advances for tool customization have been made for model debuggers (cf., for instance, [4]) but it is left open to investigate necessary and feasible customizations of other execution-based model analysis tools.

4 Conclusion

The value that is added by xDSMLs to the development of complex software-intensive systems is the capability to analyze behavioral properties of the systems early in the development process. To realize this value, xDSMLs have to be equipped with appropriate model analysis tools. Building such model analysis tools for xDSMLs manually and from scratch imposes significant engineering efforts and associated costs on the development process. Executable metamodeling seeks to overcome this deficiency by fostering the formalization of execution semantics of xDSMLs usable to automatically generate model execution infrastructures and model analysis tools. In this paper, we have discussed several key challenges that need to be addressed for fully achieving this vision of tool generation for xDSMLs.

Acknowledgement. This work is funded by the Austrian Agency for International Mobility and Cooperation in Education, Science and Research (OeAD) on behalf of the Federal Ministry for Science, Research and Economy (BMWF) under the grand number FR 08/2017, and by the French Ministries of Foreign Affairs and International Development (MAEDI) and the French Ministry of Education, Higher Education and Research (MENESR).

References

1. 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
2. Bousse, E., Corley, J., Combemale, B., Gray, J.G., Baudry, B.: Supporting efficient and advanced omniscient debugging for xDSMLs. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015), pp. 137–148. ACM (2015)
3. Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., Karsai, G.: Challenges and directions in formalizing the semantics of modeling languages. *Comput. Sci. Inf. Syst.* **8**(2), 225–253 (2011)
4. Chiş, A., Denker, M., Gîrba, T., Nierstrasz, O.: Practical domain-specific debuggers using the moldable debugger framework. *Comput. Lang. Syst. Struct.* **44**(PA), 89–113 (2015)
5. Combemale, B., Crégut, X., Garoche, P.-L., Thirioux, X.: Essay on semantics definition in MDE - an instrumented approach for model verification. *J. Softw.* **4**(9), 943–958 (2009)
6. Durán, F., Moreno-Delgado, A., Orejas, F., Zschaler, S.: Amalgamation of domain specific languages with behaviour. *J. Log. Algebr. Methods Program.* **86**(1), 208–235 (2017)
7. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pp. 471–480. ACM (2011)
8. Kleppe, A.: *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley, Boston (2008)
9. Kühne, T.: Matters of (meta-)modeling. *Softw. Syst. Model.* **5**(4), 369–385 (2006)
10. Lindeman, R.T., Kats, L.C., Visser, E.: Declaratively defining domain-specific language debuggers. In: Proceedings of the 10th International Conference on Generative Programming and Component Engineering (GPCE 2011), pp. 127–136. ACM (2011)
11. MathWorks. Simulink. <https://www.mathworks.com/products/simulink.html>
12. Meyers, B., Denil, J., Dávid, I., Vangheluwe, H.: Automated testing support for reactive domain-specific modelling languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016), pp. 181–194. ACM (2016)
13. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., Wimmer, M.: ProMoBox: a framework for generating domain-specific property languages. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 1–20. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_1
14. Object Management Group: *OMG Unified Modeling Language (OMG UML), Version 2.5*, September 2013. <http://www.omg.org/spec/UML/2.5>
15. Pavletic, D., Voelter, M., Raza, S.A., Kolb, B., Kehr, T.: Extensible debugger framework for extensible languages. In: de la Puente, J.A., Vardanega, T. (eds.) Ada-Europe 2015. LNCS, vol. 9111, pp. 33–49. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19584-1_3
16. Ráth, I., Vago, D., Varró, D.: Design-time simulation of domain-specific models by incremental pattern matching. In: Proceedings of the 2008 IEEE Symposium on

- Visual Languages and Human-Centric Computing (VL/HCC 2008), pp. 219–222. IEEE Computer Society (2008)
17. Schmidt, D.C.: Model-driven engineering. *IEEE Comput.* **39**, 25–31 (2006)
 18. Tikhonova, U.: Reusable specification templates for defining dynamic semantics of DSLs. *Softw. Syst. Model.* 1–30 (2017). <https://doi.org/10.1007/s10270-017-0590-0>
 19. Tikhonova, U., Manders, M., Boudewijns, R.: Visualization of formal specifications for understanding and debugging an industrial DSL. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) *STAF 2016*. LNCS, vol. 9946, pp. 179–195. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_13
 20. Wu, H., Gray, J., Mernik, M.: Grammar-driven generation of domain-specific language debuggers. *J. Softw.: Pract. Exp.* **38**(10), 1073–1103 (2008)

Toward Product Lines of Mathematical Models for Software Model Management

Zinovy Diskin¹(✉), Harald König², Mark Lawford¹, and Tom Maibaum¹

¹ McMaster University, Hamilton, Canada

{diskinz,lawford}@mcmaster.ca, tom@maibaum.org

² University of Applied Sciences FHDW Hannover, Hannover, Germany

harald.koenig@fhdw.de

Abstract. We present a general view on theoretical aspects of model synchronization and consistency management, and discuss technical challenges in making it sound, and cultural challenges in bringing it to practice.

1 Introduction: Why Product Lines

Software tool users would not welcome surprises in their tool's behaviour, e.g., in how the tool synchronizes the user's model with other interrelated models. To avoid surprises, the user should understand (a) what she really wants to do with her models, (b) what the tool can do, and (c) whether (b) matches (a). If the task is to place a screw into the wall, then examples of ideal matches are shown in Fig. 1(a), while Fig. 1(b) illustrates a possible mismatch. For mechanical tasks and tools, matching can be established by visual inspection, facilitated, perhaps, with a manual also referring to visual evidence. Unfortunately, software artifacts live in the conceptual rather than physical world and cannot be seen physically, and the story shown in Fig. 1(b) appears in software engineering practice more often than desired. For example, QVT's early non-success could be seen as an instance of the case (cf. [35]). For model management (MMT), the invisibility problem is intensified by the complexity and diversity of conceptual constructs and their interconnections used in the MMT tools because of the diversity of models and operations these tools should support. Navigating the design space becomes similar to navigating across unknown territory densely populated by invisible creatures connected by invisible links, so that seemingly harmless structural gaps actually hide deep chasms whose darkness obscures structural monsters.

The description above is, of course, an exaggeration (perhaps, not too strong) as the tool builders do have some "optical instruments" to observe the space—they can use operational descriptions to explain what the tool can do step by

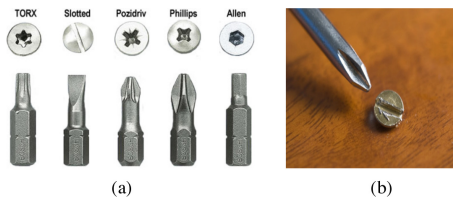


Fig. 1. A simple design space

step. Unfortunately, however, the optics of the operational semantics is too strong, and implementation details flooding the picture may essentially distort it. What we actually need is declarative semantics clearly presenting the basic features of the scenario at hand, and abstracting away all the rest. Such declarative semantics are, in fact, mathematical models, and besides “conceptual lenses” making the concepts visible, they normally bring one more benefit. Mathematical optics often allow us to see commonalities between seemingly different things, and discover essential differences between seemingly similar things (as evidenced by the history of mathematics and its application), thus packing the diversity into a firm manageable structural framework like feature models pack the diversity of products into an integral product line.¹ This aspect of mathematical modelling is exactly what is needed for building good maps of MMT design spaces, especially for synchronization encompassing an extreme diversity of different scenarios and their variations.

Indeed, in the early 2010s, Benjamin Pierce and his collaborators in the Harmony project on file synchronization introduced a simple algebraic model called a *lens* [17] (pun on our mathematical lenses above is unintended). Later, Pierce *et al.* published a series of papers uniformly titled “X lenses for Y” with (X=Resourceful, Y=String data) in [4], (Relational, Updatable Views) [5], (Quotient, \emptyset) [18], (Matching, Alignment and View Update) [3], (Symmetric, \emptyset) [21], (Edit, \emptyset) [22], where the empty string symbol \emptyset could be understood as “good for anything in synchronization”. (QVT’s analysis done by Stevens in [35] can also be classified in these terms as Lenses for Model Transformations.) All lenses above are state-based: model alignment necessary for synchronization is assumed to be reliably done by the synchronizer itself, which is a reasonable assumption for string data, but it does not always work for object alignment in the EMF world. To comply with the latter, a new family of lenses – *delta lenses* – was built by Diskin *et al.*, which extended the lens line even more [12, 13]. Even earlier, Johnson and Rosebrugh modelled updates by arrows in the respective categories of database/view states [25] and developed a theory of universal view updatability [26] within a framework that later, in the lens parlance, they termed as *categorical lenses* [28, 29].

As lenses are mathematical models, they lend themselves to classification and the respective structuring of the model synchronization design space. This premise was directly realized in [11] for a special synchronization case – bidirectional model transformation or bx. A general bx was modelled by an *organized* delta lens (od-lens), which is a delta lens with an additional structure called *org-symmetry*. The design space encompassed 44 different types of bx-scenarios (= types of od-lenses) over 10 types of the underlying computational framework (= types of delta lenses). That is, a tool implementing the specified family of delta lenses and capable of ensuring a simple org-symmetry discipline, would be able to handle 44 specified synchronization scenarios, each one with clearly

¹ For example, category theory seen as a discipline of mathematical modelling of mathematics itself, was very successful in packing the diversity of mathematical structures and operations over them into a product line of categories and categorical constructs.

specified input, output, and properties so that the situation shown in Fig. 1(b) can be avoided. Thus, although Pierce *et al.* coined the term *lens* based on quite technical reasons (arrows showing the two inverse operations in their diagrams, together form something recalling the lens shape), the lens algebraic framework did play the role of mathematical optics discussed above.

In the present paper, we discuss such a sort of activity for a bigger landscape of synchronization encompassing more than two models with the possibility of concurrent updates, and also the possibility (actually, the necessity!) of living with non-determinism and uncertainty in consistency restoration – each of these extensions dramatically changes the entire case of model synchronization and consistency management, both qualitatively (more dimensions/facets of the problem) and quantitatively (more types of scenarios). The taxonomy will go far beyond 44 cases, which brings us to the problem of indexing and navigating the space: linear indexing like in Fig. 1(a) works well for several types of screwdrivers, but is hardly functional for 44++ types. We will need to organize the space of mathematical models into a product line specified by a corresponding feature model—an idea that demonstrated its great effectiveness for software product lines and beyond, including facilitating communication between the stakeholders, planning, and management. This approach should be applicable to other MMT-operations, and we arrive at the title of the paper.

Our plan for the paper is as follows. In Sect. 2, we briefly revise several reasons for why abstract models providing declarative semantics of MMT-operations can be practically useful. Section 3 outlines a family of such model for bx, discusses their limitations for practical model synchronization scenarios and formulates four technical challenges. Finally, if even all technical challenges are successfully addressed, to ensure their practical implementation and usability, some methodological and cultural gaps are to be bridged – these are really Grand Challenges discussed in the last section.

2 Background: Why Abstract Models and Declarative Semantics

We discuss three interrelated reasons – one per subsection.

2.1 “And Suddenly the Tool Doesn’t Do Something Expected, and It Is a Nightmare for Them”

In a broad empirical study [23, 24, 39], now widely acknowledged and cited, Whittle *et al.* pointed to a number of factors that prevent wide MDE adoption in industry. As was further shown in [11, Sects. 2.3, 6.4], about a half of these factors amounts to different forms of (direct and indirect) miscommunication between the tool builders and the tool users such as (quoting the issues identified in the study) “Appropriating tools for purposes they were not designed for”, “Over-ambition: Asking too much of tools”, and “Engineers’ (mis)trust of tools”. As the authors of the study summarized these findings, “. . . *We do not*

have a fine-grained way of knowing which MDE tools are appropriate for which jobs.” However, perhaps a better way to convey the message is to cite immediately an MDE-practitioner interviewed in the study [39] – see the title of this subsection. This seems to be a precise identification of a core problem, whose most probable cause is that the tool builders do not have a suitable language for explaining *What* the tool is expected to do.

To explain *What*, we need a declarative specification framework for MMT operations, which is to be abstract enough to free specifications from unnecessary implementation details, yet be concrete enough to have essential properties of MMT tasks explicit in the specification.

2.2 Modelling Culture and Teaching It

One of the conclusions of Whittle’s *et al.* studies is that although tools could definitely be better, tools alone would not solve the problem: organizational and cultural aspects are also very important. Specifically, it is common to blame engineers for improper use of MDE tools and, wider, MDE ideas, and subsequently suggest their intensive training in tool application and proper MDE thinking. However, we would also emphasize the presence of cultural deficiencies on the opposite side of tool builders, who tend to create tools in an ad hoc manner based on weak (if any) semantic foundations. Reducing these deficiencies (say, “semantic training” for tool builders, and acquiring working habits of building tools based on clear specifications) would be also important for industrial MDE adoption. Moreover, tools based on ad hoc semantic foundations would hardly facilitate proper MDE thinking on the user side, while intelligent and mathematically supported tooling would itself promote the right MDE attitude into the user community. Both sides need teaching and training, and a clear, well-structured and well-organized theory is a cornerstone of effective teaching of software technologies.

2.3 Completeness of Classifications/Design Spaces

A proper and sufficiently complete classification of the design space is crucially important for *normal vs. radical* engineering as discussed by Vincenti [38]. Normal design considers a new product as a variant of a well understood class of products, and applies known and accepted design patterns rather than inventing them from scratch. A proper classification of the problem at hand is the very first, and maybe the most important, step. If the problem is properly identified, the engineer can use her design cookbooks and follow the known recipes and patterns with a high probability of successful design. Software engineering, despite all its important distinctions from mechanical and electrical engineering, is still a branch of engineering, and hence Vincenti’s findings are applicable to it as well (see [32] for a more detailed discussion).

3 Model Synchronization and Its Challenges

A typical MDE environment consists of several clusters of heterogeneous inter-related models, which we will call *multimodels*. Normally, all models in a multimodel should be mutually consistent and represent different but coherent views of the system. If one or several models in a multimodel change, other models should be changed accordingly to restore consistency. In other words, changes should be propagated to other models so that the new state of the multimodel is consistent again. Clearly, consistency maintenance is of paramount importance for MDE technologies, but a theoretical analysis of even simple scenarios is difficult and a sound practical implementation is challenging [14,36,37].

An important special case of consistency maintenance is when the multimodel consists of two models connected by a transformation, for example, a UML model (which is itself a multimodel, but we consider it as an integrated whole) and the Java code it generates, or an Object-Relational Mapping translating class diagrams to relational tables. An abstract mathematical model of the binary case was developed under the name of *organized delta lenses* (*od-lenses* in short), and below we will sketch basic ideas. We will begin with a primer on (symmetric) delta lenses, and then define od-lenses. After that, we will discuss several challenges of building mathematical models for consistency management beyond the current form of od-lenses.

3.1 A Sketch of (Symmetric) Delta Lenses

This section is mainly based on papers [8,9,11], but the material is structured differently and includes new elements. Specifically, we identify a new common schema for different variants of the basic algebraic laws regulating lenses' behaviour (Putput and Invertibility), in which they have *strong*, *weak*, and *mediated* versions. Some parts of this schema are justified by examples considered in [9,11], while others are mainly guesses to be checked against practical examples and an accurate formalization.

Operations. Suppose we have two model spaces, \mathbf{A} and \mathbf{B} , e.g., in the examples above, \mathbf{A} encompasses some class of UML models, and \mathbf{B} a class of Java programs generated from them, or \mathbf{A} is a class of UML class diagrams and \mathbf{B} is a class of relational schemas. A pair (A, B) of interrelated models normally comes with a traceability mapping so that our typical situation to consider is a triple (A, r, B) with A a model from \mathbf{A} , B a model from \mathbf{B} , and r some specification of correspondences between them, which we will denote by a bidirectional arrow (the top one in Fig. 2) and call a *corr*. There may be many different corrs between the same A and B , but not all of them satisfy some a priori given notion of consistency.

If model A has changed, which is specified by a vertical update arrow u (we will also say delta) in the upper square in Fig. 2, then consistency can be destroyed (imagine an inconsistent corr between models A' and B) and should be restored by an appropriate change v on the \mathbf{B} -side so that the new corr r' is consistent. Algebraically, we have an operation that takes a consistent corr r and an update u as its input (in Fig. 2, these are shown with framed nodes and solid arrows), and produces an update v and a new consistent corr r' at the output (non-framed node and dashed arrows). We call this operation *forward update propagation* and denote it by \mathbf{fPpg} . Similarly, there is an operation \mathbf{bPpg} of *backward update propagation* (the lower square in Fig. 2). Importantly, when we propagate update u to the B -side, changes in model B should be minimal in some sense (see [6] for a discussion of this non-trivial issue). Specifically, if models B have private (non-shared) data, it is natural to require these data to be preserved in B' . For example, consider a UML tool that generates code stubs from class diagrams. When a parameter is added to a method signature, code must be regenerated in such a way that the signature is updated, but the method body is preserved. That is, while code's public parts (method signatures, class names, etc.) are updated to reflect changes in the UML model, code's private data—the method bodies—are kept untouched.

Note that, similarly to corr arrows, an update arrow $u : A \rightarrow A'$ is not just a pair (A, A') but a specification of correspondences between states A and A' , or an edit log/sequence that changes A to A' . The formalism does not prescribe what updates are, but its important underlying assumption is that normally there may be different updates from state A to state A' .

We require updates to be composable: for any $u : A \rightarrow A'$ and $u' : A' \rightarrow A''$, there a unique *composed* update $u'' : A \rightarrow A''$ denoted by $u; u'$. Moreover, for any state A , we require the existence of a special *idle* update $\text{id}_A : A \rightarrow A$ that does nothing and, hence, $u^-; \text{id}_A = u^-$ for any update u^- into A , and $\text{id}_A; u^+ = u^+$ for any u^+ from A . These requirement make model space \mathbf{A} a category whose objects/nodes are models, morphisms/arrows are updates/deltas, and identity arrows are idle updates. We will write $|\mathbf{A}|$ for the class of all models in \mathbf{A} , and \mathbf{A}^Δ or $\Delta_{\mathbf{A}}$ for the class of all deltas, i.e., all arrows in \mathbf{A} . Space \mathbf{B} is also a category with objects $|\mathbf{B}|$ and arrows \mathbf{B}^Δ or $\Delta_{\mathbf{B}}$.

The propagation operations should satisfy three groups of algebraic laws described below, and then we say that the tuple $\ell = (\mathbf{A}, \mathbf{B}, \mathbf{fPpg}, \mathbf{bPpg})$ is a (*well-behaved symmetric*) *delta lens* or just a *lens*, and write $\ell : \mathbf{A} \rightleftharpoons \mathbf{B}$.

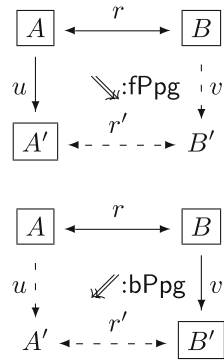


Fig. 2. Lens operations

Algebraic Laws 1: Stability and Privacy. The first group of laws concerns idle updates. A simple but important requirement is that any idle update on either side is propagated into a respective idle update on the other side with $r' = r$. In other words, the system does nothing by itself, and this law is often called *Stability*. However, there may be non-idle updates propagatable into idle updates on the other side. For example, some changes in code are not anyhow reflected in the UML model used to generate the code, and changing the layout of a UML diagram does not affect the code. Such updates are called *private* – indeed, the other side does not see them, and non-private updates are called *public*. Thus, the set of all updates on either side is partitioned into private and public, $\Delta_{\mathbf{A}} = \Delta_{\mathbf{A}}^{\text{prv}} \cup \Delta_{\mathbf{A}}^{\text{pub}}$, and $\Delta_{\mathbf{B}} = \Delta_{\mathbf{B}}^{\text{prv}} \cup \Delta_{\mathbf{B}}^{\text{pub}}$, and Stability says that idle updates are always private.

Algebraic Laws 2: PutPut. The second group of laws (called Putput in the lens jargon) considers compatibility of (horizontal) update propagation with (vertical) update composition. It is the most controversial part of the lens story: such compatibility is very desirable mathematically (as a necessary “lubricant” for categorical machineries), but rarely holds in practice in the following strong sense. Given a corr $r : A \leftrightarrow B$ and two consecutive updates on the \mathbf{A} -side, $u : A \rightarrow A'$, $u' : A' \rightarrow A''$ (see Fig. 3), we can build two updates on the \mathbf{B} -side: one is the composition of two propagations, $v'' = v; v'$ with $v = \text{fPpg}(r, u)$, $v' = \text{fPpg}(r', u')$, and the other is the propagation of the composition, $v_{\text{jump}} = \text{fPpg}(r, u; u')$. The *strong Putput* states that $v'' = v_{\text{jump}}$, which is a standard categorical condition of functoriality, and it does hold in practice when updates u and u' do not conflict, i.e., if u' does not affect elements affected by u , e.g., if both u and u' are insertions, or both are deletions (the *monotonic Putput* of Johnson and Rosebrugh [27]). For the symmetric situation, strong Putput is also valid if updates conflict, but privately, so that it is not seen on the other side. Specifically, it is the case when at least one of the updates u, u' is private. But beyond these special cases, strong Putput rarely holds in practice (which was emphasized by several authors [4, 12, 35], where demonstrating examples can be found). Thus, to make our model applicable to practice, we can only require for delta lenses *restricted strong Putput*, i.e., Putput only holding for several specified types of updates.

However, even when strong Putput is violated, the difference between $v'' = v; v'$ and v_{jump} should normally be private and not visible on the other side. We

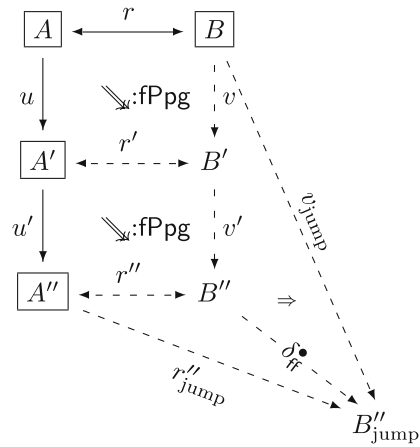


Fig. 3. Mediated Putput for fPpg

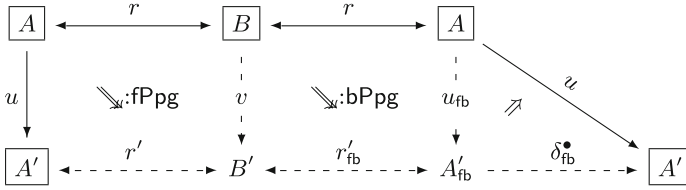


Fig. 4. Mediated invertibility/round-tripping (forward-backward, fb)

thus come to the *weak* Putput law: $\text{bPpg}(r, v'') = \text{bPpg}(r, v_{\text{jump}})$. Weak Putput has good chances of holding in practical cases, but is not very useful as it does not say much about how v'' and v_{jump} are interrelated. A better idea would be to specify the privacy of the interaction between the two updates by stating the existence of a unique private update $\delta_{\text{ff}}^{\bullet}(r, u, u')$ or just $\delta_{\text{ff}}^{\bullet} : B'' \rightarrow B''_{\text{jump}}$ such that $v''; \delta_{\text{ff}}^{\bullet} = v_{\text{jump}}$ (and the symbol \Rightarrow in Fig. 3 can be interpreted as denoting commutativity of the triangle; below we will refine this interpretation). We will call the update $\delta_{\text{ff}}^{\bullet}$ *mediating*, and the law above *mediated Putput*. Note that mediated Putput together with restricted strong Putput, imply (unrestricted) weak Putput as the mediating delta is private.

Mediated Putput is a good compromise between too restrictive strong Putput and too general weak Putput. Unfortunately, in a wide class of practically interesting cases, mediated Putput also fails as commutativity $v''; \delta_{\text{ff}}^{\bullet} = v_{\text{jump}}$ does not hold (see [9] for examples). We will return to this issue below in Sect. 3.3 (C).

Strong, weak and mediated Putput laws for the backward propagation bPpg are obvious mirror reflections of the fPpg versions.

Algebraic Laws 3: Invertibility. Invertibility (or round-tripping) laws regulate how operations fPpg and bPpg interact. The diagram in Fig. 4 specifies a simple scenario, in which an update u on the left side is propagated to the right, and then propagated back with the result denoted by u_{fb} . Similarly to Putput, there are three types of **fb**-invertibility. *Strong* invertibility states that $u_{\text{fb}} = u$, but it rarely holds in practice [13]. *Weak* invertibility states that $\text{fPpg}(r, u_{\text{fb}}) = \text{fPpg}(r, u)$, i.e., the difference between u_{fb} and u is not visible on the other side. Like weak Putput, weak invertibility is much less restrictive for practical cases, but is not very useful as it does not say how u_{fb} and u are related. The issue seems to be mitigated with *mediated* invertibility, which requires for any pair r, u as shown in Fig. 4, a uniquely determined private update $\delta_{\text{fb}}^{\bullet}(r, u)$ or just $\delta_{\text{fb}}^{\bullet}$ such that $u_{\text{fb}}; \delta_{\text{fb}}^{\bullet} = u$ (and double-arrow \nRightarrow again refers to commutativity of the triangle). Together with restricted strong Putput, it implies weak invertibility as the mediating delta is private. Unfortunately, the loss of information during propagation breaks this nice schema again: even mediated invertibility fails in a wide class of practically interesting cases (examples can be built along the lines of those in [9] for Putput). We will return to the problem in Sect. 3.3 (C). The backward-forward (bf) round-tripping law is the mirror image of the fb-law.

Asymmetric Lenses. An important class of lenses is formed by the so called (*info-*)*asymmetric* lenses. Given a lens $\ell: \mathbf{A} \rightleftharpoons \mathbf{B}$, we say that space \mathbf{A} is *less informative* (or *more abstract*) than \mathbf{B} w.r.t. ℓ if all \mathbf{A} -models do not have private data, and hence the only private updates on the \mathbf{A} -side are identities. Then we call the lens *asymmetric* and write $\mathbf{A} \leq_{\ell} \mathbf{B}$. These conditions are typical for the case when \mathbf{A} -models are (*abstract*) *views* of \mathbf{B} -models, and so asymmetric lenses model updatable views. The absence of private data on the \mathbf{A} -side allows us to simplify operation \mathbf{bPpg} (see Fig. 2(b)): it becomes an ordinary functor between model categories and is typically denoted by $\mathbf{get}: \mathbf{A} \leftarrow \mathbf{B}$ (read *get the view*)—indeed, the original state of the \mathbf{A} -model (view) is not needed anymore as views do not have private data (see [8] for details). In contrast, the forward propagation \mathbf{fPpg} from the \mathbf{A} -side to the \mathbf{B} -side does need the original state of the \mathbf{B} -model as shown in Fig. 2(a), but for the asymmetric lenses, this operation is typically denoted by \mathbf{put} (read *put the view update back*). We believe that short energetic names \mathbf{get} and \mathbf{put} for the operations, and the respective names \mathbf{PutGet} , \mathbf{GetPut} , and \mathbf{PutPut} for the laws (all coined by Pierce²), did make an essential contribution to the popularity of the lens framework.

If the \mathbf{B} -side also does not have private data w.r.t. lens ℓ , we write $\mathbf{A} \approx_{\ell} \mathbf{B}$ and call the lens (*info-*)*bijective*, while *strictly asymmetric* case is denoted by $\mathbf{A} <_{\ell} \mathbf{B}$. Thus, there are three disjoint types of lenses depending on whether exactly one, two, or neither side has private data, and writing $\mathbf{A} \leq_{\ell} \mathbf{B}$ means that either $\mathbf{A} <_{\ell} \mathbf{B}$ or $\mathbf{A} \approx_{\ell} \mathbf{B}$. When both sides have private data, the lens is called (*info-*) *symmetric* and we write $\mathbf{A} \times_{\ell} \mathbf{B}$.

3.2 Organizational Symmetry and Organized Delta Lenses

In some synchronization scenarios, all updates on either side are propagated to the other, e.g., in the roundtripped UML-Java code generation, changes are freely propagated from the model to code and back. Then we say that the two sides of synchronization are *organizationally equivalent* or *org-symmetric*. In a less symmetric situation, the synchronization policy may only allow some code updates (e.g., changes in method signatures) to be propagated to the UML model, whereas other code updates that violate consistency will be discarded. We may term this as *org-semi-symmetry*. And in (*strictly*) *org-asymmetric* cases, one side entirely dominates the other and updates from the latter are not propagatable at all. For example, the synchronization policy can prohibit any public updates in the code (but allow private code updates like comments, or, say, method bodies if the tool only generates code stubs). Model compilation presents an extreme case, when no code updates are allowed at all.

² As per personal communication with Nate Foster.

To model the above, we define the notion of an *organized lens*, *od-lens*.³ It is a triple (ℓ, P, Q) with $\ell: \mathbf{A} \rightleftharpoons \mathbf{B}$ a lens, $P \subset \Delta_{\mathbf{A}}^{\text{pub}}$ a subset of public updates called *propagatable A-updates* and $Q \subset \Delta_{\mathbf{B}}^{\text{pub}}$ a subset of *propagatable B-updates*. In fact, the pair (P, Q) is nothing but what we called above a synchronization policy.

Importantly, sets P and Q constituting a policy are determined organizationally rather than technologically, in the sense that propagation operations can be defined for non-propagatable deltas. For example, when code is generated from a UML model by a forward operation fPpg , the backward operation bPpg is defined for all code deltas, and is important for checking correctness of code generated from the model w.r.t. its conformance to the model as prescribed by the invertibility law. However, only some (or none) of code deltas are allowed to be propagated back to the model.

There are three special cases for the set P of propagatable updates: $P = \emptyset$, $P = \Delta_{\mathbf{A}}^{\text{pub}}$ and $P \subsetneq \Delta_{\mathbf{A}}^{\text{pub}}$, and similarly for Q . Different combinations of these factors define several types of organizational (a)symmetry. As shown in [11, Sect. 5.2], there are 14 species of od-lenses, i.e., 14 types of bx-scenarios, all obtained by different combination of info- and org-symmetries. Actually the number of scenarios covered by the taxonomy in [11] is even more, 44 as mentioned in the introduction, because yet another dimension of synchronization was considered: whether after a change on one side, the model on the other side is regenerated from scratch (batch-update) or incrementally. We again have different symmetries for this dimension depending on whether both, one, or neither direction of update propagation is incremental.

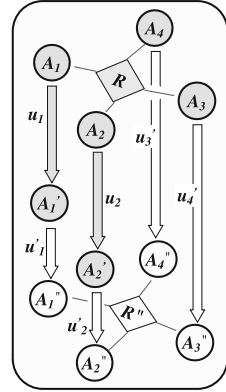
3.3 Four Technical Challenges

Our final goal with developing a mathematical framework for model synchronization is to make the tool-user vs. the tool-builder communication effective. In this section, we discuss four issues that we think are important for this task.

(A) Multi-directionality and Concurrency. *Multidirectionality* refers to synchronization of several models via multidirectional update propagation so that when one or several models change and the global consistency is violated, changes should be propagated to other models to restore consistency. Formally, we have a family of model spaces $(\mathbf{A}_i, i \in I)$ with a finite indexing set I of cardinality $n = |I|$, which are somehow inter-related by a set R of n -ry correspondence specifications, i.e., for any $r \in R$, its boundary models $\partial_i(r) \in |\mathbf{A}_i|$ are defined.

³ The notion of an organized lens was introduced in [11], although the very name *od-lens* is new.

For any $J \subseteq I$, we have an update propagating operation ppg^J that takes a multiary consistent corr r with boundaries $A_i = \partial_i(r)$, $i \in I$, and updates $(u_j : A_j \rightarrow A'_j, j \in J)$, and returns two families of updates: (i) *propagated* updates, $(u'_i : A_i \rightarrow A''_i, i \in I \setminus J)$, and (ii) *reflected* updates, $(u''_j : A'_j \rightarrow A''_j, j \in J)$, together with a new consistent corr r'' such that $\partial_i(r'') = A''_i$, $i \in I$. The inset figure illustrates this for $I = \{1..4\}$ and $J = \{1, 2\}$, the input elements for the operation are shaded, the output ones are blank.



We have concurrency (managing parallel updates) when cardinality $|J| \geq 2$, which makes sense even for the binary multimodel with $n = 2$, but it is the multiary case $n > 2$ when excluding concurrency looks unrealistic (the more so the bigger n is). For the binary case, concurrency means $J = I$, and there are two variants of the story. One is when two parallel updates do not conflict, and hence can be *merged* as described in [33]. The other variant is much more difficult to manage: if two updates overlap and conflict (e.g., one update changes an attribute of a class deleted by the other update), then consistency restoration needs a conflict resolution strategy and a partial rolling back of one or both updates. It is still an open problem even for the binary case (see [33] for discussion). For the multiary situation, the two cases above correspond to $J = I$ and obviously show more complexity. Moreover, for the multiary situation we also have the possibility of $|J| \geq 2$ but $J \subsetneq I$, which further branches into consistency restoration with no rolling back (all reflected updates being idle, i.e., $u'_1 = \text{id}_{A'_1}$, $u'_2 = \text{id}_{A'_2}$ for the inset figure), and with partial rolling back (at least one of $u'_{1,2}$ is not idle). The former of these two scenarios is much simpler but even it shows much more possibilities for consistency restoration than in the binary case: e.g., for the inset figure, we can look for u'_3 assuming $u'_4 = \text{id}_{A_4}$, or swap their roles, or consider both $u'_{3,4}$ to be non-trivial. Similarly, there are different choices for conflict resolution and partial rolling back strategies. Overall, the multitude of synchronization types for the multiary situation is much bigger than for the binary case, and its accurate mathematical modelling and the respective classification of possible scenarios, and arranging them into a navigable and manageable structure (a product line), are challenging.

(B) Uncertainty. The loss of information and its recovery are inherent in, resp., view computation and view update propagation, and make the latter inherently non-deterministic and uncertain. As a rule, consistency can be restored in many different ways and thus update propagation operations are, in general, multi-valued. Update propagation policies can narrow the multitude of solutions, but normally do not eliminate it entirely. To manage the problem, Eramo *et al.* proposed to specify the multitude by a suitable variability model, and give the user an option to choose a preferred solution [16]. A practical disadvantage of this approach is that the synchronizer has to wait until the user decides; moreover,

at the time of taking the decision, the user may not have all the necessary information and would prefer to postpone the decision for some time later. A better approach would be to encode the multitude of solutions as one uncertain/incomplete model (whose multitude of completions would be exactly the multitude of possible consistency restoration solutions) and proceed with other model management procedures. In other words, model management operations should be extended to work smoothly with uncertain/incomplete models. Including uncertain models into the realm of model management heavily complicates the mathematics as the first attempt to manage it in [10] shows (cf. also an attempt in this direction but in another context in [34]). The database community is currently dealing with a version of this problem in the context of querying data with Nulls, in fact, computing view $A = \text{get}(B)$ for a database B with uncertain data, which is qualified as a challenging problem [31]. The MDE context is even more complicated as not only attribute values in a model can be uncertain, but existence of objects and links can be optional too.

(C) Laxity. The laws of compositionality (Putput) and invertibility of update propagation are fundamental for model synchronization, but their understanding and management in the current theory are disappointing. As in both vertical composition of updates and horizontal model transformation, information is typically lost, strong Putput and strong Invertibility only hold for special updates and special corrs. The weak forms of these laws hold in general, but are not very interesting and useful: they say that two arrows should be equivalent in some sense, but do not actually relate them. The mediated form of the laws seems to be a good compromise, but unfortunately also often fails because the key commutativity requirement for the triangles in Figs. 3 and 4 fails in many practically interesting cases [9]. To fix the problem, we need to equip the mediating delta between models δ_{ff}^\bullet (see Fig. 3) with a mediating delta between updates, $\delta_{ff}^\Delta: v; v'; \delta_{ff}^\bullet \Rightarrow v_{\text{jump}}$, i.e., *2-arrow/2-delta* (note the double arrow near B'' in Fig. 3), which specifies the difference between updates. Similarly, to fix mediated invertibility, we require the uniquely determined 2-delta $\delta_{fb}^\Delta: u_{fb}; \delta_{ff}^\bullet \Rightarrow u$ shown in Fig. 4. The mirror images of these constructs are required for the backward Putput and bf-invertibility.

2-deltas between updates actually have a clear physical meaning. If we think of updates as sets of links, then 2-deltas amount to subset relations between these sets; if we think of updates as edit logs, i.e., sequences of elementary edit operations, then 2-deltas amount to sub-sequence relations between logs. Speaking categorically, adding 2-arrows to the formalism means that we consider model spaces \mathbf{A}, \mathbf{B} to be 2-categories. Adding this new piece of structure to the construct of model space implies the respective changes in the other elements of the structure: update composition, update propagation, the notion of privacy (we need *private 2-deltas* which are propagated to 2-identities, and important for proving equality between updates), etc. That is, we need to rebuild the entire delta lens framework on the premises of 2-category theory. In the latter, “equality” of two arrows up to a mediating 2-arrow is generally called *lax*: we

relax strong commutativity in the mediated form of the basic laws and replace it by lax commutativity as explained above. Laxity thus appears as an appropriate mathematical approach to deal with lost information and uncertainty inherent in model synchronization. Lax mediated laws as outlined above should be a proper trade-off between practical satisfiability and manageable mathematical patterns.

(D) Variability (mega) management. Even a relatively simple two-model synchronization without concurrency and uncertainty amounts to 44 different types of synchronization scenarios [11]. Extending the framework to meet the three challenges above would greatly expand the multitude and diversity of mathematical models we need to manage. Besides purely mathematical aspects, this management should include ways of observing the design space and navigating through it for understanding, communicating and organizing collaborative research and development. Dealing with these issues in an ad hoc way would hardly be effective, and something more technological is needed.

One side of the issue is about how to build this structure of structures, and it is a typically categorical story: we need to specify morphisms between structures of a certain type and organize them into a category, then organize the multitude of categories into a “mega” category and so on. The approach is clear but its realization for a deep chain of categories of categories of . . . could be challenging. In addition to abstract declarative models we discussed so far, the area needs models bridging the gap between declarative and operational semantics and finally implementation (we will return to this in the Conclusion).

The other side of the story is about the usability of the megastructure as outlined above, and it is natural to borrow the ideas of feature modelling and variability management developed in the area of software product lines. Indeed, feature models provide a convenient way of navigating through a big design space and hence facilitate communication between the customer (the tool user in our context) and the vendor (the tool builder); they also help to organize the production and management, which is also a concern in our setting. Integrating the abstract and seemingly cryptic categorical, and the concrete and seemingly transparent product line sides of the story is not obvious, but seems to be a reasonable way to go. Specifically, interpreting metamodels as feature models can be an important step in this direction.

4 Conclusion: Grand Challenges

The story we are telling in the paper is actually not new. Many years ago, people like Dijkstra noticed that considering the space of all programs leads to anarchy, which kills engineering, or perhaps does not allow it to be born. They proposed restricting the class of programs to be considered so that a restricted space would allow its appropriate structuring, and then effective engineering methods based on mathematics could be developed. This idea turned out to be very successful, but there is an important distinction between their and our situations. Mathematics used for structuring and engineering program spaces was

based on ordinary logic and algebra, and was familiar to an average programmer of that time (often a physicist or a mathematician). Hence, bridging the inevitable distance between high-flying methodologists and earth-bound practitioners was a manageable problem, and it was well managed indeed. In contrast, mathematics for MMT is inherently graph-based and diagrammatic, i.e., in fact, categorical and hence quite unfamiliar to an average software developer of our time. Moreover, the categorical motto of thinking formally *with* diagrams, i.e., both diagrammatically and precisely, contradicts to the typical mathematical background of a modern software developer or a computer scientist, for whom precise reasoning and diagrammatic reasoning are mutually exclusive options.

Two manifestations of this general phenomenon are important in our context. One is about a culture of building and using metamodels in MDE. As exercising precise reasoning based on ordinary mathematics on the scale of artifacts to be managed in modern MDE would be unmanageable, precision is sacrificed for manageability—a long tradition in software engineering [19] freshly instantiated in MDE with an abundance of metamodels not only missing important constraints, but with pieces of structure that are distorted or missing altogether. The discrepancy between an explicitly declared metamodel and the instances manipulated by the tool can be significant, which means that a lot of semantic information is hidden (actually buried) in the low level code rather than being explicated in models and metamodels. On the other hand, many people dealing with synchronization and lenses have a functional programming background and several academic proof-of-concept tools were created. Moreover, state-based lenses became very popular in the Haskell community after Edward Kmett wrote a library of lenses in Haskell and successfully used them in business applications⁴. However, precision is of utmost importance for the FP community, which thus tends to sacrifice deltas and switch to non-diagrammatic state-based lenses⁵—the deficiencies of this approach are discussed in [12, 13].

Much smoother is interaction between delta lenses and rule-based languages, which appear to be a common way of thinking about model management operations, and, when talking about bx, approaches based on TGG turned out to be especially appropriate [1, 20]. However, TGG are designed for the binary synchronization case, and it may be challenging to extend them for the multiary situation (but see a promising attempt in [30]). It may also be difficult to bridge the gap in mathematical modelling attitudes: abstract algebra (inherently declarative) on the one side and operational rule-based engineering on the other.

Thus, the barriers between the mathematical framework for synchronization based on explicit deltas and its practical implementation within the modern coding frameworks are of a cultural nature and hence not easy to overcome. Indeed, an effective solution to cultural problems is usually found on evolutionary, rather than revolutionary, paths with significant investment of time and efforts. An important step in this direction would be in keeping close interaction between

⁴ <https://github.com/ekmett/lens>.

⁵ Even though FP proponents are familiar with monads and other categorical constructs employed in FP.

potential tool builders and future tool users right from the beginning. Model management scenarios should jointly be discussed by all participating stakeholders, including a mathematical expert able to exercise a cultural bx: transforming MMT notions into mathematical constructs, and support the backward transformation with simple examples and easy-to-follow diagrams.⁶ The decisions about the scope of the tool and its underlying operational framework (algebraic, rule-based, or other) should be guided by these joint discussions rather than solely by the tool builder. Then, perhaps, in some future time not so far away, the statement in the title of Sect. 2.1 will no longer dominate the MDE practice!

Acknowledgement. The idea to write the paper in its current form emerged at the Grand Workshop that provided a full day of provocative presentations, stimulating discussions, and creative overall atmosphere. We are grateful to all participants and the organizers for creating this inspiring event.

References

1. Anjorin, A., Diskin, Z., Jouault, F., Ko, H., Leblebici, E., Westfechtel, B.: Benchmark reloaded: a practical benchmark framework for bidirectional transformations. In: Eramo and Johnson [15], pp. 15–30
2. Anjorin, A., Gibbons, J., (eds.) Proceedings of the 5th International Workshop on Bidirectional Transformations, BX 2016, Co-located with the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, 8 April 2016, CEUR Workshop Proceedings, vol. 1571. CEUR-WS.org (2016)
3. Barbosa, D.M.J., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: ICFP 2010, pp. 193–204 (2010)
4. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: POPL 2008, pp. 407–419 (2008)
5. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: PODS 2006, pp. 338–347 (2006)
6. Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: On principles of least change and least surprise for bidirectional transformations. *J. Object Technol.* **16**(1), 3:1–31 (2017)
7. Coecke, B., Kissinger, A.: *Picturing Quantum Processes. A First Course in Quantum Theory and Diagrammatic Reasoning.* Cambridge University Press, Cambridge (2017)
8. Diskin, Z.: An algebraic semantics for bidirectional model synchronisation. University of Waterloo, Technical report GSDLab-TR 2014–04-01 (2014)
9. Diskin, Z.: Compositionality of update propagation: lax putput. In: Eramo and Johnson [15], pp. 74–89
10. Diskin, Z., Eramo, R., Pierantonio, A., Czarnecki, K.: Incorporating uncertainty into bidirectional model transformations and their delta-lens formalization. In: Anjorin and Gibbons [2], pp. 15–31
11. Diskin, Z., Gholizadeh, H., Wider, A., Czarnecki, K.: A three-dimensional taxonomy for bidirectional model synchronization. *J. Syst. Softw.* **111**, 298–322 (2016)

⁶ Indeed, even quantum phenomena can be specified in a picturesque way as demonstrated in Coecke and Kissinger’s book [7].

12. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. *J. Object Technol.* **10**(6), 1–25 (2011)
13. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state-to delta-based bidirectional model transformations: the symmetric case. In: Whittle, J., Clark, T., Kühne, T. (eds.) *MODELS 2011*. LNCS, vol. 6981, pp. 304–318. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_22
14. Eged, A.: Fixing inconsistencies in UML design models. In: *ICSE 2007*, pp. 292–301 (2007)
15. Eramo, R., Johnson, M., (eds.) *Proceedings of the 6th International Workshop on Bidirectional Transformations co-located with the European Joint Conferences on Theory and Practice of Software, BX@ETAPS 2017, Uppsala, Sweden, 29 April 2017, CEUR Workshop Proceedings*, vol. 1827. CEUR-WS.org (2017)
16. Eramo, R., Pierantonio, A., Rosa, G.: Managing uncertainty in bidirectional model transformations. In: *SLE 2015*, pp. 49–58 (2015)
17. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In: *POPL 2005*, pp. 233–246 (2005)
18. Foster, J.N., Pilkiewicz, A., Pierce, B.C.: Quotient lenses. In: *ICFP 2008*, pp. 383–396 (2008)
19. Haeberer, A.M., Maibaum, T.S.E.: Scientific rigour, an answer to a pragmatic question: a linguistic framework for software engineering. In: *ICSE 2001*, pp. 463–472 (2001)
20. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., Engel, T.: Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Softw. Syst. Model.* **14**(1), 241–269 (2015)
21. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: *POPL 2011*, pp. 371–384 (2011)
22. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: *POPL 2012*, pp. 495–508 (2012)
23. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: *ICSE 2011*, pp. 633–642. IEEE, ACM (2011)
24. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: *ICSE 2011*, pp. 471–480. IEEE, ACM (2011)
25. Johnson, M., Rosebrugh, R.D.: View updatability based on the models of a formal specification. In: *FME 2001: Formal Methods for Increasing Software Productivity*, pp. 534–549 (2001)
26. Johnson, M., Rosebrugh, R.D.: Fibrations and universal view updatability. *Theor. Comput. Sci.* **388**(1–3), 109–129 (2007)
27. Johnson, M., Rosebrugh, R.D.: Lens put-put laws: monotonic and mixed. *ECEASST*, **49** (2012)
28. Johnson, M., Rosebrugh, R.D.: Delta lenses and opfibrations. *ECEASST*, **57** (2013)
29. Johnson, M., Rosebrugh, R.D.: Unifying set-based, delta-based and edit-based lenses. In: Anjorin and Gibbons [2], pp. 1–13
30. Königs, A., Schürr, A.: MDI: a rule-based multi-document and tool integration approach. *Softw. Syst. Model.* **5**(4), 349–368 (2006)
31. Libkin, L.: Certain answers as objects and knowledge. *Artif. Intell.* **232**, 1–19 (2016)
32. Maibaum, T.S.E.: What we teach software engineers in the university: do we take engineering seriously? In: *ESEC/FSE*, pp. 40–50 (1997)
33. Orejas, F., Boronat, A., Ehrig, H., Hermann, F., Schölzel, H.: On propagation-based concurrent model synchronization. *ECEASST*, **57** (2013)

34. Salay, R., Famelis, M., Rubin, J., Sandro, A.D., Chechik, M.: Lifting model transformations to product lines. In: ICSE 2014 (2014)
35. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. *Softw. Syst. Model.* **9**(1), 7–20 (2010)
36. Stevens, P.: Bidirectional transformations in the large. In: MODELS (2017 to appear)
37. Taentzer, G., Ohrndorf, M., Lamo, Y., Rutle, A.: Change-preserving model repair. In: FASE 2017, pp. 283–299 (2017)
38. Vincenti, W.: What engineers know and how they know it: analytical studies from aeronautical history. *Johns Hopkins Studies in the History of Technology* (1993)
39. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial adoption of model-driven engineering: are the tools really the problem? In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 1–17. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_1

Model-Driven Robot Software Engineering (MORSE)

Model-Driven Interaction Design for Social Robots

Gary Cornelius^(✉), Nico Hochgeschwender, and Holger Voos

Interdisciplinary Centre for Security, Reliability and Trust,
University of Luxembourg, Luxembourg, Luxembourg
{gary.cornelius,nico.hochgeschwender,holger.voos}@uni.lu

Abstract. Robotic software development frameworks lack a possibility to present, validate and generate qualitative complex human robot interactions and robot developers are mostly left with unclear informal project specifications. The development of a human-robot interaction is a complex task and involves different experts, for example, the need for human-robot interaction (HRI) specialists, who know about the psychological impact of the robot's movements during the interaction in order to design the best possible user experience. In this paper, we present a new project that aims to provide exactly this. Focusing on the interaction flow and movements of a robot for human-robot interactions we aim to provide a set of modelling languages for human-robot interaction which serves as a common, more formal, discussion point between the different stakeholders. This is a new project and the main topics of this publication are the scenario description, the analysis of the different stakeholders, our experience as robot application developers for our partner, as well as the future work we plan to achieve.

Keywords: Robot · Interaction · HRI · Human-robot interaction
MDSO · Software engineering

1 Introduction

Mobile robots are expected to provide all kinds of services for humans in various application scenarios and a dramatic increase of such service robot solutions is foreseen for the near future. However, in many of those scenarios the robots must be able to socially interact with people to respond appropriately to human behaviours and language, to learn and to collaborate with humans on human terms, as well as to act safely in the vicinity of humans. Social robotics aims to achieve this through development of social and communicative skills for physical robots and has become a very active research area in recent years [1, 2]. While many research results exist in single specific areas that contribute to social robotics and while novel mobile robotic platforms offer considerable functionalities for the realisation of social robots at a comparatively low price, the efficient programming of social robots for a target application is still a very challenging

problem. Most often the interdisciplinary integration of the different functionalities such as speech processing, gesture detection, computer vision etc. is solved in an ad-hoc manner for very specific problems, where knowledge and assumptions about the robot's software remain implicit. Additionally, human users show a wide range of possible behaviours creating a high level of interaction uncertainty. Furthermore, social robots often have to be programmed together with domain experts for specific scenarios, these experts are most of the time not robotic experts. A promising approach for the programming of mobile robots in general is model-driven software development. Model-driven approaches are among the most prominent research topics in software engineering and hence several attempts of domain-specific modelling and languages are recently also proposed in robotics. However, many of these approaches do not support the aforementioned special aspects of social robots [3]. The goal of the project together with our partner is to deploy the Pepper robot in a museum environment where it will teach the visitor in an interactive and interesting way about Luxembourg city history. For the moment, we are working on this application using standard modelling tools and languages known from already established software engineering processes. Unfortunately, we quickly realised that they are not really suitable for the design of human robot interactions. The programming of a story telling robot, with all its movement possibilities is a challenging task, even more if the programmer is not in possession of a clear specification. In our lab, the responsible persons for the programming of our Pepper and NAO robots are mostly computer science students with no background in HRI or dialog creation. These are highly complex fields with their own experts. In our case, we consult social science researchers specialised in new technologies for this task. We believe that they have the necessary social experience to become HRI experts. These people however, do not necessarily possess the needed programming skills which makes the whole development process quite long and slow. For every assessment of the robot a meeting is held and the reactions of the robot are discussed and orally agreed upon. This solves the problem of the user friendly design of human-robot interaction, but does not solve the problem that the developer has, namely imprecise specifications of the application. Therefore, we argue that there is a real need for a set of domain-specific languages (DSL) which target the area of human-robot interaction.

In Sect. 2, we describe our project in detail, starting with the robot and its task inside the museum. In Sect. 3, we describe the problems encountered while developing robot applications. Here we also describe the different stakeholders that we consider important for the successful development of sophisticated social robot applications. In Sect. 4, we define the goals that we plan to achieve during the project's evolution. We shortly describe how we plan to tackle the aforementioned problems and conclude in the last section.

2 Project Description

In the project that we initiated together with our partner, the City of Luxembourg, the goal is to use a robot to provide an interactive learning

experience to the visitors of the City History Museum. In a first step, the robot will be programmed to provide the visitors with detailed information about the museum's 360° panorama of an important place in the city centre. For this purpose we acquired the Pepper robot produced by Softbank Robotics¹. We decided to use this robot, because Pepper is a human-shaped robot providing many different interaction possibilities and is among the most prominent commercial robots available for a use-case like ours. We believe that a robot is the right tool for this purpose and in fact, several works analyse the social impact of physical embodiment on social presence of social robots in contrast to a virtual agent solution [4, 5].

In a first step we want to focus on a dialogue language based on state-based, frame-based and plan based techniques [7–9]. It is important to consider these 3 techniques, because they offer different levels of restrictions and complexity, which would allow the programming of a story telling robot as well as applications which allow for greater degrees of user initiative like, for example, an collaborative robots in industry. In a later step, we combine these interaction management methodologies with relevant movement animations inspired by robot control languages like, for example, DANCE [10]. These languages will be implemented inside a tool-chain for robot interaction design that we will provide. Our research will focus on the model-driven software engineering solutions for human-robot interactions, taking into account dialog management as well as robot control.

To develop our toolbox we work together with domain experts from the beginning. For the human-robot interaction we closely work together with researchers from the social science field, situated also at the University of Luxembourg. In future they are supposed to model, using our DSLs, the interactions that will later be programmed by the robot-developer or interpreted directly by the robot. We plan to develop multiple DSLs especially crafted for the dialogue between a robot and a human.

This work will support the design of interactions combining speech and related robot movement behaviours in a way that domain experts can easily understand, implement, or, in the case of the robot developer, transform to robot code. Our past experiences made it clear that designing a human-robot interaction is a difficult task which needs serious input from social experts. Furthermore, we found it very difficult to model and evaluate/discuss human-robot interactions before development due to the lack of suitable languages for this domain.

3 Problem Description & Stakeholders

Current development is done by defining what the robot will say and afterwards the robot's movements are implemented. These movements are selected by the robot developer ad-hoc. One of these behaviours, for example, could be the blinking of the robots eyes before expecting an answer from the user [6]. This

¹ <https://www.ald.softbankrobotics.com/en/cool-robots/pepper>.

basic eye blinking implementation was done by the robot developer without any formal specification and was then changed during meetings together with the social science researcher, who did not necessarily understand why such small programming tasks took such a great amount of time. This process can result in frustration and disappointment on both sides. In the optimal case, instead of focusing on the fine tuning of the robot's interaction, the developer could focus on more complex coding tasks. The fine tuning of the interaction should be left to the social science expert, who understands the complex impact that dialog, motion and emotions have in a human-robot interaction. The aforementioned problems are caused by both the knowledge and expectation gap of the different actors as well as the lack of DSLs which allow domain experts to model such complex multimodal robot dialogues. Such DSLs are important because not only would they speed up the programming process for the robot developer, but they could also allow the testing of robot behaviours in a simulator. Therefore, well designed tools and a well defined development process can, for example, be used by the robot developer to support his programming work. These DSLs will be developed in such a way that they raise the abstraction to such a level that they can be reused in project meetings as a common discussion point which is understood by all involved actors. Considering our scenario, we analyse the different stakeholders to be:

Client: We see the director of the museum as our client. She participates in the requirement elicitation at the beginning of the project. The produced interaction model will be evaluated, thereby we directly involve the museum director in the project and avoid miss-communication between employee and management level, which might happen because some employees' interests might differ from the museum's.

Museum Employee (Historian): The employee responsible for this project is our main contact person. For the moment, this person is in charge of defining, together with our team, the objectives of the project. This is not a very effective way as the employee's goals might be different to the museum director's. Furthermore, often it is the case that this person, as well as the director, is a complete non-technical person which only knows robots from Sci-Fi movies and has ideas which are completely surrealistic. Among his participation during the requirement elicitation, the employee's main output should be the delivery of historical data.

Social Science Expert: Having knowledge about computer's, they do not necessarily have a robotic background so we need to explain the general capabilities of a robot and especially its interaction capabilities. This robot specification will be the same as for the other actors with the main difference that this person needs to look at a lower level of abstraction. He needs, for example, to know what lights on the robot can be controlled and what their constraints are.

Robot Developer: Robot developers have experience in the programming of complex robot applications and are usually not experts in social science or HRI.

She is in charge of the requirement elicitation and manages the different stakeholders. During the implementation phase the developer should be supported by a clear specification of the robot's interaction model developed by the HRI expert. The whole development process should be iterative, based on the interaction models not by trial and error coding (Fig. 1).

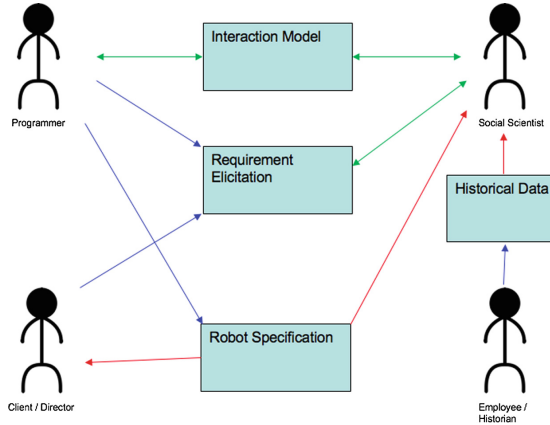


Fig. 1. Expected Inputs and Outputs of the different stakeholders of the project. The arrows represent **Output**, **Input** and **Input/Output**

4 Future Work

In our future work we develop a set of DSLs which allow the modelling of complex human-robot interactions on a higher level of abstraction. In the HRI field, user testing is important and there is a real need for a toolbox which enables fast modelling, development and testing of interactions for different kinds of robots. This toolbox can be used to rapidly develop different versions of a dialog and analyse user feedback to, for example, different dialog motions. Therefore, the final goal of our research is to propose a toolbox that allows HRI experts to design, implement and test complex human-robot interactions.

To achieve this, first we present a dialogue language based on state-based, frame-based and plan based techniques [7–9]. These 3 dialog management techniques offer different levels of restrictions and complexity, which allows the programming of any kind of human-robot dialog.

In a the next step, we combine these interaction management methodologies with relevant movement animations inspired by the robot control languages DANCE [10]. The challenge here is to synchronise dialogue and motion without raising the complexity of the modelling process.

The resulting models should also be usable in simulation environment which allows HRI experts to analyse and fine-tune robot interactions without the direct need of a physical robot, which is time consuming and not always at hand.

5 Conclusion

In this work, we described our project and analysed the different stakeholders. We talked about our experience as robot application developers and highlighted the necessity of a set of DSLs which combine dialog and motion in robotics. Furthermore we gave an outlook on our future works in this field, which we will be starting with the robot dialog language. To our knowledge this domain is not wildly explored and we want to say at this place that any comments or wishes from HRI experts are very welcome.

Acknowledgment. Supported by the Fonds National de la Recherche, Luxembourg (Project ID:11609420)

References

1. Foster, M.E., Alami, R., Gestranus, O., Lemon, O., Niemelä, M., Odobez, J.-M., Pandey, A.K.: The MuMMER project: engaging human-robot interaction in real-world public spaces. In: Agah, A., Cabibihan, J.-J., Howard, A.M., Salichs, M.A., He, H. (eds.) ICSR 2016. LNCS (LNAI), vol. 9979, pp. 753–763. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47437-3_74
2. Pandey, A.K., Alami, R., Kawamura, K.: Developmental social robotics: an applied perspective. *Int. J. Soc. Robot.* **7**, 417–420 (2015)
3. Nordmann, A., Hochgeschwender, N., Wigand, D.L., Wrede, S.: A survey on domain-specific modeling and languages in robotics. *J. Softw. Eng. Robot.* **7**(1) (2016)
4. Jung, Y., Lee, K.M.: Effects of physical embodiment on social presence of social robots. In: *Proceedings of Presence*, pp. 80–87 (2004)
5. Kidd, C.: Sociable robots: the role of presence and task in sociable robots: the role of presence and task in human-robot interaction. Ph.D. dissertation, Massachusetts Institute of Technology (2000)
6. Arend, B., Sunnen, P., Caire, P.: Investigating breakdowns in human robot interaction: a conversation analysis guided single case study of a human-robot communication in a museum environment. *World Acad. Sci. Eng. Technol. Int. J. Mech. Aerosp. Ind. Mechatron. Manuf. Eng.* **11**(5), 839–845 (2017)
7. McTear, M.: Modelling spoken dialogues with state transition diagrams: experiences of the CSLU toolkit. In: *Proceedings of the International Conference on Spoken Language Processing*, Sydney, Australia, vol. 4, pp. 1223–1226. Australian Speech Science and Technology Association, Incorporated (1998)
8. van Zanten, G.V.: Pragmatic interpretation and dialogue management in spoken-language systems. In: LuperFoy, S., Nijholt, A., Veldhuijzen van Zanten, G.E. (eds.) *Dialogue Management in Natural Language Systems*, TWLT11. University of Twente, pp. 81–88 (1996)
9. Allen, J., Byron, D., Dzikovska, M., Ferguson, G., Galescu, L., Stent, A.: Towards conversational human-computer interaction. *AI Mag.* **22**, 27 (2001)
10. Huang, L., Hudak, P.: *Dance: a declarative language for the control of humanoid robots*. Yale, Department of Computer Science, Yale University New Haven, CT 06520, Technical report (2003)

Towards Integration of Context-Based and Scenario-Based Development

Achiya Elyasaf¹, David Harel¹, Assaf Marron¹(✉), and Gera Weiss²

¹ Weizmann Institute of Science, Rehovot, Israel
assaf.marron@weizmann.ac.il

² Ben-Gurion University of the Negev, Beersheba, Israel

Abstract. In scenario-based models of reactive systems complex specifications are divided into artifacts corresponding to separate aspects of overall system behavior, as they may appear, e.g., in a robot’s requirements document or user specifications. The advantages of scenario-based development include intuitiveness and clarity, the ability to execute or simulate specifications of early prototypes and of final systems, and the ability to verify the specification for early detection of conflicts, omissions, and errors. In this position paper we discuss two issues that emerge when applying scenario-based development in complex cases: (a) simple scenarios become unwieldy when subjected to a growing number of conditions, exceptions and refinements, and (b) it is hard to understand and maintain a large ‘flat’ specification, consisting of an unorganized list of independently-specified scenarios, simple as they may individually be. We address these issues by basing certain facets of scenario design on *context*, an increasingly popular foundational consideration in software engineering. We first show how one can incorporate context into the graphical language of live sequence charts (LSC) using existing LSC idioms. We then outline two other possibilities: (i) enriching the LSC language, or (ii) embedding LSCs within hierarchical state machines, namely, statecharts. We believe that this research can contribute to the broader goals of developing complex and powerful reactive systems in intuitive and robust ways.

1 Introduction

In *scenario-based programming* (SBP) one develops software and systems such that distinct aspects of overall system behavior, both desired and forbidden, are implemented in separate behavioral modules, termed *scenarios*. For example, individual paragraphs of a requirement document or of a user manual for a robotic system, are likely to be implemented as separate scenarios. The approach was first introduced by Damm, Harel and Marelly in [4,9] with the graphical language of *live sequence charts* (LSC). It was subsequently generalized and implemented in procedural languages such as Java, C++ and JavaScript [10]. Strengthened with suitable tools, SBP (a.k.a. *behavioral programming*) was shown to have a broad range of advantages, including intuitiveness, clarity and

succinctness of specifications, the ability to directly execute or simulate specifications of early prototypes and of final systems, and the ability to verify such specifications in order to facilitate early detection of conflicts, omissions, and errors (see, e.g., [6–8] and references therein). The research and development of SBP has regularly tried to tackle emerging new challenges that appear (at least at first) to be particular to SBP. An example for such research question is whether the SBP call for independent specification of scenarios increases the risk of specification conflicts. The answer turns out to be that not only do such conflicts often exist already in the originally-stated requirements, but also that the abstractions provided by SBP, and the verification tools that were developed in fact contribute to the early discovery and resolution of such conflicts.

As research of SBP matures and renders the approach suitable for more complex tasks, two new issues arise. The first is that scenarios that start out as simple rules become complex and unwieldy when subjected to a growing number of conditions, exceptions and refinements, coming from all stakeholders, as well as from standards and regulations, as is commonly expected in real-world applications. Consider for example a (futuristic) home-assistant robot which needs to automatically detect and clean up dropped or spilled food. Though the required sensors and actuators are quite sophisticated, the behavioral rules themselves seem simple: *“When food is dropped, clean it up.”*. However, many exceptions can then emerge: *“but not late at night”* (due to vacuum-cleaner noise), or *“but not when anyone is asleep or is on the phone in the same room.”*, or *“but not when the dirt canister is full.”*, etc. Without careful design, such many-to-many relationships between environment conditions and actions can turn a simple specification into a ‘spaghetti’ of exceptions, refinements and alternative paths.

The second issue is that even if individual scenarios are simple, a large specification may become hard to understand and maintain. SBP’s powerful scenario composition is not visible in the individual scenarios, and there is no direct way to capture the organization that may exist in the engineer’s mind. For example, in a use case similar to the DARPA-challenge, a robot that has to drive a car designed for humans, walk over a pile of rubble, climb a ladder and close a valve has to deal with many scenarios. The scenarios’ dependencies may be handled correctly at run time, but, during development, it may be difficult for engineers to allocate development tasks, demonstrate partial prototypes, and plan systematic testing. In fact, the intuitiveness of requirement documents and user manuals stems not only from the natural language of individual sentences but from the document organization, which allows both omission of what is understood from the context, and out-of-context cross-referencing (as is done, e.g., in appendices).

Clearly, solutions to these two issues would align well with the concept of context and context awareness, which have been addressed extensively in software engineering. In this paper, following a brief introduction to scenario-based programming and a discussion of general view on context awareness, we propose solutions to these two issues, which rely on existing LSC constructs and do not require new language idioms or run-time infrastructure. The result is an

approach that further enables the creation of specifications that are intuitive, expressive and powerful, and, most importantly, are executable by a computer. We then proceed to briefly discuss separate research activities and new language constructs aimed at even greater simplification of adding context awareness to scenario-based programming.

2 Scenario-Based Programming with Live Sequence Charts

The LSC language extends *Message Sequence Charts* with rich syntax and semantics that enable intuitive event-based abstraction of behavior to serve both as formal specifications and as the running code in the compositional execution (termed *play-out*) of the final system. The PlayGo tool provides an interactive development and simulation environment, and a stand-alone LSC run-time infrastructure. Similar syntax and semantics were adopted in UML sequence diagrams (SD). Each LSC chart (see example in Fig. 1 depicts a scenario of system behavior. Behavior is represented as event arrows between vertical lines representing objects, with time flowing from top to bottom. Blue and red distinguish events that may happen from those that must happen, and solid arrows represent requests to execute/trigger events while dashed arrows depict events that should be merely waited for, i.e., monitored. Other notations specify forbidden events, if-then-else conditions, loops, and more. The play-out algorithm runs all scenarios in parallel, in a fully synchronized, lockstep manner. Following an environment event, all affected scenarios advance; their declarations of what events *must*, *may*, or *must not* be triggered are consolidated, and an event is selected according to a prescribed strategy (random, priority, or based on look-ahead). All scenarios are notified of this selection and the affected ones proceed accordingly. When all system reactions are complete, the next external, environment-generated event can be dealt with.

3 Context-Based Specifications

There are many approaches to context-oriented programming and to ending procedural programs with context awareness (see, e.g., [1, 3, 11, 12]). We will not delve into the relevant definitions here, nor will we discuss how context-related approaches differ from dealing with environment conditions in standard programming. Instead, we hope that readers will find that our proposals for how to subject intuitive executable specifications to complex conditions fit a variety of needs and design patterns that can qualify as being context based. Nevertheless, to properly set readers' expectations, below are additional context-related examples of the kind we would like to handle: e.g., whether a client or server in a distributed application is initiating an interaction (in sending mode) or listening out for notifications (in receiving mode); how presence of a human, or collaboration with one, affects an industrial robot's operation; how the location

(in orbit or on the ground) affects an autonomous satellite’s handling of events; how battery-charge level of a mobile phone affects its autonomous features; or, how an autonomous car’s speed is to be affected when the road is narrow and/or curved and/or poorly lit.

A contextual condition is not necessarily external and uncontrollable: a robot encountering poor lighting conditions might be able to turn on additional lights and change the context. We also ignore here the fact that particular contextual information (“battery is low”), may also be part of a very particular condition (“battery is now 7% full”);

Context-based designs also allow one to incrementally constrain the system. E.g., if the design (and testing) of a home-assistant robot assumed only typical indoor lighting, and a last-minute pre-shipping concern questions its functioning in a dark room or in a sunlit porch, a makeshift solution can be to physically limit the entire robot behavior to indoor lighting conditions, and when these are not present to pause all activities. Similarly, when verification or extensive testing are to be carried out, the size of the state space or the extent of test-coverage goals can be reduced using context awareness, to enforce simplifying assumptions throughout.

4 Context-Based Design in Native LSC

First, a key methodological point we propose is that contexts should play a primary role in initial analysis. Hence, entities that may otherwise be modeled as properties or as inter-object behavior (e.g., the facts that someone is asleep in the house, or that two robots are collaborating) may need to be modeled as objects in their own right.

Second, for context-based design in native LSCs (abbr. CBLSC), we propose that instead of refining scenarios as one would refine ordinary programs, i.e., by adding context conditions locally prior to triggering the actions that depend on them, the activation (namely, the very relevance) of entire scenarios should be subjected to the presence of desired contexts, as follows (see also Fig. 1):

Dynamic objects. In LSC, objects of all types can be created and destroyed dynamically. This can be done from any scenario by executing an appropriate event.

Binding expressions. The binding of a lifeline to an object instance can be subjected to a binding expression that specifies the instance(s) to be bound (if more than one, the scenario is replicated).

Dynamic binding. By default a scenario is not active. When a monitored event that appears as the first one in a scenario, is triggered by the environment or by the system, the scenario is activated and a new *live copy* thereof participates in play-out until its termination. Lifelines are then bound dynamically as specified. But, if there is a lifeline that cannot be bound, as no object satisfies its binding expression, the live copy terminates.

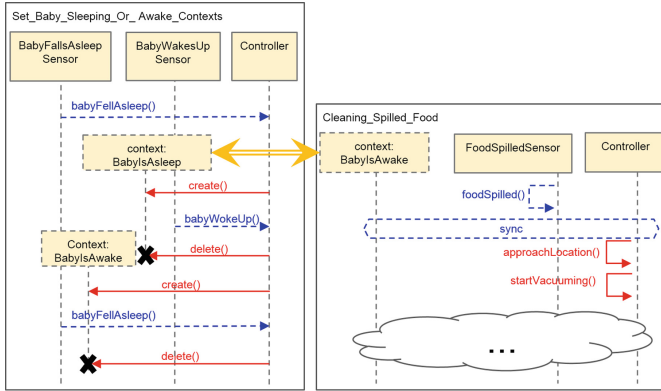


Fig. 1. Creating some context objects for a home-assistant robot (left) and subjecting a noisy home-cleaning scenario to one such context (right). (Color figure online)

Context objects. It is common to infer a current context by checking the values of object properties, like `babyIsAsleep==true` or `batteryPercent<10`. The relevant objects usually persist despite changes in these values. By contrast, in CBLSC, whether a context holds or not could depend on whether or not an object like `babyIsAsleep` or `batteryLow` is instantiated.

Scenario-driven creation of context objects. The examination of possibly complex conditions and events that determine whether a context holds is done in one or more dedicated scenarios. Composite contexts can be similarly created by monitoring conjunctions, disjunctions and other relationships of other contexts.

Subjecting scenarios to context objects. Scenarios specify context dependencies by having lifelines for relevant context objects (even if no events occur in these lifelines). When contexts apply to only a small part of a scenario, one can split the scenario into its parts, or replicate, or use condition constructs instead.

Context-termination handling. Graceful context termination is still the developer's task, e.g., terminate the affected scenario immediately; activate scenarios to handle the new situation (including completing in-flight activities as in exception handling); or use events to notify active scenarios that they need to terminate ASAP.

Summary. With the above constructs, each scenario can concisely specify both the required behavior and the contexts in which the specified reactivity applies.

5 Towards Intuitive Organization of Context-Based Specifications

To streamline the management of large flat collections of independently-specified scenarios, we propose to add the following to specification management in LSC tools:

Textual scenario views. Optionally hide the graphical chart view, and show select details thereof, such as name, text comments, affected objects, relevant contexts, key events, or a textual description of the scenario’s flow (now available automatically).

Navigable specifications. Navigate specifications according to function, context, structure and dependencies. This can be done by adding indexing, queries and filters, sorted lists, trees and rectangle-containment views, dependency graphs, etc.

Feature-model-like design of context awareness. We propose to design context/scenario relationships to resemble feature models [2] and software product lines, aligning contexts with key user requirements, system functions, or target environments.

Multi-hierarchies of contexts. We believe that humans find it easier to understand and manage contexts that are hierarchical, with sub-division of properties such as time (r.g., day vs. night and then specific hour) or location (e.g., city, street, building, room). As orthogonal hierarchies intersect, they can still be navigated and understood using the above idioms. Intuitive visual representations, such as multi-hierarchies, include a forest of tree hierarchies. Intersections can be shown with directed edges between trees while keeping the entire graph acyclic, or by connecting context nodes from different trees to a common set of scenarios.

Summary. Once scenarios are both *subjected to* and *organized by* contexts, several potential advantages emerge as compared with implementing contexts as in-line conditions (clearly, empirical quantification and assessment remain as future work): (a) When a scenario is subjected to context objects such as `No_One_Is_Asleep` or `DayTime`, it is clear that it is applicable when the context conditions do hold. With statements like `if NoOneIsAsleep` or `if TimeOfDay >= 22 and TimeOfDay <= 07`, even rich classical search commands cannot readily inform of both the properties checked, their desired values, and the actions that are taken or skipped. (b) With contexts, one can readily check against the requirements which scenarios are applicable in which contexts without examining implementation code. (c) When in-line conditions are replaced with contexts, each scenario can be better understood as doing just one or very few tasks.

6 Research on New Language Idioms

We are pursuing two additional lines of work related to context orientation. One is adding specific syntax and semantics to LSC for creating and destroying context objects, subjecting scenarios to contexts, and other features related to context based design (see a report in www.b-prog.org/morse17s).

Another direction is based on embedding LSC within the intuitive hierarchical structure of statecharts [5], which allows both state containment and orthogonality, which in turn align well with contexts. Contexts will be associated with statechart states, and LSC scenarios that are associated with a context state

will participate in play-out only when the system transitions into that state. This will be complementary to our work on incorporating statecharts within an LSC scenario (see a report in www.b-prog.org/morse17s). Another advantage of statecharts is that their concurrency feature, namely, the ability to condition a transition on whether other orthogonal parts of the system are in a certain state, is an excellent basis for implementing multi-hierarchies.

Another relevant question is whether a scenario should have access to context objects in which it is invoked, with all their details, causes, and other dependencies.

We believe that contexts are a central concept in system analysis, and have shown how context-based design can be incorporated into executable, scenario-based specifications, using existing LSC idioms. We have also outlined approaches for making the entire specification easier to understand via navigational features, new language idioms, and integration with statecharts. We hope that our work will contribute to the search for languages that can produce intuitive models that are also powerful, executable programs.

References

1. Abowd, G.D., Dey, A.K., Brown, P.J., Davies, N., Smith, M., Steggles, P.: Towards a better understanding of context and context-awareness. In: Gellersen, H.-W. (ed.) HUC 1999. LNCS, vol. 1707, pp. 304–307. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48157-5_29
2. Apel, S., Kästner, C.: An overview of feature-oriented software development. *J. Object Technol.* **8**(5), 49–84 (2009)
3. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: context-oriented programming with Java. *Inf. Media Technol.* **6**(2), 399–419 (2011)
4. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. *J. Form. Methods Syst. Des.* **19**, 45–80 (2001)
5. Harel, D.: Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.* **8**(3), 231–274 (1987)
6. Harel, D.: Can programming be liberated, period? *IEEE Comput.* **41**(1), 28–37 (2008)
7. Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., Weiss, G.: On composing and proving the correctness of reactive behavior. In: EMSOFT (2013)
8. Harel, D., Katz, G., Lampert, R., Marron, A., Weiss, G.: On the succinctness of idioms for concurrent programming. In: CONCUR, pp. 85–99 (2015)
9. Harel, D., Marelly, R.: *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, Heidelberg (2003). <https://doi.org/10.1007/978-3-642-19029-2>
10. Harel, D., Marron, A., Weiss, G.: Behavioral programming. *Commun. ACM* **55**, 90–100 (2012)
11. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. *J. Object Technol.* **7**(3), 125–151 (2008)
12. Keays, R., Rakotonirainy, A.: Context-oriented programming. In: *MobiDE 2003* (2003)

(An Example for) Formally Modeling Robot Behavior with UML and OCL

Martin Gogolla¹ and Antonio Vallecillo²(✉)

¹ University of Bremen, Bremen, Germany
gogolla@informatik.uni-bremen.de

² Universidad de Málaga, Málaga, Spain
av@lcc.uma.es

Abstract. One of the problems that the design and development of robotic applications currently have is the lack of unified formal modeling notations and tools that can address the many different aspects of these kinds of applications. This paper presents a small example of a chain of robotized arms that move parts in a production line, modeled using a combination of UML and OCL. We show the possibilities that these high-level notations provide to describe the structure and behaviour of the system, to model some novel aspects such as measurement uncertainty and tolerance of physical elements, and to perform several kinds of analyses.

1 Introduction

Robotic applications are difficult to design, develop and check because of the inherent properties of these kinds of systems and their multi-faceted characteristics. For example, they are composed of heterogeneous parts difficult to model at the same level of abstraction, and to describe with a single notation. Besides, the heterogeneity of the available hardware platforms for robots and the lack of hardware and software standardization severely hampers cross-product development. Having to deal with physical components also implies the need to incorporate some particular properties such as continuous flows, mechanical forces, tolerance and accuracy. Finally, their different nature from software systems hinders in theory their specification with traditional software modeling notations and tools.

This paper presents a small example of a chain of robotized arms that move parts in a production line, modelled using a combination of UML and OCL. We show how these high-level and platform-independent notations permit modeling the system in a formal manner, taking into account novel aspects such as tolerance and measurement uncertainty, and allow performing several interesting analyses on the system, such as visualization of the system in operation, simulation of its behaviour, and validation of several properties of interest to the designer. We claim the importance of having a unified language (e.g., UML and OCL) for describing the functionality of the different parts within a robotic system in order to understand and analyse the system and its different parts,

at least for the central functionalities. In this respect, our approach is different from the various domain-specific languages/proposals for robotic applications that combine specialized languages, since we want to explore the possibilities that UML and OCL offer to model robotic systems.

The rest of the paper is structured as follows. The next Sect. 2 describes the example and shows interesting properties using the USE modeling environment [6, 7]. Section 3 discusses related work. The paper ends with a conclusion and future work.

2 A UML and OCL Model for a Production Line with Robotized Arms

This section will first discuss structural system elements before we turn to the behavioral aspects. After that various visual property analysis options and formal test aspects of our approach are debated. All codes and some additional material about the example can be found at <https://goo.gl/DdLivi>.

Structural Elements. To illustrate our approach suppose a system composed of producers and consumers, as shown in Fig. 1. **Producer** machines generate **Items**, which once finished are placed in **Trays**. When informed that there is an element ready in a tray, a **Consumer** takes the generated item from that tray, performs some further work units on it, and stores it in a second tray (the **storageTray**). Assuming we are in a robotized environment, each tray has a **RobotArm** in charge of physically moving the items around. When asked to perform a **put** operation, the tray asks the arm to go where the item is, **grasp** it, **move** it to the position of the tray, and **drop** it there. Similarly, a **get(c:Coordinate)** operation on a tray makes the arm grasp the item from the tray, move it to the position that the caller has indicated (therefore the parameter of this operation), and drop it there. Consumers and producers keep a **counter** with the items they have handled, and trays have a limit on their capacity (attribute **cap**).

An important characteristic of any system that deals with physical objects is the associated measurement uncertainty, due to tolerance of the mechanical parts and the lack of precision of the arm movements. In order to deal with this kind of uncertainty, we make use of an extension of OCL and UML type **Real**, called **UReal** [17] that permits expressing values of physical quantities as pairs (x, u) where x represent the value and u the associated uncertainty, following the International Guide to the Expression of Uncertainty in Measurement (GUM) [9]. The corresponding operations on this type take also into consideration the propagation of uncertainty when uncertain values are added, multiplied, or other arithmetical operations are performed [10, 17]. This is why all coordinates are expressed by **UReal** numbers.

Each robot arm in this system also has an associated **tolerance** that represents the deviation that the arm may introduce when performing a movement. Besides, when the arm is asked to grasp an item, we check whether the position

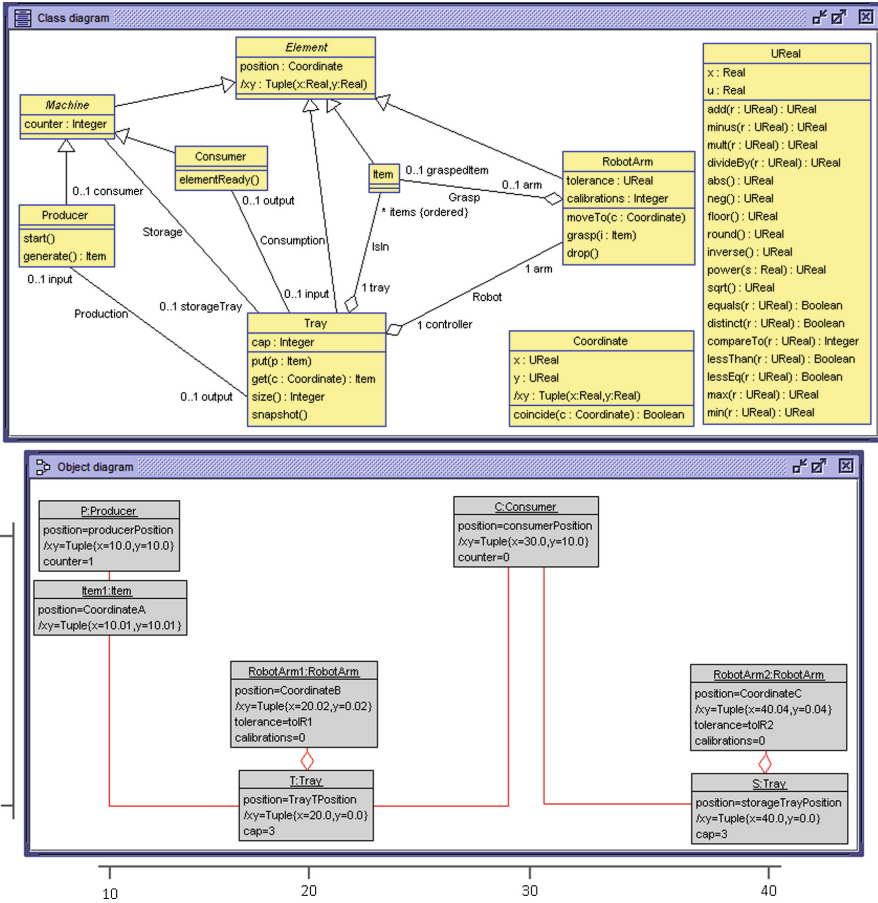


Fig. 1. Class and object diagram for robot arm example.

of the arm `coincides` with the position of the item. In case it does not (due to accumulated uncertainty or excessive tolerance), the arm needs to be calibrated and this is stored in an attribute that keeps track of how many `calibrations` each arm has already needed. Note that every calibration introduces a delay in the system and may have associated costs, and this is why it is important to know how often they occur. Finally, the derived attribute `/xy` is used to facilitate (an approximation of) the visual representation of the objects' coordinates in the real world and the UML diagrams.

Behavioral Elements. The behavior of the system can be expressed in UML and OCL by different means. First, pre and postconditions can be specified on the operations, as shown here for `grasp()` and `drop()` operations of a `RobotArm`.

```

grasp(i:Item)
  pre  notWithItem: graspedItem.oclIsUndefined()
  post withItem: graspedItem=i
  post  calibrationsCount: not self.position@pre.coincide(i.position)
        implies calibrations = calibrations@pre + 1
  post  reposition: self.position.coincide(i.position)
drop()
  post  notWithItem: graspedItem.oclIsUndefined()

```

State machines can also be specified on objects. For example, the following listing shows the specification of the *state machine* of a Tray.

```

psm PutGet
states
  init:  initial
  Empty [self.items->size()=0]
  Normal [0<self.items->size() and self.items->size()<self.cap]
  Full  [self.items->size()=self.cap]
transitions
  init  -> Empty { create }
  Empty -> Normal { [self.cap>1] put() }
  Normal -> Normal { [self.items->size()<cap-1] put() }
  Normal -> Full { [self.cap>1 and self.items->size()=cap-1] put() }
  Empty -> Full { [self.cap=1] put() }
  Full -> Empty { [self.cap=1] get() }
  Full -> Normal { [self.cap>1] get() }
  Normal -> Normal { [self.cap>1 and self.items->size()>1] get() }
  Normal -> Empty { [self.items->size()=1] get() }
end

```

On top of that, USE also permits to specify the behavior of operations using a simple executable language called SOIL [4]. For instance, the behavior of `Tray::put()` and `RobotArm::moveTo()` operations can be specified as follows.

```

put(p:Item)
  begin
    insert(self,p) into IsIn;
    self.arm.moveTo(p.position);
    self.arm.grasp(p);
    self.arm.moveTo(self.position);
    self.arm.drop();
  end
  pre  notFull: self.items->size()<cap
  pre  armNotWithItemAtPre: arm.graspedItem=null
  post ElementAdded: self.items=self.items@pre->append(p)
  post armNotWithItemAtPost: arm.graspedItem=null
moveTo(c:Coordinate)
  begin
    declare aux:Coordinate;
    aux := new Coordinate;
    aux.x := c.x.add(self.tolerance);
    aux.y := c.y.add(self.tolerance);
    self.position := aux;
    if self.graspedItem->size() > 0 then
      self.graspedItem.position:=self.position;
    end
  end
end

```

Expressing and Proving Properties. Once we have the specifications, there are different kinds of analyses that we can perform on the system that show some of the potential advantages of developing model-driven robot descriptions with UML and OCL, such as:

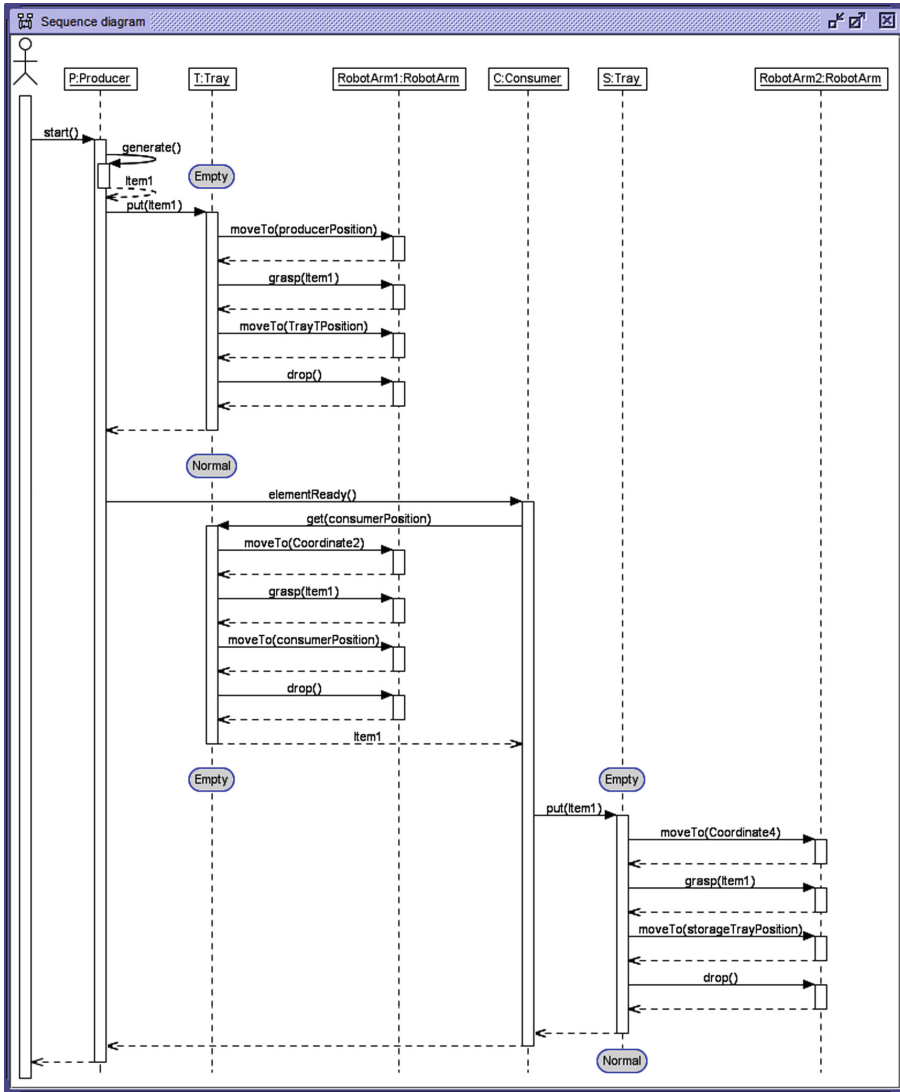


Fig. 2. Sequence diagram displaying the behavior of the system.

- visualization of complex structures and processes
- execution and simulation of scenarios (operation call sequences)
 - different scenarios with different structural properties e.g. trays with different capacities
 - variations of a single scenario with equivalence checking by analysing different operation call orders
- checking structural properties within states by OCL queries
 - e.g. calculating the number of currently produced items

- checking behavioral properties
 - e.g. testing the executability of an operation by testing its preconditions
- checking for weakening or strengthening model properties (invariants, contracts, guards) by executing a scenario with modified constraints
- proving general properties within the finite search space with the USE model validator [7]
 - structural consistency, i.e. all classes are instantiable
 - behavioral consistency, i.e. all operations can be executed
 - checking for deadlocks, e.g. construction of deadlock scenarios due to inadequate tray capacities.

For example, based on the specifications above we are able to simulate the system, by creating an initial model of the system and invoking the `start()` operation to the producer. Then, if we have created a system with one producer and one consumer, and the producer just generates one item, the behavior of the system is recreated as shown using the UML sequence diagram in Fig. 2. The sequence diagram shows lifelines for objects and called messages. The evolution of a `Tray` object can also be traced by checking the statechart states that are placed on the lifelines. The behavior can also be displayed as a communication diagram (Fig. 3). As one detail, we emphasize that the `RobotArm1` with the second `moveTo` call moves to `Coordinate2` (displayed in the shown object diagram). `Coordinate2` is close to the `TrayTPosition` but not the exact `TrayTPosition`. This is possible in our approach that allows for uncertain real values.

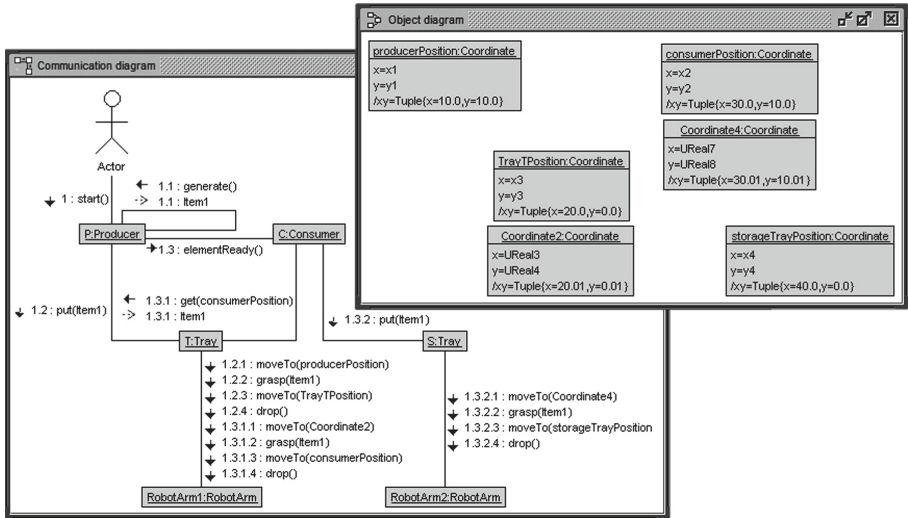


Fig. 3. Communication diagram.

Similarly, for every step we obtain the state machines of the `Tray` objects, which can be shown as depicted in Fig. 4.

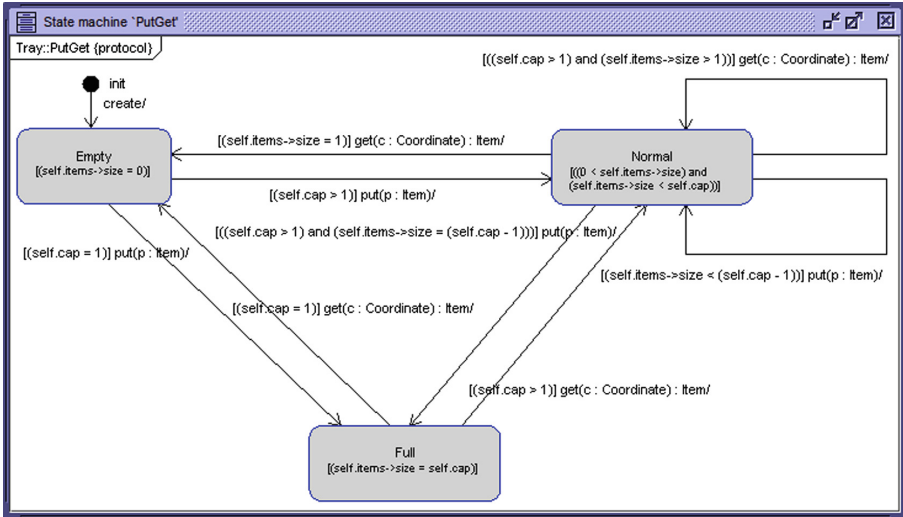


Fig. 4. State machine for tray objects.

Finally, the last object diagram in Fig. 6 shows the resulting system state after the system has gone through an iteration. We can see the final positions of the item and the arms. We can also see how the high tolerance that we have indicated for the two robot arms has caused two calibrations.

Figures 5 and 6 pictorially show a filmstrip of the behavior of the system as a sequence of snapshots after every robot arm operation. One can trace in the figures the movements of the *Item*, the *RobotArm1* and the *RobotArm2*. These two figures show two different aspects: a time dimension (through the sequence of diagrams) and a space dimension (within the single object diagrams). The physical placement of the objects is captured by their position in the diagrams: some objects have a fixed position (e.g. the producer, consumer and trays) while others ‘move’ from object diagram to object diagram as in the real process, e.g. the *Item1* and the two robot arms. Uncertainty is captured through *UReal* values. Aggregation associations are used to visualize ‘ownership’ between objects (e.g. a robot arm has an item, or an item is placed on a tray).

Another interesting representation of the system behavior is shown in Fig. 7. It depicts a communication diagram showing the operation calls involving an *Item* object. We have also included the associations in which the *Item* engages during the execution of the system (*IsIn*, *Grasp*) as a result of the operations. This diagram is very useful to check the order in which operations are called, and their effects. One can trace that the item *Item1* is first created, then put in the first tray, and finally put in the second tray.

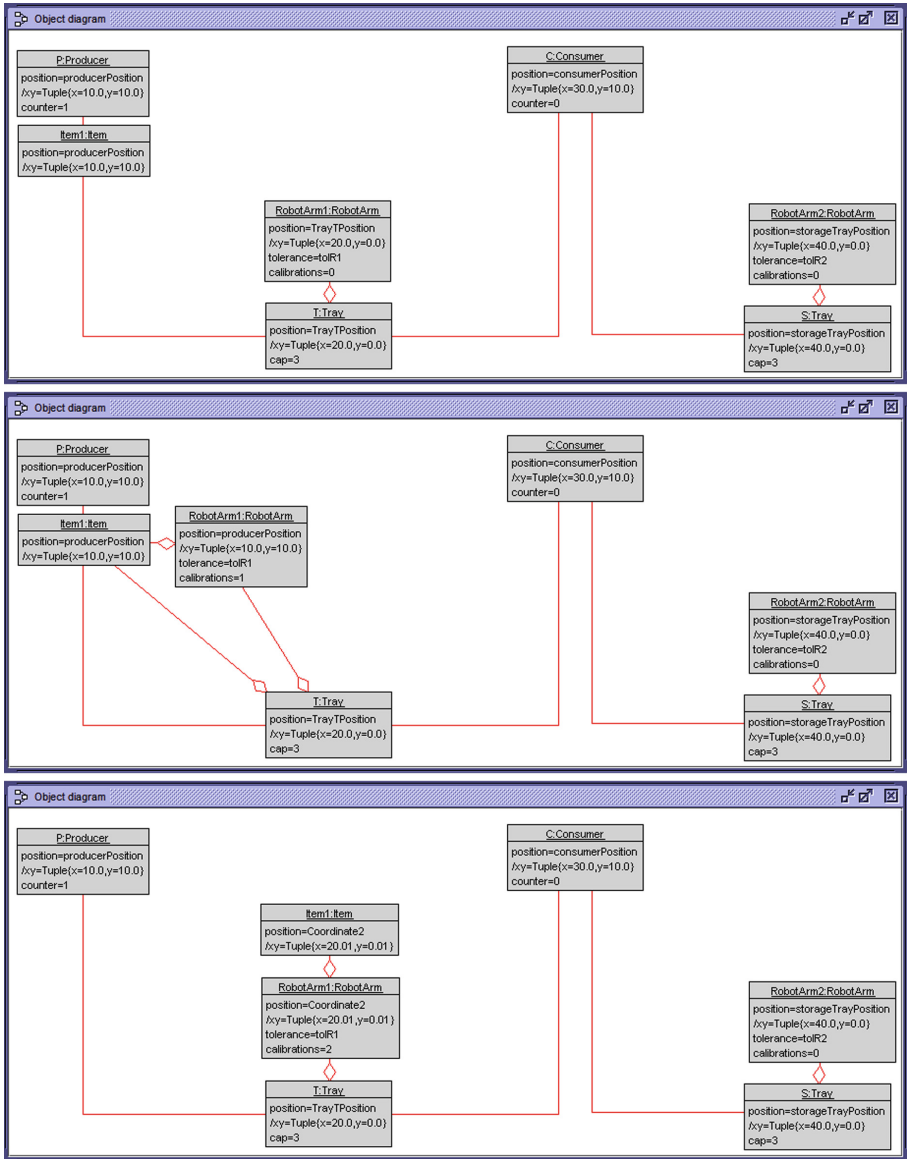


Fig. 5. Object diagram sequence displaying the behavior of the system (Part 1).

Validation and Verification Through Testing. Finally, we want to highlight the importance of running structural tests on the metamodels. One of them concerns their instantiability and their ability to faithfully represent the application domain. For example, we decided to ask the USE model validator [7] to generate a producer-consumer-item-tray constellation using the system metamodel.

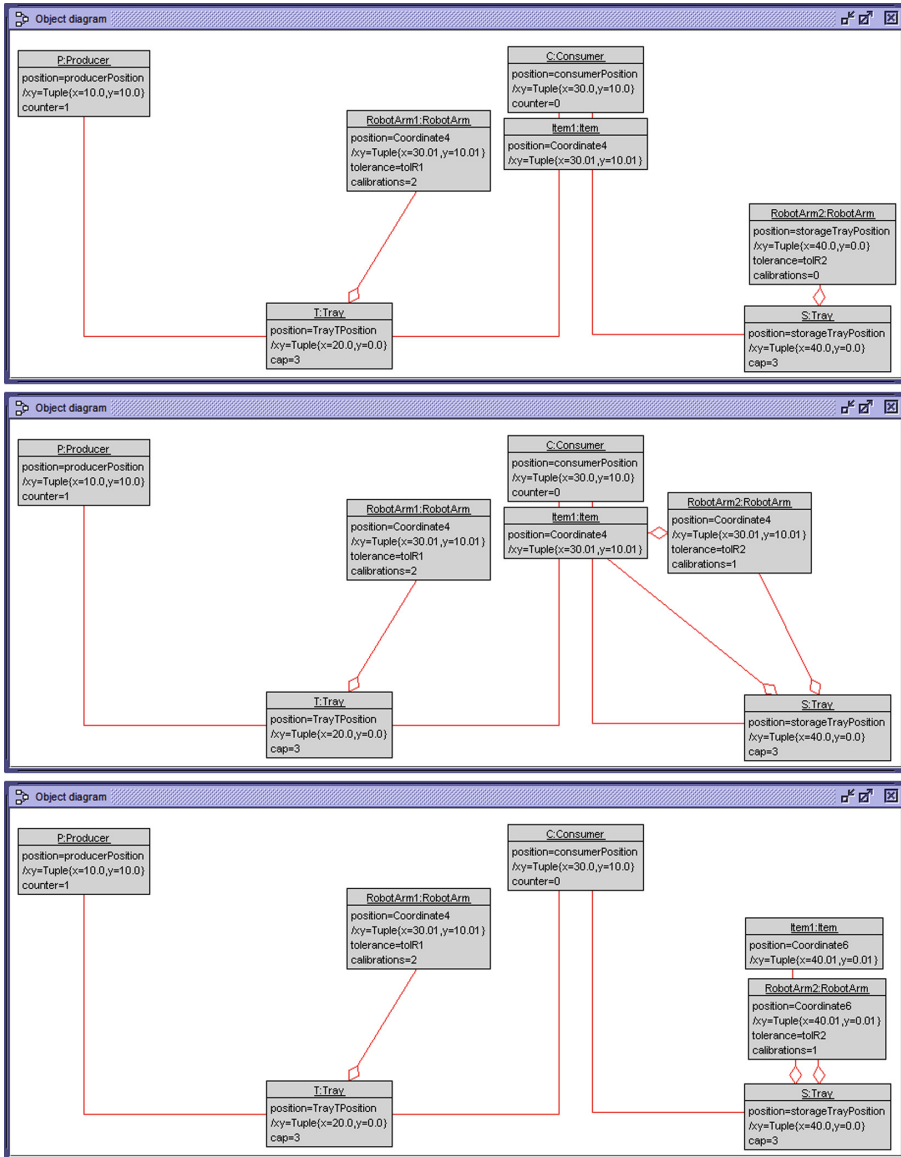


Fig. 6. Object diagram sequence displaying the behavior of the system (Part 2).

The resulting object diagram is shown in Fig. 8, together with the model validator configuration that we employed (e.g., optional minimum and mandatory maximum number of instances per class; analogous specifications for associations and datatypes). Interestingly, the produced system is wrong! For example, the producer and consumer are disconnected, sharing no tray between them.

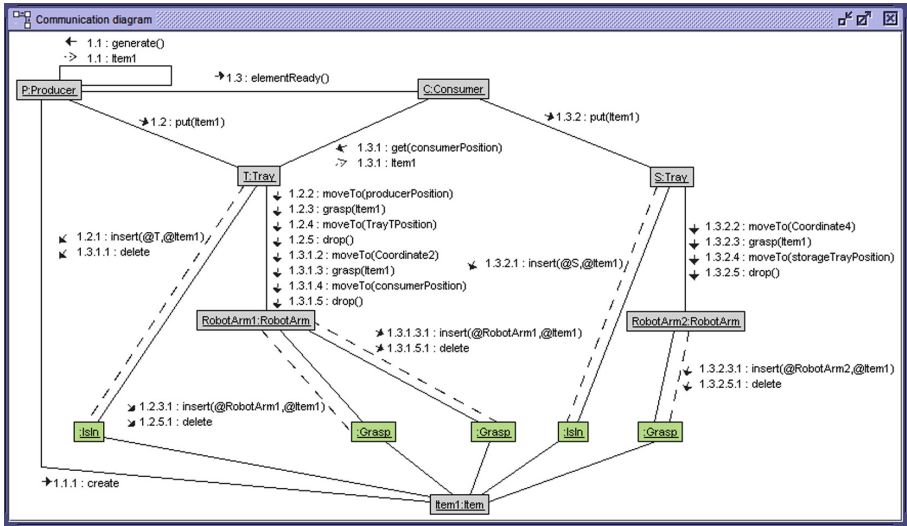


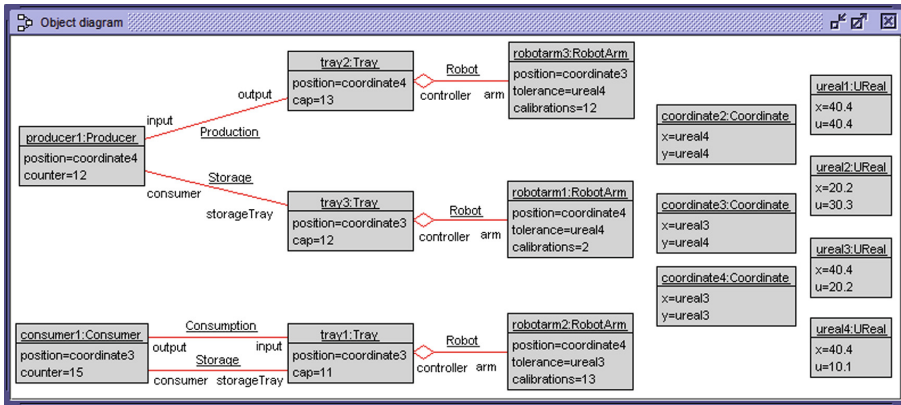
Fig. 7. Communication diagram showing operation calls involving an Item object.

This motivates the need to develop additional, currently missing invariants (on the structural system constellation level) and demonstrates the potential usefulness of this approach for validating this kind of properties which are normally overlooked for being considered obvious.

Figure 9 shows another generated test case indicating missing constraints. This time an additionally loaded invariant guarantees a proper Producer-Consumer-Tray constellation. However, erroneously `robotarm2` can grasp an item from the storage tray, and `robotarm1` can grasp an item from the producer output tray. Furthermore, the test case reveals that constraints on the coordinates of machines and trays are missing. The underlying model validator configuration basically looks like the one presented in Fig. 8.

Another analysis option in our approach with regard to behavioral aspects is, that the developer can check in a system state for the applicability of an operation, for example for the operation `Tray::get(c:Coordinate)`. One can check whether the preconditions of an operation when applied on a particular object together with appropriate parameters are satisfied. One can also construct a respective set of tuples with possible objects to apply the operation to and corresponding parameters.

The OCL query below retrieves from the last object diagram in Fig. 6 the possible operation calls by returning tuples with a tray `TRAY` whose item set is not empty and whose robot arm is not grasping an item (as required in the preconditions of the operation `get`) together with the coordinate `COORD` of the tray: the result is constructed in such a way that the preconditions of the operation call `TRAY.get(COOR)` would be satisfied for all elements of the returned tuple



Model element	Model validator bounds
Producer	1..1
Production (input:Producer, output:Tray)	1..*
Consumer	1..1
Consumption (input:Tray, output:Consumer)	1..*
Storage (consumer:Machine, storageTray:Tray)	1..*
RobotArm	3..3
Item	0..0
Grasp (arm:RobotArm, graspedItem:Item)	0..*
Tray	3..3
IsIn (tray:Tray, items:Item)	0..*
Robot (controller:Tray, arm:RobotArm)	1..*
Coordinate	0..4
URReal	0..4
Integer	0..15
Real	0.1, 0.2, 0.3, 0.4, 10.1, 20.2, 30.3, 40.4

Fig. 8. Generated test case for Producer-Consumer-Tray configuration.

set, a singleton in this case. Concerning the object diagram in Fig. 9, the query would show that the operation `get` is not applicable there.

```

get(c:Coordinate):Item
pre notEmpty: self.items->size()>0
pre armNotWithItem: self.arm.graspedItem=null

Tray.allInstances->
  select(self|self.items->size()>0)->
    select(self|self.arm.graspedItem=null)->
      collect(t| Tuple{TRAY:t, COOR:t.position})->asSet()

Set{Tuple{TRAY=S, COOR=storageTrayPosition}}:
Set(Tuple(TRAY:Tray, COOR:Coordinate))
    
```

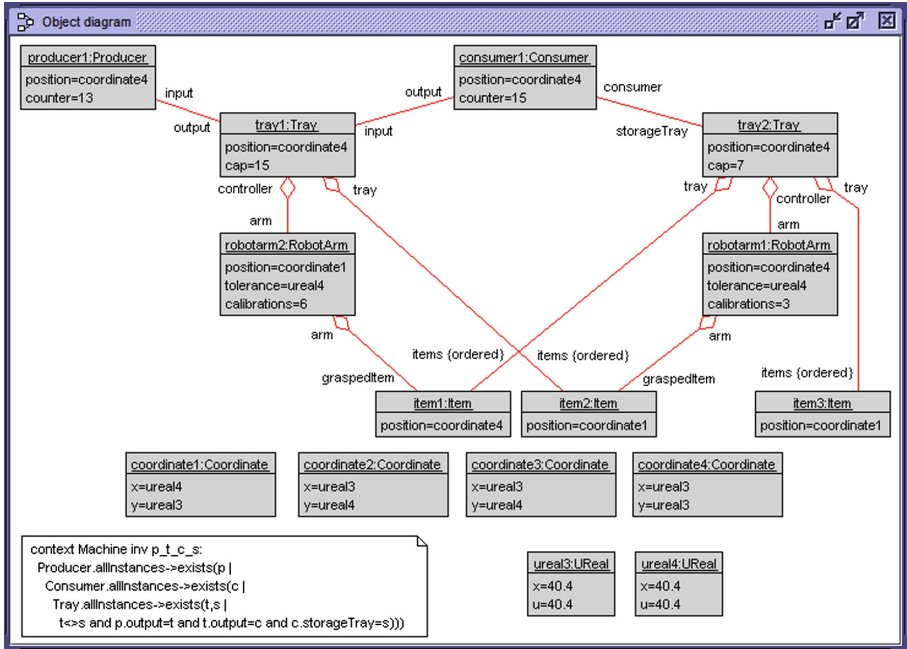


Fig. 9. Test case showing missing constraints for IsIn and Grasp associations.

3 Related Work

There are different kinds of works that use MDE techniques for modeling robotic applications, depending on their purpose. One set of works focuses on the automatic generation of components, control logic and other artefacts for the implementation of robotic systems [5, 19, 20]. Other works focus on transformations between models of different analysis tools that serve as bridges between the separate semantic domains [8, 11]. And there are those works that propose models for describing at a high-level and in a platform- and technology-agnostic manner the algorithms and choreography that robotic systems composed of several cooperating agents have to perform to achieve their goals [16, 22].

Our paper is more closely related to those works that focus on the specification of the robotic systems themselves. Here the discussion happens between those that propose the use of separate (related) views of the system, using independent domain-specific languages, and those that try to use general purpose modeling languages. One of the major problems with the former approach is the combination of the languages, both horizontally (i.e., at the same level of abstraction—see, e.g. [23]) and vertically (one example of this kind of vertical combination for robotic systems is [2], that uses deep metamodeling [21] to combine system descriptions at different levels). Among the latter, the most widely known ones use high-level component-based architectures with the functional decomposition of the robotic systems, using block-diagrams and/or UML

components. Examples include SafeRobots [18], RobotML¹, SmartSoft², BCM³, V3CMM [1] and HyperFlex [3]. Our approach sits at a higher level of abstraction, when not even the architecture of the system needs to be considered, just its basic functionality, and hence many of the details can be abstracted away for later consideration.

We have explored in this paper the option of using a widely used general-purpose modeling language, such as UML, augmented with OCL for the specification of integrity constraints, and pre- and postconditions of operations. On top of them we have used some extensions and tools: (a) to be able to execute the specifications we have used SOIL [4]; (b) the USE model validator has been employed to generate instances of the model; finally, we have shown how the UML/OCL type system can be easily extended to account for some specific features—namely measurement uncertainty, by defining type `UReal` as an extension of type `Real`. We wanted to follow this path to study its feasibility and expressive power, departing from other approaches that enrich UML with Profiles (such as MARTE [15] or SysML [13]) and make use of action languages like Alf [12] for executing fUML [14] specifications.

A comparison with the pros and cons of our approach with regard to those others is part of our future work, now that we have seen that we are able to get a relevant set of meaningful and workable specifications of these kinds of systems.

4 Conclusions and Future Work

In this paper we have illustrated the possibilities that UML and OCL offer to model robotic systems at a high-level of abstraction but still providing some key benefits to the system designer. In particular, we are able to describe in a formal manner its structure and behaviour; to incorporate some physical characteristics such as measurement uncertainty; to validate of some of the structural and behavioural properties of the system, and to perform simulation.

There are several lines of work that we plan to address next. First, we want to explore the limitations of our approach due to the type of notations employed. For example, both UML and OCL can handle discrete quantities but are not naturally devised to deal with continuous variables. Some of them are difficult to overcome, but others could have relatively easy solutions. For example, we want to add randomness and other types of uncertainty into our OCL models—e.g., the fact that up to 5% of the generated parts can be defective. We also want to be able to conduct performance analyses about the production time of the system, using e.g. model attributes that specify the time each machine needs to process a part—adding probability distributions to the description of the types of the attributes.

Finally, given that our models just represent early prototypes of the system to study the feasibility of the solution, we want to connect our models to the

¹ <http://robotml.github.io/>.

² <http://smart-robotics.sourceforge.net/>.

³ <http://www.best-of-robotics.org/bride/bcm.html>.

different analysis and simulation tools currently used in industry, each one able to conduct more fine-grained and precise validations, but of a more heterogeneous nature. In this way, we expect our high-level models to play a pivotal and unifying role that permit connecting the modeling and simulation tools needed for the complete design and validation of these systems.

References

1. Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor, J., Alvarez, B.: V3CMM: a 3-view component meta-model for model-driven robotic software development. *J. Softw. Eng. Robot.* **1**(1), 3–17 (2010)
2. Atkinson, C., Gerbig, R., Markert, K., Zrianina, M., Egurnov, A., Kajzar, F.: Towards a deep, domain specific modeling framework for robot applications. In: *Proceedings of MORSE 2014. CEUR WS Proceedings*, vol. 1319, pp. 1–12 (2014)
3. Brugali, D., Gherardi, L.: HyperFlex: a model driven toolchain for designing and configuring software control systems for autonomous robots. In: Koubaa, A. (ed.) *Robot Operating System (ROS)*. *SCI*, vol. 625, pp. 509–534. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26054-9_20
4. Büttner, F., Gogolla, M.: On OCL-based imperative languages. *Sci. Comput. Program.* **92**, 162–178 (2014)
5. Djukić, V., Popović, A., Tolvanen, J.P.: Domain-specific modeling for robotics: from language construction to ready-made controllers and end-user applications. In: *Proceedings of the 3rd WS Model-Driven Robot Software Engineering, MORSE 2016*, pp. 47–54. ACM (2016)
6. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.* **69**, 27–34 (2007)
7. 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)*, GI, LNI, vol. 254, pp. 203–218 (2016)
8. 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: *Proceedings of the 2015 Joint MORSE/VAO WS Model-Driven Robot Software Engineering and View-based Software-Engineering*, pp. 9–15. ACM (2015)
9. 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
10. Mayerhofer, T., Wimmer, M., Vallecillo, A.: Computing with quantities (2016) <https://github.com/moliz/moliz.quantitytypes>
11. Morozov, A., Janschek, K., Krüger, T., Schiele, A.: Stochastic error propagation analysis of model-driven space robotic software implemented in Simulink. In: *Proceedings of the 3rd WS Model-Driven Robot Software Engineering, MORSE 2016*, pp. 24–31. ACM (2016)
12. Object Management Group: Action language for foundational UML (FUML), version 1.0.1. (October 2013). OMG Document formal/2013-09-01. <http://www.omg.org/spec/ALF/1.0.1/PDF/>
13. Object Management Group: OMG Systems Modeling Language (SysML), version 1.4. (January 2016). OMG Document formal/2016-01-05

14. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (FUML), version 1.2.1. (January 2016). OMG Document formal/2016-01-05. <http://www.omg.org/spec/FUML/1.2.1/PDF/>
15. OMG: UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE). Object Management Group. (June 2008). OMG doc. ptc/08-06-08
16. Opfer, S., Niemczyk, S., Geihs, K.: Multi-agent plan verification with answer set programming. In: Proceedings of the 3rd WS Model-Driven Robot Software Engineering, MORSE 2016, pp. 32–39. ACM (2016)
17. Orue, P., Morcillo, C., Vallecillo, A.: Expressing measurement uncertainty in software models. In: Proceedings of QUATIC 2016, pp. 1–10 (2016)
18. Ramaswamy, A., Monsuez, B., Tapus, A.: Model-driven software development approaches in robotics research. In: Proceedings of MISE 2014, pp. 43–48. ACM (2014)
19. Ringert, J.O., Rumpe, B., Wortmann, A.: Tailoring the MontiArcAutomaton component & connector ADL for generative development. In: Proceedings of the 2015 Joint MORSE/VAO WS Model-Driven Robot Software Engineering and View-Based Software-Engineering, pp. 41–47. ACM (2015)
20. Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: Code generator composition for model-driven engineering of robotics component and connector systems. In: Proceedings of MORSE 2014. CEUR WS Proceedings, vol. 1319, pp. 63–74 (2014)
21. Rossini, A., de Lara, J., Guerra, E., Rutle, A., Wolter, U.: A formalisation of deep metamodelling. *Formal Asp. Comput.* **26**(6), 1115–1152 (2014)
22. Ruscio, D.D., Malavolta, I., Pelliccione, P.: A family of domain-specific languages for specifying civilian missions of multi-robot systems. In: Proceedings of MORSE 2014. CEUR WS Proceedings, vol. 1319, pp. 13–26 (2014)
23. Vallecillo, A.: On the combination of domain specific modeling languages. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 305–320. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13595-8_24

Synthesizing Executable PLC Code for Robots from Scenario-Based GR(1) Specifications

Daniel Gritzner^(✉) and Joel Greenyer^(b)

Fachgebiet Software Engineering, Leibniz Universität Hannover,
Welfengarten 1, 30167 Hannover, Germany
{daniel.gritzner,greenyer}@inf.uni-hannover.de

Abstract. Robots are found in most, if not all, modern production facilities and they increasingly enter other domains, e.g., health care. Robots participate in complex processes and often need to cooperate with other robots to fulfill their goals. They must react to a variety of events, both external, e.g., user inputs, and internal, i.e., actions of other components or robots in the system. Designing such a system, in particular developing the software for the robots contained in it, is a difficult and error-prone task. We developed a formal scenario-based modeling method which supports engineers in this task. Using short, intuitive scenarios engineers can express requirements, desired behavior, and assumptions made about the system's environment. These models can be created early in the design process and enable simulation as well as an automated formal analysis of the system and its components. Scenario-based models can drive the execution at runtime or can be used to generate executable code, e.g., programmable logic controller code. In this paper we describe how to use our scenario-based approach to not only improve the quality of a system through formal methods, but also how to reduce the manual implementation effort by generating executable PLC code.

Keywords: Code generation · Robot · Scenario · GR(1) specification

1 Introduction

Robots are found in many domains, e.g., manufacturing, transportation, or health care. Especially in manufacturing they are ubiquitous. Modern production systems implement complex processes, often requiring the cooperation of many robots to achieve their desired goals. Each robot may even be involved in several concurrent processes, making the design of its behavior a difficult and error-prone task. The robot has to react to a multitude of events, both external events, e.g., sensor inputs, and internal events, i.e., actions of other robots in the system. The inherent complexities of modern manufacturing processes make it difficult to develop robot software which is free of defects, that is, which makes the robot act or react properly under all possible circumstances. The specification, from which an implementation is derived, may be inconsistent and the

manual implementation thereof itself may introduce further defects. The task of designing such systems becomes even more difficult when considering non-functional requirements such as reducing the system’s energy consumption.

We developed a formal, yet still intuitive scenario-based specification approach to support engineers with the difficult design of such systems. Our approach uses short scenarios to model *guarantees* (goals, requirements, or desired behavior) and *assumptions* made about the environment. Scenarios are sequences of events, similar to how engineers describe requirements to each other, e.g., “When A and B happen, then component C_1 must do D , followed by C_2 doing E .” These sequences are used to intuitively describe when events or actions may, must, or must not occur [1, 14]. The formal nature of scenario-based specifications allows applying powerful analysis techniques early in the design process. Through simulation and controller synthesis, which, if successful, can prove that the requirements defined in the specification are consistent, defects can be found and fixed early during development. The same techniques used for simulation can be used to directly execute a specification at runtime [15] and the techniques used for controller synthesis can be used to automatically generate executable code. This reduces manual implementation effort significantly, thus mitigating some of the cost of writing a formal specification. With mature enough tool support, an overall reduction in development costs could even be achieved.

The contribution of this paper is an approach for generating executable code for Programmable Logic Controllers (PLCs) from aforementioned scenario-based specifications. This enables engineers to use formal methods such as checking if all requirements are consistent to ensure the correctness of the specification and to generate code which is correct by construction. A PLC program must handle two concerns: (1) it must correctly decide when to perform which atomic action, e.g., when to move which robot arm to which location, and (2) it must implement each atomic action, e.g., moving a specific robot arm to a specific location. Our approach generates code handling the first concern, leaving only the manual implementation of atomic actions to engineers. From the point of view of Model Driven Architecture [20], a scenario-based specification would be a Platform Independent Model of a system and the generated PLC code, after an implementation of each atomic action has been added, would be a Platform Specific Model of the same system. The latter can then be used directly as the software for an actual physical version of the specified system.

The remainder of this paper is structured as follows. Section 2 introduces an example used for explanation and discussion throughout the paper. Sections 3 and 4 introduce scenario-based modeling and controller synthesis. Section 5 builds on these foundations to describe how to generate PLC code from such a controller. The paper finishes with related work and a conclusion in Sects. 6 and 7.

2 Example

To explain and discuss our approach we use a production system example, shown in Fig. 1. It models a typical manufacturing process. Blank work items arrive via

a feed belt, which has a sensor telling a controller about the arrival of new work items. These blanks are then picked up by a robot arm and put into a press, which will press the blanks into useful items. These pressed items are then picked up by another robot arm which will put the items on a deposit belt which will transport the items to their next destination.

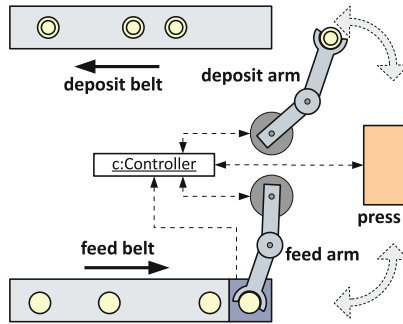


Fig. 1. A production system consisting of two robot arms, each adjacent to a conveyor belt, a press, and a software-based controller sending instructions to other components as well as processing their sensor inputs.

The specification we use for this example models the following guarantees **G** and assumptions **A**:

- G1** When a new blank arrives, the feed arm must pick it up when possible.
- G2** After picking up an item, the feed arm must move to the press, release the item into the press (when the press is ready), and finally move back to the feed belt.
- G3** When an item is put into the press, the press must start pressing.
- G4** When the press finishes, the deposit arm must pick the pressed item up when possible.
- G5** After picking up an item, the deposit arm must move to the deposit belt, release the item onto the deposit belt, and finally move back to the press.
- A1** The feed arm is able to pick up every blank before the next one arrives.
- A2** After being instructed to press an item, the press will eventually finish.
- A3** After a robot arm is instructed to move to a new location, it will eventually arrive at the new location.
- A4** After a robot arm is instructed to pick up an item, it will eventually pick up that item.
- A5** After a robot arm is instructed to release an item, it will eventually release that item.

Guarantees **G1–G5** define the system’s desired behavior and requirements it must fulfill as described at the beginning of this chapter. They also include additional conditions, e.g., “[...] the feed arm must pick it up *when possible*.” in

G1. These conditions express additional structural conditions required to fulfill certain goals. In the same example, **G1**, a new blank may arrive while the feed arm is still delivering the previous blank or is still on its way back to the feed belt. In these cases the feed arm must only be instructed to pick up the newly arrived blank when it is back at the press.

The assumptions specify what the engineers assume is true about the environment the system will operate in. As an example, **A1** specifies that the feed arm is able to pick up arriving blanks more frequently than the frequency of arrival of new blanks. This assumption implies that no queue of unprocessed blanks forms at the feed belt. Assumptions **A2–A5** specify that robot arms and the press will eventually finish their tasks after being instructed to perform a certain action. These assumptions are actually important to ensure that the specification is realizable, since they basically specify that system is operating normally, i.e., the components are working as intended. A correctly working system only needs to fulfill its guarantees as long as all assumptions hold.

3 Scenario-Based Modeling

In this section we introduce our scenario-based modeling approach, which we use to write formal specifications. It is based on a DSL we developed for modeling scenarios, called the Scenario Modeling Language (SML) [16].

SML offers engineers an easy to use way to write formal, scenario-based specifications. It is a text-based variant of Life Sequence Charts [10, 18], offering a similar feature set with a few extensions. Listing 1 shows the specification of our production system example. Comments next to each scenario indicate which guarantee or assumption they represent. A scenario-based specification also consist of a class model, called domain model, describing the different components of the system and an instance thereof, an object model. The object model contains a concrete instance for every physical component of the specified system.

A specification references a domain model (line 1) and has a name (line 2). In our example the domain model contains classes such as *RobotArm* and *Press*. These classes model each component type’s attributes and possible events it can receive. Events can be either actions it should perform or sensor events it may be notified of. Our production system example includes events such as a *RobotArm* being told to pick up an item or the *Controller* being notified of the arrival of a new blank. The specification defines which components are software-controllable (line 4) with all other classes automatically being interpreted as *uncontrollable*, also called environment-controllable. Non-spontaneous events (lines 5–13) are events which cannot occur unless enabled, e.g., the event *pressingFinished* cannot occur unless assumption **A2** is active (lines 51–54) and is in a state in which the second line is expected next. Other events, sent by uncontrollable objects and being the initializing event of a scenario (e.g., *blankArrived*) can occur spontaneously. This then triggers the creation of an instance of a scenario called an *active scenario*. Active scenarios have one or more references to events they expected next, called *enabled events*. When *PressEventuallyFinishes* (lines

```

1  import "../model/productioncell.ecore"
2  specification ProductioncellSpecification {
3      domain productioncell
4      controllable { Controller }
5      non-spontaneous events {
6          Controller.pickedUpItem
7          Controller.arrivedAt
8          Controller.releasedItem
9          Controller.pressingFinished
10         RobotArm.setCarriesItem
11         RobotArm.setLocation
12         Press.setHasItem
13     }
14     collaboration FeedBeltBehavior {
15         static role Controller controller
16         static role ConveyorBelt feedBelt
17         static role RobotArm feedArm
18         static role Press press
19
20         guarantee scenario BlankArrives { // G1
21             feedBelt -> controller.blankArrived()
22             wait [feedArm.location == feedBelt && !feedArm.carriesItem]
23             urgent controller -> feedArm.pickUp()
24         }
25         guarantee scenario ArmDeliversItemToPress { // G2
26             feedArm -> controller.pickedUpItem()
27             urgent controller -> feedArm.moveTo(press)
28             feedArm -> controller.arrivedAt(press)
29             wait [!press.hasItem]
30             urgent controller -> feedArm.releaseItem()
31             feedArm -> controller.releasedItem()
32             urgent controller -> feedArm.moveTo(feedBelt)
33         }
34         ... // new blanks are picked up before next one arrives (A1)
35     }
36     collaboration PressBehavior {
37         static role Controller controller
38         static role RobotArm feedArm
39         static role RobotArm depositArm
40         static role Press press
41
42         guarantee scenario PressStartsPressing { // G3
43             feedArm -> controller.releasedItem()
44             urgent controller -> press.startPressing()
45         }
46         guarantee scenario PickupPressedItem { // G4
47             press -> controller.pressingFinished()
48             wait [depositArm.location == press && !depositArm.carriesItem]
49             urgent controller -> depositArm.pickUp()
50         }
51         assumption scenario PressEventuallyFinishes { // A2
52             controller -> press.startPressing()
53             strict eventually press -> controller.pressingFinished()
54         }
55     }
56     collaboration DepositBeltBehavior {
57         ... // deposit arm transports pressed items (G5); similar to G2
58     }
59     collaboration RobotArmBehavior {
60         dynamic role Controller controller
61         dynamic role RobotArm arm
62         dynamic role Location targetLocation
63         static role Press press
64
65         assumption scenario ArmMovesToLocation { // A3
66             controller -> arm.moveTo(bind targetLocation)
67             strict eventually arm -> controller.arrivedAt(targetLocation)
68             strict committed arm -> arm.setLocation(targetLocation)
69         }
70         ... /* arm picks up item (A4) and arm releases item (A5); both similar
71            to A3 */
72     }
73 }

```

Listing 1. Excerpt of a specification for our production system example; some scenarios have been omitted for brevity

51–54) is activated by a *startPressing* event, it will point to line 53, indicating that this scenario waits for a *pressingFinished* event. When an event enabled in an active scenario occurs, the reference to this enabled event advances to the next event. When all references advance past the last event in a scenario, it terminates.

Roles (e.g., lines 15–18) are used similarly to lifelines in sequence diagrams. Static roles are bound when the system is initialized and dynamic roles are bound when an active scenario is created. Binding a role means assigning an object from the object model to this role. The abstraction through roles allows reusing the same specification for different object models modeling different configurations of the same type of system, e.g., production systems with varying numbers of robots. In lines 60–69 the use of dynamic roles is shown. Any object of the proper class from the object model can be bound to these roles. As an example, when an object of class *Controller* sends the event *moveTo* to an object of class *RobotArm*, an active instance of the scenario *ArmMovesToLocation* (lines 65–69) is created. In this active scenario, the role *controller* is played by the object which sent the initial event and the role *arm* is played by the object which received the event. Dynamic roles can even be bound to parameters (line 66) or to an object referenced by an object already bound to a role (not shown). Multiple copies of the same scenario with different role bindings can be active concurrently.

Events use different keywords to enforce *liveness* and *safety conditions*. Events flagged as *committed*, *urgent*, or *eventually* must not be enabled forever. Committed and urgent events must occur immediately, allowing only other committed or urgent events to occur beforehand. Committed events take priority over urgent events. An event which must occur eventually can occur at an arbitrary time in the future, i.e., the system can choose to wait. *Strict* events enforce a strict order. Events which occur out of order generally terminate a scenario early by *interrupting* it. If line 22 in an active scenario is enabled and *blankArrived* occurs (line 21; same active scenario), this active scenario is interrupted. However, if at least one enabled event is strict, an interruption causes a safety violation instead. Safety violations must never occur.

Additional keywords offer flow control. *Wait* is used to wait for a certain condition to be satisfied before the next message is enabled. The keywords *interrupt* and *violation* can be used to specify conditions, which are checked when the event becomes enabled and may cause an interruption or a safety violation. If the condition is not satisfied, the next event is immediately enabled instead. Furthermore, there are *while* (repeat an event sequence while a condition holds), *alternative* (branching within a scenario), and *parallel* (concurrent event sequences). *Collaborations* are used to group scenarios together and do not have any semantic implications beyond providing a scope for roles.

We implemented SML and algorithms for simulating and analyzing SML specifications as a collection of ECLIPSE plug-ins called SCENARIOTOOLS. We use the Eclipse Modeling Framework (EMF) [27] and leverages this to integrate other powerful tools such as OCL [30] and Henshin [3]. This enables engineers to enhance SML specification with tools they already are familiar with while

still being able to use SCENARIOTOOLS' simulation and analysis features, e.g., checking if a specification is realizable.

4 Controller Synthesis

In this section we give an overview of how controller synthesis works. We briefly explain the play-out algorithm, which gives our specifications execution semantics used for simulation, analysis, and controller synthesis, and how it induces a state space. Furthermore, we briefly explain controller synthesis, that is, generating a strategy for the system to behave such that it fulfills a given specification.

4.1 Play-Out

The play-out algorithm [18,19] defines how scenarios can be interwoven into valid event sequences. Basically, the algorithm waits for the environment to choose an event, activates and progress scenarios accordingly, and then picks a reaction which is valid according to all active scenarios. Environment events can either be spontaneous events or enabled non-spontaneous events. They are events sent by uncontrollable objects. When at least one system event with a liveness condition, e.g., *urgent*, sent by a controllable object is enabled, play-out will pick one of these events. It honors particular priorities such as picking committed events first. In case all such events are flagged as *eventually*, the algorithm may also choose to wait for further environment events. Events are considered to be blocked if they would directly lead to a safety violation due to the strictness of an enabled event. The play-out algorithm never picks blocked events. A sequence of events sent by system objects enclosed by one environment event on either end is called a *super step*.

For any given set of scenarios and a given object model the play-out algorithm generally has multiple valid events to choose from at any point. It is non-deterministic. This property induces a state space or graph as shown in Fig. 2, an excerpt of the graph of our production system example (cf. Sects. 2 and 3). Each node represents a *state*, characterized by its active scenarios and the attribute values of all objects. Every edge/transition represents an event. While the edge labels in Fig. 2 seem to reference roles from the SML specification, they actually reference objects from the object model. The events in such a state graph are always concrete events sent from one object to another object using concrete parameter values (if applicable).

Such a state space is actually a game graph. Each state is either controllable by the system (= has only controllable outgoing transitions) or by the environment (= has only uncontrollable outgoing transitions). These two players play against each other. The system tries to fulfill its guarantees infinitely often given that the assumptions hold. More precisely, for every guarantee, it always tries to reach states in which no liveness condition must be fulfilled and to reach them via a sequence of events which do not cause a safety violation. The environment aims for the opposite. It tries to fulfill all assumptions the same way the system

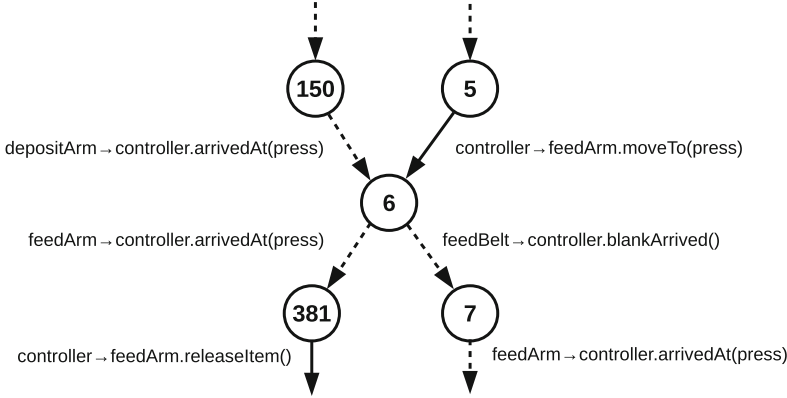


Fig. 2. Excerpt of a game graph induced by our example specification. Controllable/system events are represented by solid arrows. Uncontrollable events are represented by dashed arrows. *Set*-events, e.g., *setLocation*, have been omitted for brevity

fulfills the guarantees, but at the same time it tries to force the system to violate at least one of the guarantees. This type of game is called a *GR(1) game*. We impose an additional goal on the system, in particular we enforce the condition that each super step must be finite to ensure that the system will eventually be able to react to external events from the environment again.

4.2 Synthesis

Our controller synthesis is an implementation of Chatterjee’s attractor-based General Reactivity of rank 1 (GR(1)) game solving algorithm [9]. A GR(1) condition is based on assumptions and guarantees. Formally, as Linear Temporal Logic [25] formula, it is

$$\left(\bigwedge_i \square \diamond a_i \right) \implies \left(\bigwedge_j \square \diamond g_j \right) \tag{1}$$

with a_i = “assumption i is satisfied” and g_j = “guarantee j is satisfied”. Informally, this formula is true iff at least one assumption can only be fulfilled finitely often (i.e., goal states of this assumption are only visited a finite number of times in any infinite execution of the system) or all guarantees can be fulfilled infinitely often.

We map our specifications to a GR(1) condition by mapping active assumption scenarios to assumptions a_i and by mapping active guarantee scenarios to guarantees g_j . The goal states a_i of an active assumption scenario Sc are all those states in which Sc has no liveness condition to fulfill and has never been violated (tracked via a Boolean flag). Guarantee scenarios are mapped analogously. Additionally, we introduce an extra guarantee whose goal states are all

environment controlled states to ensure that all super steps are finite for well-separated specifications. In a well-separated specification [24], the system cannot force the environment into a violation of the assumptions by any action it takes. Well-separation is a desirable property of a good specification.

Chatterjee's aforementioned game solving algorithm uses the assumptions' and guarantees' goal states to calculate *attractors*. Attractors of a condition are all states from which a player can guarantee reaching a goal state of this condition. A system attractor of g_j is a state from which the system can ensure to visit a goal state of g_j regardless of the environment's behavior. Chatterjee's algorithm iteratively removes environment dominions from the game graph. Environment dominions are subsets of the game graph in which the environment can fulfill all assumptions but the system is unable to fulfill at least one of the guarantees. Environment dominions are identified by finding states which are not system attractors for at least one g_j . Using the environment attractors of all a_i , Chatterjee's algorithm determines if the environment can fulfill all assumptions in the subgraph defined by the non-attractor states of aforementioned g_j . These iterations are performed until the game graph cannot be reduced further.

The states retained after the algorithm finishes are called *winning states*. They contain a strategy in which the system can guarantee to fulfill the GR(1) condition defined by all assumptions and guarantees. If the initial state of the game graph is a winning state, the specification is *realizable*, i.e., the requirements and behavior defined by the scenarios are consistent. Using the same attractor approach, we can extract a *strategy* (also: *controller*) from the winning states. A strategy is similar to a game graph but contains exactly one outgoing transition for each controllable state (Fig. 2 happens to be a strategy). It deterministically specifies what the system must do for any valid environment. These strategies serve as the basis for generating Structured Text to execute on a PLC.

5 Generating Executable Code

In this section we describe how to generate Structured Text from a synthesized controller which is correct by construction. A synthesized controller contains some events which are only necessary for defining and checking a GR(1) condition but which serve no purpose in the generated PLC code. Thus, we explain a pre-processing step of the controller to reduce it to events of interest for code generation. After that, we describe how to generate executable PLC and finish the section with a discussion of possible extensions to our approach.

5.1 Pre-processing the Controller

Figure 3 shows an excerpt of a synthesized controller including a *setLocation* event which is required to be able to express conditions such as the *wait* condition in line 48 of Listing 1. However, this event is not useful for code generation and should be removed, as shown in Fig. 2. In general, expert knowledge of the domain is necessary to identify events to remove and thus an engineer should be

able to provide a list of such events. A tool can still provide helpful suggestions for removal based on heuristics, though. We propose two heuristics, (1) events sent by uncontrollable objects to other uncontrollable objects, and (2) *set*-events. Either of these two heuristics would be sufficient to propose the proper list of events to remove to the engineer in our example specification. When removing events, transitions have to be updated, such as the outgoing transition of state 150 in Fig. 3 which must point directly to state 6 after the removal of 151, which is no longer necessary after removing *setLocation*(***).

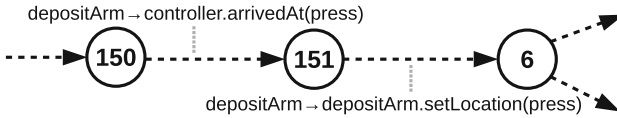


Fig. 3. Variant of Fig. 2 including a *setLocation* event previously omitted.

In Structured Text, components are controlled by setting the appropriate input attributes of function blocks, e.g., a block representing a specific robot arm of the system, and waiting for the output attributes to be set to values signaling that the desired action has been performed. The paradigm is: a component is instructed to do something (setting of input attributes) and it signals when it is done (setting of output attributes). In our approach, we adopt this paradigm by having the engineer define event pairs which correspond to “do X” and “X is done”. Such a pair is shown in Fig. 2: *moveTo*(*press*) (transition from 5 to 6) and *arrivedAt*(*press*) (transition from 6 to 381; also outgoing transition of 7). These event pairs are characterized by a controllable object *S* (here: *controller*) sending an event to an uncontrollable object *E* (here: *feedArm*), instructing *E* to perform an action (here: *moveTo*(*press*)). Later, *E* signals back to *S* that it is now done performing this action. Again, heuristics can be used to support the engineer in defining these pairs. We observed that these pairs often occur in adjacent lines in scenarios, e.g., lines 27–28 and 30–31 in Listing 1. These event pairs are necessary in the next step, the actual code generation.

5.2 Generating Structured Text

We use the pre-processed controller and event pair definitions provided by the engineer to generate Structured Text which is executable on PLCs. For simplicity, we assume that there is exactly one controllable object in the system, e.g., the controller shown in the center of Fig. 1. Our generated code consists of multiple state machines. We translate the pre-processed controller to one state machine representing the controllable object. We call this the *primary* state machine, as it governs the whole process: it tells each component, via the other state machines, when to perform which action. We furthermore generate one state machine for each uncontrollable object which receives events, i.e., represent components having to perform an action. These state machines, called *secondary* state machines,

```

1  CASE controllerState OF // primary state machine
2      0:
3          // idle
4      1:
5          ... // omitted for brevity
6      5:
7          feedArmState := 1;
8          controllerState := 6;
9      6:
10         IF feedBelt_controller_blankArrived THEN
11             feedBelt_controller_blankArrived := FALSE;
12             controllerState := 7;
13         ELSIF feedArm_controller_arrivedAt_press THEN
14             feedArm_controller_arrivedAt_press := FALSE;
15             controllerState := 381;
16         END_IF
17     7:
18         ... // omitted for brevity
19 END_CASE

```

Listing 2. Generated PLC code (Structured Text) of the primary state machine

are much simpler. They consist of an idle state, which is their initial state, and one additional state for each action that must be performed. Listings 2 and 3 show examples of the generated code.

Events sent by uncontrollable objects are mapped to Boolean variables, e.g., *feedBelt_controller_blankArrived* which corresponds to the sensor event triggered by the arrival of a new blank item. These variables are used by the primary state machine to decide when to switch to which state (lines 10–16 in Listing 2). This state machine instructs the secondary state machines to perform actions as called for by the synthesized controller, e.g., lines 7–8 correspond to the transition from state 5 to 6 in Fig. 2. The previously defined event pairs are used to generate this code. Based on the knowledge that *controller* → *feedArm.moveTo(press)* and *feedArm* → *controller.arrivedAt(press)* are a pair, line 7 can be generated to instruct the feed arm’s state machine (Listing 3) to switch to the proper state to perform this action. The same pair definition is used to generate line 9 in the secondary state machine, in which the feed arm informs the controller via a Boolean variable that is done performing the desired action. This separation into primary and secondary state machines allows any arbitrary combination of actions to be performed concurrently by different components.

Separating the generated state machines into different code files has proven to be a good practice when regeneration of the PLC code is a concern. By keeping state machines separate and the order of states in the secondary state machines deterministic and consistent, only the (fully generated) primary state machine has to be replaced after regenerating the PLC code. More elaborate changes


```

1  CASE feedArmState OF // secondary state machine for feed
   arm
2      0:
3          // idle
4      1:
5          // controller->feedArm.moveTo(press)
6          feedArmFB.xMoveRelExecute := TRUE; // perform
           example action
7          IF feedArmFB.xFunDone THEN // is example action
           done?
8              feedArmFB.xMoveRelExecute := FALSE; // clean-
           up after example action
9              feedArm_controller_arrivedAt_press := TRUE;
10             feedArmState := 0;
11         END_IF
12     2:
13         // controller->feedArm.moveTo(feedBelt)
14         ... // omitted for brevity
15 END_CASE

```

Listing 3. Generated PLC code (Structured Text) of a secondary state machine

to the model, such as adding or removing actions components must perform, require some manual migration effort when regenerating code.

The primary state machine is fully generated and does not need to be modified. The secondary state machines are however actually only stubs after generation. Listing 3 shows an example after an engineer manually added the code in lines 6 and 8 and the condition in line 7. In general, after generating the Structured Text from a synthesized controller, each state in the secondary state machines contains some boiler plate code, in particular the if-statement with an empty condition but a body that already sets the appropriate Boolean and state variables (lines 9–10), and some comments telling the engineer which atomic action should be performed in this state. These stubs can easily be extended by an engineer by setting and checking the inputs and outputs of the appropriate function block. The proper function block definitions, as well as any initializations, have to be added manually as they are platform-specific. Additionally, code for checking sensor events which are not part of an event pair, e.g., when to set *feedBelt_controller_blankArrived* to *TRUE*, has to be added manually.

When generating PLC code from Listing 1 and using rotations at varying directions and speeds of a single axis (one for each component, i.e., both robot arms and the press) to represent actions such as movement or picking up a work item, 59 lines of code had to be written manually, 9 lines of code had to be modified (conditions which check whether an action has been performed successfully), and 1355 lines of code were generated automatically. While this is not an exhaustive evaluation, these numbers already point towards a significant

reduction in the required manual implementation effort. In particular, the complex interleaving of concurrent actions and events is fully generated.

5.3 Extensions

We assumed that there is only one controllable object in the system. As an extension to support multiple controllable objects, i.e., multiple software controllers, we are looking into algorithms to create multiple distributed controllers which automatically synchronize with each other when necessary.

Event pairs are defined during pre-processing. This implies that only the success case, i.e., the action can actually be performed, can be modeled. Instead, defining a mapping from controllable events (instructions) to sets of uncontrollable events (outcomes of the instructions) can easily rectify this. Different outcomes for each action can be defined and the specification can include appropriate reactions for each possible outcome.

By including checks of the Boolean variables of environment events, which are not expected to occur in a given state of the primary state machine, violations of the assumptions can be detected. These could be used to put the system into an emergency state which performs a shut down procedure.

6 Related Work

There exists previous work on synthesizing controllers from LSC/SML-style scenarios [6, 8, 17, 29], and other forms of scenarios [22, 31]. Most of these approaches produce finite state controllers or state machines as output, from which code can be generated. Some consider code generation from such synthesized controllers in particular for robotics/embedded applications [4, 21].

The novelty of our synthesis procedure w.r.t. to the above is, first, that it supports scenario-based specifications with a greater expressive power—assume/guarantee specifications with multiple liveness objectives (GR(1)). Second, we describe a scenario-based modeling and code generation methodology that specifically targets the typical structure and nature of PLC software.

There is work on generating PLC code from state machines [26] or Petri nets [12, 28], and formal methods are used also for verifying PLC code [5, 13].

Other work considers synthesis and code generation, some specifically for robotics applications, based on temporal logic specifications such as LTL and its GR(1) fragment [2, 7, 11, 23]. In contrast to temporal logics based approaches, LSCs/SML aim to provide a more intuitive language that is easier to use.

In previous own work, we considered the direct execution of SML specifications as *scenarios@run.time* [15]. Here, the scenarios are executed without the prior synthesis of a finite-state controller. Such an approach has advantages and disadvantages. For example, the prior synthesis does not only detect specification flaws, but a synthesized controller can also contain the solution for resolving issues related with under-specification. On the other hand, controller synthesis, due to its computational complexity, may not be possible for larger specifications, in which case direct execution is a valuable option.

7 Conclusion

In this paper we presented an approach for generating Structured Text executable on PLCs commonly found in the industry. We generate this code from scenario-based specifications written in an intuitive DSL we developed. Using this DSL, called Scenario Modeling Language (SML), engineers can easily define requirements, desired behavior, and environment assumptions of a system. These are defined in the form of assumption and guarantee scenarios, which have to be fulfilled infinitely often, i.e., SML specifications express GR(1) conditions, giving engineers a powerful class of conditions to express their goals in. The generated code, which is correct by construction, uses multiple state machines to separate the decision “when to perform which atomic action” from the implementation of each atomic action. After code generation, engineers only need to implement the atomic actions, with their complex interleaving into an implementation of the desired process having already been generated.

Acknowledgment. This research is funded by the DFG project EffiSynth.

References

1. Alexandron, G., Armoni, M., Gordon, M., Harel, D.: Scenario-based programming: reducing the cognitive load, fostering abstract thinking. In: Proceedings of the 36th International Conference on Software Engineering (ICSE), pp. 311–320 (2014)
2. Alur, R., Moarref, S., Topcu, U.: Compositional synthesis of reactive controllers for multi-agent systems. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 251–269. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_14
3. 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
4. Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Thiele, S., Schäfer, W., Meyer, M., Pohlmann, U., Priesterjahn, C., Tichy, M.: The MechatronicUML design method - process and language for platform-independent modeling (2014)
5. Biallas, S., Brauer, J., Kowalewski, S.: Arcade.PLC: a verification platform for programmable logic controllers. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 338–341, September 2012
6. Bontemps, Y., Heymans, P.: From live sequence charts to state machines and back: a guided tour. *IEEE Trans. Softw. Eng.* **31**(12), 999–1014 (2005)
7. Braberman, V., D’Ippolito, N., Piterman, N., Sykes, D., Uchitel, S.: Controller synthesis: from modelling to enactment. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, Piscataway, NJ, USA, pp. 1347–1350. IEEE Press (2013)
8. Brenner, C., Greenyer, J., Schäfer, W.: On-the-fly synthesis of scarcely synchronizing distributed controllers from scenario-based specifications. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 51–65. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_4

9. Chatterjee, K., Dvorák, W., Henzinger, M., Loitzenbauer, V.: Conditionally optimal algorithms for generalized büchi games. In: Faliszewski, P., Muscholl, A., Niedermeier, R. (eds.) 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 58, pp. 25:1–25:15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016)
10. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. *Formal Methods Syst. Des.* **19**, 45–80 (2001)
11. Ehlers, R., Raman, V.: **Slugs**: extensible GR(1) synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 333–339. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_18
12. Frey, G.: Automatic implementation of Petri net based control algorithms on PLC. In: Proceedings of the 2000 American Control Conference, ACC (IEEE Cat. No.00CH36334), vol. 4, pp. 2819–2823 (2000)
13. Frey, G., Litz, L.: Formal methods in PLC programming. In: 2000 IEEE International Conference on Systems, Man, and Cybernetics, vol. 4, pp. 2431–2436 (2000)
14. Gordon, M., Marron, A., Meerbaum-Salant, O.: Spaghetti for the main course? Observations on the naturalness of scenario-based programming. In: Proceedings of the 17th Conference on Innovation and Technology in Computer Science Education (ITICSE), pp. 198–203 (2012)
15. Greenyer, J., Gritzner, D., Gutjahr, T., Duente, T., Dulle, S., Deppe, F.D., Glade, N., Hilbich, M., Koenig, F., Luennemann, J., Prenner, N., Raetz, K., Schnelle, T., Singer, M., Tempelmeier, N., Voges, R.: Scenarios@run.time - distributed execution of specifications on IoT-connected robots. In: Proceedings of the 10th International Workshop on Models@Run.Time (MRT 2015), co-located with MODELS 2015. CEUR Workshop Proceedings (2015)
16. Greenyer, J., Gritzner, D., Katz, G., Marron, A.: Scenario-based modeling and synthesis for reactive systems with dynamic system structure in scenariotools. In: de Lara, J., Clarke, P.J., Sabetzadeh, M. (eds.) 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), vol. 1725, pp. 16–32. CEUR (2016)
17. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. *Found. Comput. Sci.* **13**(1), 5–51 (2002)
18. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003). <https://doi.org/10.1007/978-3-642-19029-2>
19. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: the play-in/play-out approach. *SoSyM* **2**, 82–107 (2003)
20. Kleppe, A.G., Warmer, J.B., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional, Boston (2003)
21. La Manna, V.P., Greenyer, J., Clun, D., Ghezzi, C.: Towards executing dynamically updating finite-state controllers on a robot system. In: Proceedings of the Seventh International Workshop on Modeling in Software Engineering, MiSE 2015, Piscataway, NJ, USA, pp. 42–47. IEEE Press (2015)
22. Liang, H., Dingel, J., Diskin, Z.: A comparative survey of scenario-based to state-based model synthesis approaches. In: Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, SCESM 2006, pp. 5–12. ACM, New York (2006)

23. Maoz, S., Ringert, J.O.: Synthesizing a lego forklift controller in GR(1): a case study. In: Proceedings of the 4th Workshop on Synthesis (SYNT), co-located with CAV 2015 (2015)
24. Maoz, S., Ringert, J.O.: On well-separation of GR(1) specifications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 362–372. ACM (2016)
25. Pnueli, A.: The temporal logic of programs. In: 1977 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
26. Sacha, K.: Automatic code generation for PLC controllers. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, pp. 303–316. Springer, Heidelberg (2005). https://doi.org/10.1007/11563228_23
27. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education, London (2008)
28. Thapa, D., Dangol, S., Wang, G.N.: Transformation from petri nets model to programmable logic controller using one-to-one mapping technique. In: International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC 2006), vol. 2, pp. 228–233, November 2005
29. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Softw. Eng.* **35**(3), 384–406 (2009)
30. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional, Boston (2003)
31. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proceedings of the 22nd International Conference on Software Engineering, ICSE 2000, pp. 314–323 (2000)

Evaluating a Graph Query Language for Human-Robot Interaction Data in Smart Environments

Norman Köster¹(✉), Sebastian Wrede^{1,2}, and Philipp Cimiano¹

¹ Cluster of Excellence Center in Cognitive Interactive Technology (CITEC),
Bielefeld University, Bielefeld, Germany

{nkoester,swrede,cimiano}@techfak.uni-bielefeld.de

² Research Institute for Cognition and Robotics (CoR-Lab), Bielefeld University,
Bielefeld, Germany

Abstract. Solutions for efficient querying of long-term human-robot interaction data require in-depth knowledge of the involved domains and represents a very difficult and error prone task due to the inherent (system) complexity. Developers require detailed knowledge with respect to the different underlying data schemata, semantic mappings, and, most importantly, the query language used by the storage system (e.g. SPARQL, SQL, or general-purpose language interfaces/APIs). While for instance database developers are familiar with technical aspects of query languages, application developers of interactive scenarios typically lack the specific knowledge to efficiently work with complex database management systems. Addressing this gap, in this paper we describe a model-driven software development based approach to create a long-term storage system to be employed in the domain of embodied interaction in smart environments (EISE). To support this, we created multiple domain specific languages using *Jetbrains MPS* to model the high level EISE domain, to represent the employed graph query language *Cypher* and to perform necessary model-to-model transformations. As main result, we present the *EISE Query-Designer*, a fully integrated workbench to facilitate data storage and retrieval by supporting and guiding developers in the query design process and allowing direct query execution without the need to have prior in-depth knowledge of the domain at hand. In this paper we report in detail on the study design, execution, first knowledge gained from our experiments, and lastly the lessons learned from the development process up to this point.

1 Introduction

Smart home technology is gaining more and more popularity and becomes increasingly widespread. The most prominent implementations target support for private households and are available in various complexities from a full system, such as a *KNX*¹ system or an *Apple Home Kit*², to rather simple personal

¹ <https://www.knx.org>.

² <https://www.apple.com/ios/home/>.

assistants, such as *Alexa*³ or the *Amazon Dash Button*. Beyond the deployment of smart home technology in private homes, one can observe an increased adoption in elderly care settings. Work in this area often further additionally incorporates personal robots to support humans in their daily living and provide an embodied interaction for them [1, 2]. Our laboratory setup, the Cognitive Service Robotics Apartment (CSRA), provides such an embodied interactive smart environment (c.f. Fig. 1) [3]. It is a fully equipped apartment that is extended with various sensors (e.g. depth sensors, cameras, capacitive floor, light/temperature sensors) and actuators (e.g. screens, colourable lights, audio). Besides two virtual agents which allow users to interact verbally, there is a bi-manual mobile robot named *Floka* operating autonomously within the apartment that allows embodied interaction.

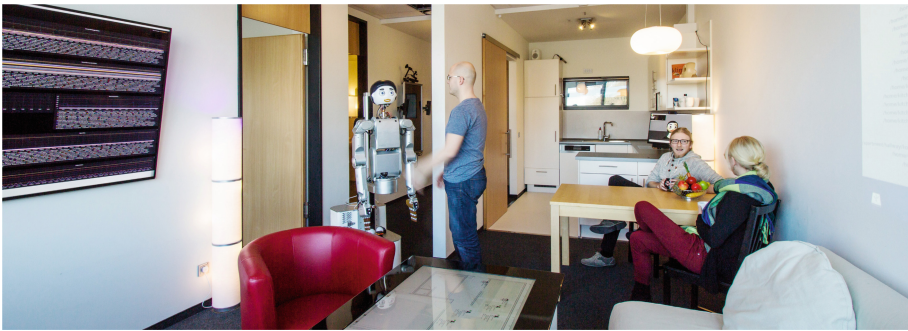


Fig. 1. An example picture from within the CSRA from the living room showing an interaction with Floka in the apartment.

The Cognitive Service Robotics Apartment is used to develop new smart home technology systems as well as to study human-machine interaction in the context of smart environments [4]. One central aspect of the CSRA project is concerned with interaction relevant data/knowledge storage, retrieval and transfer between agents and/or robots. Additionally to the robot eco-system, this environment consists of multiple devices that provide data as well as software packages performing more complex perception including person tracking, and situation recognition. These and other components employ different and often multiple protocols, such as KNX, RSB, ROS, or REST.

In this context, application developers are responsible to create interactive scenarios and therefore require access to current and previous sensor, actuator, and aggregated/derived data and knowledge available. For example, when designing a greeting scenario, the application developer can incorporate knowledge of previous interactions. Possible questions would be ‘Have I seen this person before?’, ‘What is this person’s name?’, ‘What were topics of our last conversation?’, or ‘Does this person know where to find drinks in the apartment?’.

³ <https://developer.amazon.com/alexa>.

Given the aforementioned number of sensors and actuators employed, there is hence a large amount of data comprising of various modalities available which needs to be stored in a fashion so that data can be readily queried by application developers within interactive scenarios. Providing such storage and retrieval functionality over a longer-term while supporting easy access is a challenging endeavour. The developers creating applications in this domain need to have detailed knowledge with respect to the different underlying data schemata, semantic mappings, etc. In addition, extensive knowledge about the query language used by the storage system is a requirement and depends on the chosen storage solution. While database developers are familiar with the required details, application developers typically lack the specific knowledge to efficiently work with complex database management systems.

Generally, there are a number of advantages in applying model-driven software development (MDS) techniques in robotics. The advantages include the ability to generate code automatically, analysis/checking while programming/writing statements in the provided language as well as platform independence [5]. A core advantage for the application in the EISE domain is the fact that developers receive feedback on the queries they are writing at design time rather than after execution. We thus decided to follow a model-driven approach to create a querying environment to be employed in the domain of embodied interaction in smart environments with the primary goal to support and ease the task of querying the data. As a basis for the approach, we previously presented an ontology for modeling human machine interaction in smart environments [6]. Using this declarative specification of the EISE domain as a starting point, we derived multiple external domain specific languages (DSLs). Generally, there are clear benefits of DSL applications, such as increased productivity, quality, validation and verification, and lastly productive tooling [5]. In our application domain, the latter can especially provide developers with helpful functionality such as static analysis, code completion, visualisations, or debugging at design time.

In particular, our model-driven approach provides generated artifacts (e.g. a specific IDE and access APIs) to support the query design and execution for application developers. One central tool in this context is the *EISE Query-Designer*, a full IDE that allows developers to design and execute queries against the database setup within the fully integrated CSRA environment. This tool is the result of a composition of multiple individual models designed independently following a model-driven software development approach. While such tools have a number of advantages for developers (e.g. reduction of complexity, static analysis, etc.), there is still a need for proper evaluation of the usefulness and usability for developers. In this paper we hence focus to report on our approach for evaluating and quantifying the advantage of this specific IDE.

The remainder of this paper is structured as follows. In Sect. 2 we briefly describe the created and reused languages and solutions as well as their composition that allows to produce a standalone IDE. In Sect. 3 we present a detailed description of the study design and implementation we plan to use to evaluate

our work and present preliminary results from a small pilot study in Sect. 4. Section 5 then discusses the results and lessons learned during the development process. Lastly, after giving a short overview about related work in Sect. 6, we end with a brief conclusion in Sect. 7.

2 Language Modeling

In the following we briefly describe the modeling approach and present the resulting implementations. We chose *Jetbrains MPS* - one of the most feature rich integrated DSL development platforms - over other tools (e.g. Eclipse Xtext⁴) to implement our work due to reasons put forth by Voelter [7, 8]. The general design of languages is supported well within *MPS*, as it provides an integrated support for the development of structure, syntax, type systems, IDEs, and model-to-model transformations. The differentiation of *languages*, *solutions* (more specifically: runtime solutions and build solutions), and their interoperability allow for complex but yet flexible and easily extensible constructs. Within *MPS* a *language* allows the language designer to abstract form the domain at hand by modeling important concepts and their properties, while the final users can use so-called *solutions* to implement their user model. Further, *MPS* directly supports the generation of standalone IDEs and specific workbenches tailored to individual specific requirements. Another very important feature is the possibility to easily have multiple projections of our language(s) allowing us to design different views for various roles in the development cycle. For example, a database developer will have to make changes to the lowest levels of the modeling using read-write queries while an application developer only needs to execute simple read-only queries without write access to the database. This difference can be addressed by providing multiple projections within the same artifact or alternatively by generating different role-specific artifacts based on the individual needs. We make explicit use of language extension and language embedding (meant as a special case of language reuse) to model the domain in our framework (c.f. Fig. 2) [5]. From an architectural perspective, we hence separate the framework into three languages: (1) The *RelationalDomainDescription* (RDD) language, (2) the *Cypher* language, and (3) the *CypherRDD* language. We chose this level of abstraction as we intend to expand and add in further functionality to allow for example annotation and grounding for data types, time (intervals), as well as database back-ends.

Relational Domain Description (RDD)

The Relational Domain Description language supports the representation of an application domain as a graph by providing nodes and edges alongside their properties and according data types. The main reason to implement this meta language is the fact that we perceive the application domain description as a dynamic and changing process. When using a *MPS* language to model this sub-domain, domain experts would have to perform changes in

⁴ <http://eclipse.org/xtext>.

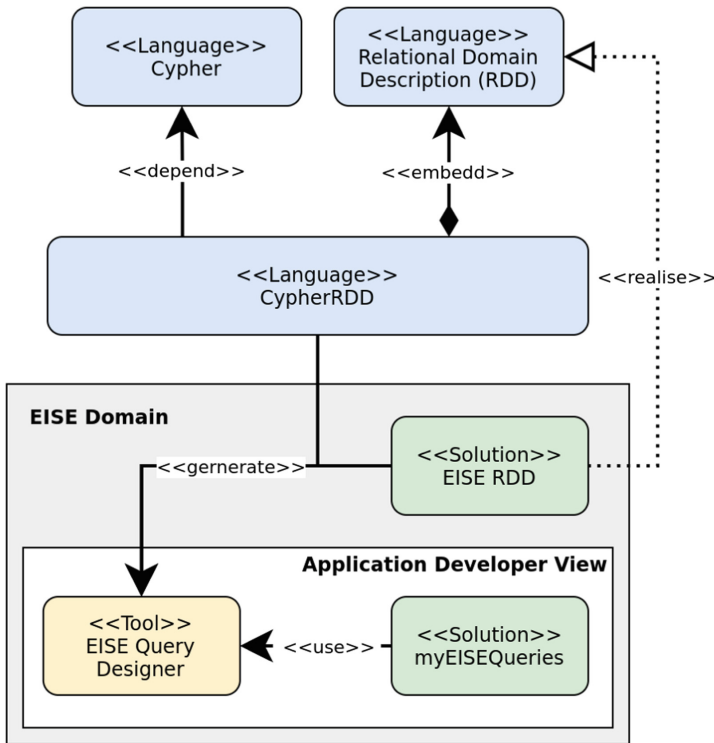


Fig. 2. Architectural overview of the individual *MPS* languages, solutions and their connections required to generate the artifacts for our user study. Central languages are Cypher, RelationalDomainDescription (RDD), and the composing CypherRDD.

this rather strict and complex environment. Defining this meta language allows us to model the application domain as a solution which can easily be modified while being easy to understand and requiring less detailed knowledge about language design using *MPS*. We used this language to realise the *EISE* solution which models the embodied interaction in smart environments domain.

Cypher

We decided to represent interaction relevant data as a graph and chose Neo4j⁵ as our database back end. One important feature is Neo4j’s query language Cypher, which is currently trying to gain further adoption with the *open-Cypher* initiative⁶ [9]. This language provides a good interface for non domain experts to abstract and formulate their queries. The Cypher language hence provides the Cypher graph query language as a *MPS* language. It adds all required concepts to provide the functionality to compose and execute queries

⁵ <http://neo4j.com>.

⁶ <http://www.opencypher.org>.

against a database. In order to realise this it uses a runtime solution that provides the official Java bindings of Cypher within MPS. Further, this language also allows to embed Cypher constructs within Java programs. An initial language was already available as an open source project⁷, which we adopted and extended where necessary.

CypherRDD

Describing the application domain as a solution of the RDD language requires us to create a composing language that allows to combine any RDD solution with the Cypher language. To provide this functionality, the CypherRDD language has a dependency to the Cypher language and embeds RDD concepts within it. The language itself is very simple and is mainly concerned with providing the correct scoping for individual concepts of each of the extended languages. As a result, a build solution can combine this language with any corresponding application domain description defined as a RDD solution to provide a custom IDE. For our study we hence created the EISE Query-Designer that uses the CypherRDD language and embeds the EISE solution concepts (as depicted in Fig. 2). Users can create their own solutions containing the designed queries with the help of this IDE. The projection presented to the user is based on a combination of the Cypher abstract syntax in which only the concepts of the used RDD solution are valid entries. This IDE provides the user with domain specific support and auto completion during the query design process. Further, quick-fixes for the query design are provided for common and mundane tasks. The created query can then be executed directly within the IDE via the provided Java integration and the results are presented to the user.

3 Workbench Evaluation

The evaluation of DSLs and IDEs represents a challenge as assessment of their advantages requires to analyse various properties. Especially due to their complexity, the evaluation of a full integrated workbench is not as straightforward and may require long-term observation of target users. Case studies which draw lessons learned are an option for evaluation - especially when the benefit is obvious and the user base is large [10]. A good alternative is to follow an iterative testing approach and focus on clearly defined metrics (such as lines of code or perceived usability).

To properly evaluate our approach, we therefore decided to conduct a full user study with potential application developers in an early phase of the development to be able to feedback the results into the development. We let the target audience use the EISE Query-Designer to solve several tasks and compare their performance against a group which uses a baseline default environment (i.e. the default Neo4j web interface). Our primary supporting hypothesis (**H1**) for this study is that programmers formulate queries of various complexities easier and

⁷ <https://github.com/rduga/Neo4jCypher>.

quicker when using the extended Cypher Query language embedded in a specific IDE compared to the normal condition. Secondary (**H2**), we expect programmers who use the provided tool to exhibit an improved learning curve when solving similar tasks during the study (even though they have to familiarise themselves with the tool first).

3.1 Study Design

We employ a between-group design with the following two distinct conditions (c.f. Fig. 3):

- (A) **Normal condition:** Participants have no IDE support. Instead they use the default web interface provided by Neo4j which will allow them to write plain text Cypher queries to solve the four sets. The comprehension tasks present queries to the participants using the default syntax highlighting in the web interface (c.f. Fig. 4(a))
- (B) **Extended condition:** Participants will use the EISE Query-Designer and benefit from the IDE support. The IDE provides a custom projection of the Cypher language and incorporates the EISE domain knowledge. Comprehension tasks are presented directly within the IDE and therefore use the custom projection. In the study setup, query results are displayed as a tabular listing - further representations such as a graph are planned (c.f. Fig. 4(b)).

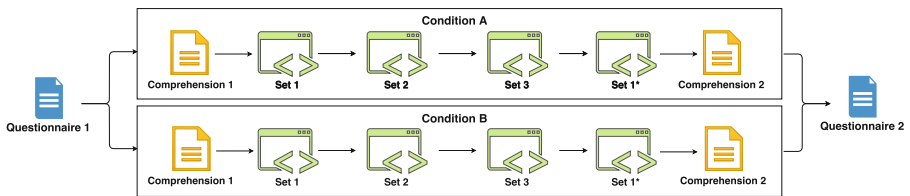
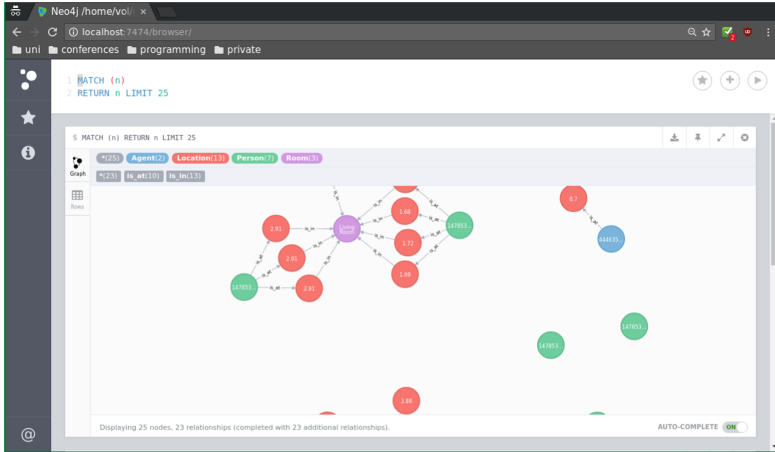


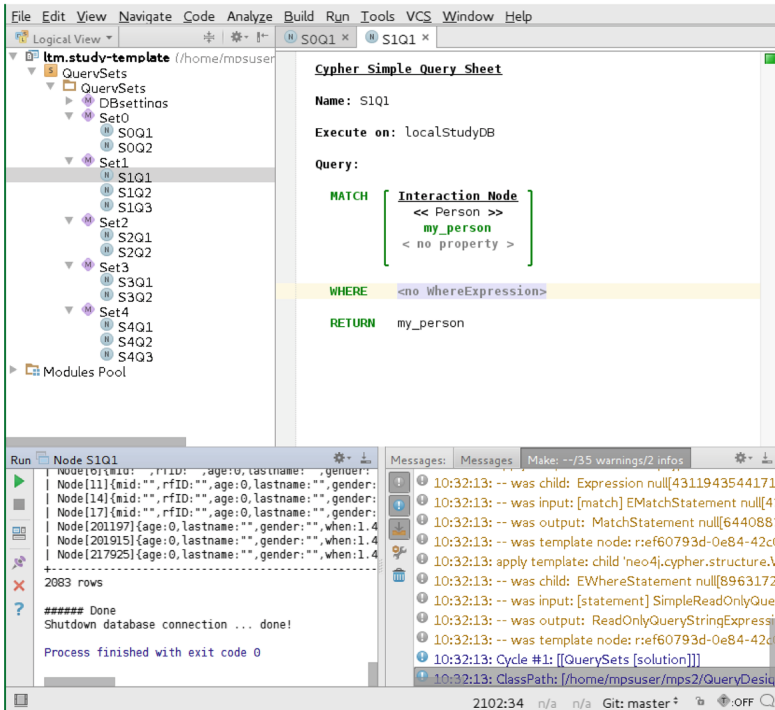
Fig. 3. We chose a between-group study design setup with two conditions: (A) users use the default Neo4j interface, and (B) users use the EISE Query-Designer. Both conditions have to create and execute the same queries (Set 1–Set 1*). Before and afterwards, users have to describe the meaning of queries presented to them in the comprehension task (Comprehension 1 and 2).

3.2 Tasks

First, the “Cypher Cheat Sheet” provides the participants with basic knowledge about the Cypher query language and its constructs relevant to compose the queries. This is the same across conditions. Second, a “Tool Sheet” is provided describing the basic usage of the tools used in the according condition (web interface or IDE) together with basic examples, shortcuts, language reference



(a) Screenshot of the Neo4j web interface which allows to write and execute the queries. Results can be inspected as a graph or table.



(b) Screenshot of the EISE Query-Designer which allows to write and execute the queries. Results can be inspected as a table at the bottom.

Fig. 4. Screenshot of the tooling used by each condition.

and explanations. Third and last, the “EISE Domain Sheet” contains a graph describing the overall schema of the prepared dataset and is identical across the conditions. It represents a simple visualisation of the EISE solution described in Sect. 2. Given this material, participants have to solve a total of 6 sets of tasks that are divided into two types: comprehension and work tasks. In the following we briefly explain each of the tasks in more detail.

Comprehension Tasks

Comprehension tasks present the participants with example queries in their environment of their condition to investigate their level of understanding of existing queries. The goal is to investigate the ability to read and interpret existing queries without prior domain knowledge of the underlying graph structure. Each set is presented before and after the work tasks (c.f. Fig. 2, Comprehension 1 and 2) to investigate our second hypothesis (**H2**).

Work Tasks

Work tasks are presented to the participants in natural language text and describe a question to be answered by a query (c.f. Fig. 5 for an example). The participants have to write a Cypher query based on this provided question. Each question is annotated with a time to give an estimate on the expected required effort. Based on pre-study tests we also provide textual hints as an additional support to elaborate on the query and avoid misunderstandings. Further, the expected result is also listed so that participants can easily spot correct and incorrect queries. In total, there are four sets of work tasks: Set 1 (3 questions), Set 2 (2 questions), Set 3 (2 questions), and Set 1* (3 questions). The difficulty rises from Set 1 to 3 and each set introduces new concepts of the Cypher language which the participants have to use to write successful queries. The last work task (Set 1*) is a modified Set 1 question group with the goal to quantify the expected user learning effects. There is also a Set 0 (omitted in the graphical representation) which is used to present the task-questionnaire procedure to the participants and to foreclose eventual execution errors. For each condition in this study, we use the same pre-populated EISE dataset which we gathered within the CSRA laboratory setup. This allows us to formulate a gold standard for each question against which results of the participants can be compared.

3.3 Participant Preconditions

With study participants having to actually use the query language and the according tools for each condition, they have to fulfill certain (rather demanding) criteria. We require participants to have a certain basic knowledge concerning databases and database access (e.g. SQL, SPARQL, etc.). However, they are not required to have strong programming skills as both conditions do not require to write any surrounding source code and allow direct query execution. Their understanding and knowledge about the tools (Neo4j and MPS), query language (Cypher) and overall domain (the EISE domain) for the experiment

Set 2 (S2)

Query 2 (S2Q2) [5 minutes]

How many conversations are in the database in which persons and agents were active together?

Hint:

1. Refers to the amount of conversations to which persons had an *involved_in* relationship and at the same time agents also have an *involved_in* relationship to.
2. Use multiple relationships within a MATCH clause (alternatively it is also possible to use multiple MATCH clauses).

Expected result: 243

Fig. 5. Exemplary work task as presented to the study participants.

should however be on a similar level and not differ significantly to allow us to draw conclusions for a representative group. We ensure this by adding according items to our questionnaires asking the participants about their knowledge about the involved elements.

3.4 Questionnaire

To assess the usability of our approach from a quantitative point of view, we asked users to fill in several questionnaires on a separate computer [11]. Between each set of tasks, participants have to answer the 6 item *Task Load Index* (TLX) to measure their cognitive load during each set [12]. Besides measuring the TLX metric, this also allows us to gather durations for each set of tasks independently from the condition. Once finished, the last questionnaire asks the user to fill in the *System Usability Scale* (SUS) as well as the *User Experience Questionnaire* (UEQ) in order to assess the tool usability [11, 13]. Further properties are recorded on the executing computer allowing us to investigate metrics such as time per task set, key strokes, correct and incorrect queries, etc. We also recorded the experiment using screen capture techniques in order to allow for further qualitative analysis of the experimental data. The targeted sample size is 15 to 20 participants per condition.

3.5 Expectations

With this given study setup we have certain expectations towards each group's performance. As participants in the extended condition (B) will use the IDE to compose queries, we expect them to perform better compared to the normal condition (A). We expect a higher accuracy and lower error rates due to syntax and error checking that is provided by the IDE. A related hypothesis we postulate is that the overall duration for each task and the keystrokes required

will be higher for the normal condition as the users receive fewer feedback and support when writing the queries. With the separation in multiple sets we expect participants of condition (B) to learn how to create queries faster - even though they will have to familiarise themselves with a more complex tool. We expect this familiarisation step to be a constant that will impact the initial sets but otherwise will not be present in later more difficult tasks.

Table 1. Average perceived usability of pilot study.

	Metric	Condition (A)	Condition (B)
SUS	Score	75	63.33
	Attractiveness	1.111	0.944
	Perspicuity	1.417	1.083
UEQ	Efficiency	0.917	0.833
	Dependability	1.500	0
	Stimulation	0.833	0.917
	Novelty	-0.083	0.667

4 Pilot Study Results

In a pilot study we applied the study design to a total of six participants (three per condition). Preliminary results of the user SUS and UEQ questionnaires are listed in Table 1. It shows that the SUS score for condition (B) ranks within an average level while condition (A) is slightly above average [14]. Further, we observed that for all UEQ questionnaire based usability metrics the web interface (condition (A)) scores slightly higher with stimulation and novelty being the exceptions. The pilot study results regarding the actual time the participants required per task set is depicted in Fig. 6. It shows that the baseline condition (A) allowed participants to finish easier tasks (Set 1, Set 2, and Set 1*) faster when compared to condition (B). However, the set with more difficult tasks (Set 3) demanded more time investment from participants in condition (A). An analogous result is observable for the cognitive load which participants experiences during each set (c.f. Fig. 7).

5 Discussion and Lessons Learned

Though the preliminary study had only few participants, its results shows recognisable trends. In the first two sets the participants in the baseline condition (condition A) were faster and experienced less cognitive load. We think this is due to the steep learning curve projectional editors (i.e. the EISE Query-Designer) have. Participants did not use any comparable projectional editing

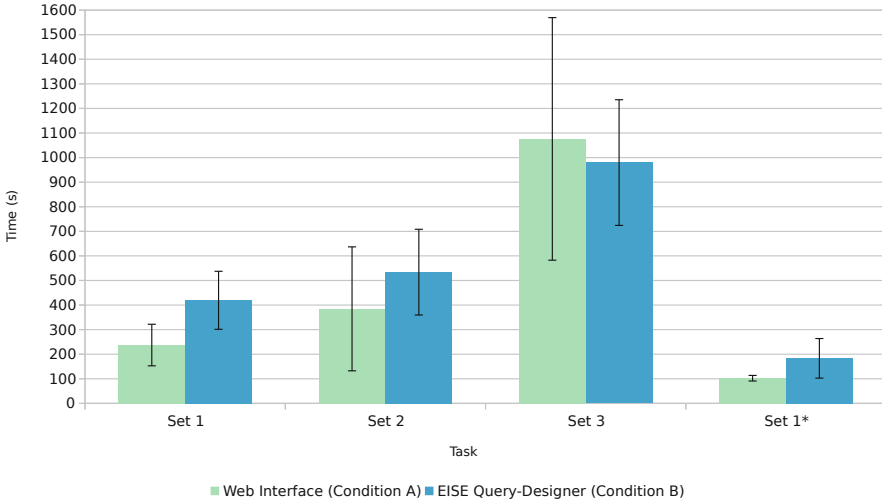


Fig. 6. Average time it took participants to finish each set of tasks.

software before and had to familiarise themselves with the concept of direct abstract syntax tree manipulation. Compared to this the baseline participants could directly begin their work with common source editing. Once participants of condition (B) overcome this initial difficulty, one observes that the approach starts to show advantages as users are able to perform more complex tasks (i.e. Set 3) in less time, while they experience less cognitive load when compared to the base line condition. Further, for all conditions a learning effect seems to be present: In both conditions the participants finished Set 1 (a permuted Set 1) faster than the initial Set 1 itself. The significance of this effect will have to be proven once an adequate sample size is collected.

Qualitatively we realised that presenting participants with an unknown domain, tools and DSLs requires well written introduction material. This leads to on average 20 min that are necessary to fully read the provided material. As a result we had to reduce the amount of tasks per set and removed the comprehension tasks completely to stay within planned 1 h maximum duration.

The development of the tool and all corresponding individual languages and solutions left us with several lessons learned. Language design is a difficult task - especially with a small team size. This leaves us with a recommendation for good prioritisation of sub tasks in the development life cycle. Feedback from target IDE users is very valuable and their acceptance is influenced by multiple factors of the tool. The provided editor/projection is an important element in this context as it is the first entry point for users. It will impact user performance and acceptance when simple functionalities (e.g. auto completion of simple data types) do not work as expected. These rather marginal properties can overshadow the valuable actual modeling of the domain the tool provides.

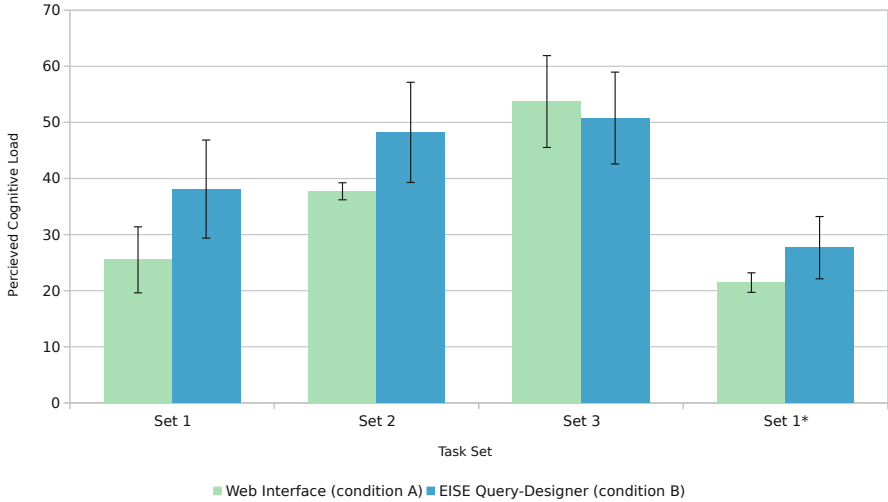


Fig. 7. Average cognitive load participants perceived per task set.

Another issue we encountered is concerned with the reuse of existing languages. As mentioned by Voelter, the so-called *DSL Hell* should be avoided and reuse should be preferred over developing own languages [5]. Having this in mind, we reused an existing *MPS* Cypher language and could thus reduce our workload significantly. However, a reused language introduce have its own problems, including faulty/unfinished design decisions, requirements/dependencies on legacy languages or software, abandonment, and others. To avoid mitigating workarounds in the surrounding new languages, the only option is to improve the reused language itself and feedback the improvements. Lastly, along goes the need for language versioning, which is a key discipline to be employed from the start in the language engineering process as it will otherwise hinder the development. With a recent update *Jetbrains* even addresses this issue and provides build-in support in *MPS*.

6 Related Work

Other than Cypher, there exist other extensive approaches to compose queries against graph databases, most notably Gremlin, SPARQL or GraphQL [15–17]. Gremlin is a functional and data-flow oriented graph traversal language of Apache TinkerPop. In contrast to the sub-graph pattern matching of Cypher, Germlin handles the graph traversals as sequences of steps which individually execute atomic steps on the available data stream. SPARQL on the other hand is a declarative query language crated by the W3C to perform queries on Resource Description Framework (RDF) graphs. Similarly to Cypher, SPARQL provides capabilities to query graph patterns, whereby queries comprise of a projection (SELECT) to specify the variables to project to as well as a body consisting of

so-called basic graph patterns that constitute filtering conditions. In addition, SPARQL supports conjunction and disjunction (UNION) as well as filtering, aggregation, counting, etc. After investigating a number of NoSQL databases and corresponding query languages for suitability to our goals, we selected Neo4j as a suitable graph database with Cypher being the directly supported option to query the data [18, 19]. The similarity of the querying language to SQL makes the language accessible and easy to learn for developers in the domain of human robot interaction. We have implemented domain specific languages that allow to model, generate and execute queries in the Cypher language. A transfer to other query languages (such as Gremlin or SPARQL) simply requires one to create an according MPS language as well as the necessary model-to-model transformations. With reasonable effort, all created Cypher based queries can be transferred other languages and/or executed on other database backends.

While usability is a well-researched and standardised field in software engineering, the benefits of specific IDEs or workbenches are often difficult to be evaluated. Improvement claims can be supported either formally, automatically, heuristically, or empirically. Bari et al. therefore proposed an evaluation process for the usability of DSLs that is applied during the development life cycle via various metrics, including questionnaires targeting the subjective measures such as cognitive load or perceived usability [20]. Further, others propose an integrated iterative testing approach that focuses on clearly defined metrics [21–23]. The core idea is to let evaluation span the entire DSL life cycle by assessing motivation, carrying our qualitative interviews, validating the DSL design, and quantifying benefits. According to Wegeler et al. a mix of quantitative and qualitative criteria is required as simple metrics cannot cover all advantages and risks [21]. However, each measure is important and should impact the DSL development process.

In practice, evaluation of DSLs is typically carried out with the involvement of domain experts. For example, Kärnä et al. used and evaluated their developed solution in the context of product development [24]. They let six users familiar with their target domain develop an application using their tool and compared the outcomes w.r.t. the three factors of developer productivity, product quality and the general usability of the tooling. An alternative is to carry out an extensive case study analysis involving a large user base. This is a valid evaluation approach especially when the presented tool already has a big user base that makes extensive use of the provided functionalities. Voelter et al. recently provided an excellent example case study providing great insight into the *mbeddr* platform [10]. This extensive review evaluates the language engineering process using JetBrains *MPS* as a language workbench and provides valuable lessons learned. From their point of view, designing a language that handles complex domains and is yet modular and scalable is feasible using *MPS*.

Finally, on a meta level, benchmarks for language workbenches themselves are being researched. In this context, the annual Language Workbench Challenge was instantiated in 2011 to provide an opportunity to quantitatively and qualitatively

compare approaches [8]. They compared 10 workbenches and provide a great overview presenting the state of the art options for language developers.

7 Conclusion and Outlook

In this paper, we present our current work to support query design and execution for application developers in the domain of embodied interaction in smart environments. Following a model-driven approach, we developed multiple *MPS* languages and solutions with the goal to support developers that need access and query interaction data in smart environments. We make explicit use of a multitude of features provided by *MPS*, most notably the generation of a domain specific IDE to be used by application developers. While the evaluation of IDEs and DSLs is a challenging endeavour, we presented study design that we have implemented to gather first qualitative observations and quantitative data on performance of developers using a small population sample as proof-of-concept of the design. Our preliminary results are promising, indicating that participants require less time and perceive lower cognitive load when designing more complex queries with the provided IDE. However, continued further research is required to verify the initial results with a larger group. For example, not all described architectural decisions could be implemented as planned due to time and resource constraints. As a result some functionalities have been implemented in different languages contrary to the original plan. Further, all languages require fine tuning, especially the entry points for users (i.e. the language editors/projections). Additionally, the planned extension points (such as data type mapping) have to be integrated into the architecture. Finally, we plan to further include the complete approach in the CSRA project and hence create stronger dependencies.

Acknowledgements. This research/work was supported by the Cluster of Excellence Cognitive Interaction Technology ‘CITEC’ (EXC 277) at Bielefeld University, which is funded by the German Research Foundation (DFG).

Ethical Approval and Informed Consent

All procedures performed in studies involving human participants were in accordance with the ethical standards of the institutional and/or national research committee and with the 1964 Helsinki declaration and its later amendments or comparable ethical standards. Informed consent was obtained from all individual participants included in the study.

References

1. Adair, B., Miller, K., Ozanne, E., Hansen, R., Pearce, A.J., Santamaria, N., Viegas, L., Long, M., Said, C.M.: Smart-home technologies to assist older people to live well at home. *J. Aging Sci.* **1**, 1–9 (2013)
2. Cavallo, F., Aquilano, M., Bonaccorsi, M., Limosani, R., Manzi, A., Carrozza, M.C., Dario, P.: Improving domiciliary robotic services by integrating the ASTRO robot in an AmI infrastructure. In: Röhrbein, F., Veiga, G., Natale, C. (eds.)

- Gearing Up and Accelerating Cross-Fertilization between Academic and Industrial Robotics Research in Europe: STAR, vol. 94, pp. 267–282. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-03838-4_13
3. Wrede, S., Leichsenring, C., Holthaus, P., Hermann, T., Wachsmuth, S.: The cognitive service robotics apartment: a versatile environment for human-machine interaction research. *KI-Künstl. Intell. (Spec. Issue Smart Env.)* **31**, 299–304 (2017)
 4. Holthaus, P., Leichsenring, C., Bernotat, J., Richter, V., Pohling, M., Carlmeyer, B., Köster, N., Meyer zu Borgsen, S., Zorn, R., Schiffhauer, B., et al.: How to address smart homes with a social robot? A multi-modal corpus of user interactions with an intelligent environment. In: *Proceedings of the 10th Language Resources and Evaluation Conference* (2016)
 5. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C., Visser, E., Wachsmuth, G.: *DSL engineering: designing, implementing and using domain-specific languages* (2013). dslbook.org
 6. Köster, N., Wrede, S., Cimiano, P.: An Ontology of Human Machine Interaction Data in Smart Environments. In: *SAI Intelligent Systems Conference 2016*. IEEE (2016)
 7. Voelter, M., Pech, V.: Language modularity with the MPS language workbench. In: *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1449–1450. IEEE (2012)
 8. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., van der Woning, J.: Evaluating and comparing language workbenches. *Comput. Lang. Syst. Struct.* **44**(A:2447), 1–38 (2015)
 9. Neo Technology Inc.: openCypher. <http://www.opencypher.org/> (2015)
 10. Voelter, M., Kolb, B., Szabó, T., Ratiu, D., Deursen, A.V.: Lessons learned from developing mbeddr: a case study in language engineering with MPS. *Softw. Syst. Model.* **16**, 1–46 (2017)
 11. Laugwitz, B., Held, T., Schrepp, M.: Construction and evaluation of a user experience questionnaire. In: Holzinger, A. (ed.) *USAB 2008*. LNCS, vol. 5298, pp. 63–76. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89350-9_6
 12. Hart, S.G., Staveland, L.E.: Development of NASA-TLX (Task Load Index): results of empirical and theoretical research. *Adv. Psychol.* **52**, 139–183 (1988)
 13. Brooke, J., et al.: SUS-a quick and dirty usability scale. *Usability Eval. Ind.* **189**(194), 4–7 (1996)
 14. Sauro, J.: *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best Practices*. Measuring Usability LLC, Denver (2011)
 15. Rodriguez, M.A.: The Gremlin graph traversal machine and language. In: *Proceedings of the 15th Symposium on Database Programming Languages*, pp. 1–10 (2015)
 16. Prud, E., Seaborne, A., et al.: *SPARQL query language for RDF* (2006)
 17. He, H., Singh, A.K.: Graphs-at-a-time: query language and access methods for graph databases categories and subject descriptors. In: *SIGMOD*, pp. 405–418 (2008)
 18. Tudorica, B.G., Bucur, C.: A comparison between several NoSQL databases with comments and notes. In: *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, pp. 1–5, June 2011
 19. Hecht, R., Jablonski, S.: NoSQL evaluation: a use case oriented survey. In: *Proceedings - 2011 International Conference on Cloud and Service Computing, CSC 2011*, pp. 336–341, December 2011

20. Barišić, A., Amaral, V., Goulao, M., Barroca, B.: Recent advances in multi-paradigm modeling (MPM 2011). *Electr. Commun. EASST* **50**, 220–224 (2011)
21. Wegeler, T., Gutzeit, F., Destailleur, A., Dock, B.: Evaluating the benefits of using domain-specific modeling languages - an experience report categories and subject descriptors. In: *Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling* (2013)
22. Barić, A., Amaral, V., Goulao, M.: Usability evaluation of domain-specific languages. In: *2012 Eighth International Conference on the Quality of Information and Communications Technology*, pp. 342–347 (2012)
23. Barišić, A.: Iterative evaluation of domain-specific languages. *CEUR Workshop Proc.* **1115**, 100–105 (2013)
24. Kärnä, J., Tolvanen, J.P., Kelly, S.: Evaluating the use of domain-specific modeling in practice. In: *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling* (2009)

A Simulation Framework to Analyze Knowledge Exchange Strategies in Distributed Self-adaptive Systems

Christopher Werner^(✉), Sebastian Götz, and Uwe Afmann

Software Technology Group, Technische Universität Dresden, Dresden, Germany
{christopher.werner,uwe.assmann}@tu-dresden.de, sebastian.goetz@acm.org
<https://tu-dresden.de/ing/informatik/smt/st>

Abstract. Distributed self-adaptive systems are on the verge of becoming an essential part of personal life. They consist of connected subsystems, which work together to serve a higher goal. The highly distributed and self-organizing nature of the resulting system poses the need for runtime management. Here, a particular problem of interest is to determine an optimal approach for knowledge exchange between the constituent systems. In the context of multi-agent systems, a lot of theoretical work investigating this problem has been conducted over the past decades, showing that different approaches are optimal in different situations. Thus, to actually build such systems, the insights from existing theoretical approaches need to be validated against concrete situations. For this purpose, we present a simulation platform to test different knowledge exchange strategies in a test scenario. We used the open source context simulator Siafu as a basis for our simulation. The described platform enables the user to easily specify new types of constituent systems and their communication mechanisms. Moreover, the platform offers several integrated metrics, which are easily extensible. We evaluate the applicability of the platform using three different collaboration scenarios.

Keywords: Distributed self-adaptive systems · Simulation
Multi-agent systems · Role-oriented programming

1 Introduction

Mobile devices, with the ability to sense and adapt themselves to a changing environment, are getting omnipresent in our society. Among them are smart watches, fitness trackers, (cleaning) robots, and wearables. To fully utilize these devices, they need to be integrated, which leads to complex systems, where different subsystems have to communicate with one another and establish different kinds of collaborations on the fly. For example, in a future (smart) office with two or more cleaning robots, where some are specialized on dry cleaning and others on wet cleaning, there is a need for coordination among them. The highly distributed

nature of such systems demands runtime management of each individual subsystem and optimization of the system as a whole to assure user-specified higher goals.

Self-adaptive systems are a promising approach for the development of such robotic systems utilizing self-organization and self-optimization techniques [14]. More precisely, robotic systems are usually characterized as distributed self-adaptive systems (D-SAS). Consequently, each subsystem is autonomous and makes decisions based on its own knowledge. However, each subsystem has to take the invariants of the system as a whole and its environment into account. To ensure that actions of one subsystem do not negatively impact another subsystem, a coordination mechanism is required. To realize this coordination, a spectrum of approaches can be used. At one extreme, a single, centralized system can be used to collect the knowledge from all subsystems and to influence them. At the other extreme, all systems directly exchange their knowledge in a peer-to-peer manner and, thereby are enabled to reason on the effects of their decisions on other subsystems. An approach for systems with vast amounts of subsystems is a hierarchic coordination, where the children of a node are coordinated by their parent node [8].

In either case, the central research question, we addressed in our previous work [8], is: which knowledge distribution strategy is the best for the current collaboration in a D-SAS.

The answer to this question heavily depends on (a) the environment, (b) the characteristics of constituting self-adaptive systems, and (c) the goals imposed on the system as a whole. Among these parameters, the trade-off to negotiate can be characterized as follows. The less information individual subsystems exchange with each other, the less are the implied costs, but the higher is the probability of them to make decisions having a negative effect on other subsystems. To practically decide which knowledge exchange strategy is the best, a more detailed specification of the costs, which are usually domain-specific, is required. For the case study, presented in this paper, we used the following properties as cost/quality:

- **Q1 Performance.** The performance of the system will decrease if there is unnecessary knowledge exchange. This means more network communication and more computational work for the system.
- **Q2 Real-Time.** Systems can have time restrictions (deadlines), which are not to be missed.
- **Q3 Energy Consumption.** The more information is exchanged, the more energy is spent on it, but the capacity of the participating subsystems is often restricted.
- **Q4 Memory Consumption.** Small devices are often limited in terms of their memory. Thus, gathering all available information on a single device can be impossible.
- **Q5 Privacy & Risk.** If the system comprises devices from different owners, policies to prevent unauthorized information exchange are required. The communication and collaboration constraints create security in the overall system.

The goal of this paper is to enable system developers to identify the optimal strategy before deploying it and researchers to investigate novel algorithms and approaches for knowledge exchange in D-SAS. The research questions addressed in this paper are:

- **RQ1:** How can the quality of a knowledge exchange strategy in a D-SAS be analyzed?
- **RQ2:** How should different knowledge exchange strategy in a D-SAS be compared with one another?

Therefore, this paper presents a reusable simulation framework called SAKE, which allows for testing different strategies in concrete scenarios. The framework is easily extensible w.r.t. new system types, knowledge exchange strategies and cost/quality characterizations (i.e., measurements). The simulation framework uses the open source context simulator Siafu [11] as a basis and is available on GitHub¹. As evaluation, we show three experiments conducted using the simulation framework, where different knowledge exchange strategies for a fleet of specialized cleaning agents are investigated on different maps.

The remainder of this paper is structured as follows. The next section provides an in-depth discussion on the concepts provided by the simulation framework, its extensibility, and its measurements. The three case studies we conducted are presented in Sect. 3. We demarcate our approach from related work in Sect. 4. Finally, in Sect. 5, we conclude the paper and discuss possible lines of future work.

2 A Simulation Infrastructure for Knowledge Exchange Strategies

In this section, we introduce Siafu which is used as time simulation framework and front-end for SAKE. Furthermore, we provide an overview of our proposed simulation framework in terms of its key concepts, its measurements, and its extensibility.

2.1 Siafu Simulator

The open source context simulator Siafu [11] acts as user interface and controller for each SAKE simulation. Siafu is implemented in Java and works on two dimensional maps and models agents and places which are located in the map. The concept of Siafu is modeled with a central *World* and *Agents* in it. The *World* is one object, that has a list of *Agents*, *Places*, and *Overlays*. The *Agents* and *Places* have a *Position* to specify the location in the map, whereas the *Overlays* hold information about the context of the world, e.g., temperature, sun intensity, and dirt level. For the start configuration, Siafu reads data from images with the same size. The input data is transferred over an interface to an

¹ <http://github.com/sgoetz-tud/sake>.

external simulation. This interface represents the connection between Siafu and SAKE or possibly any other simulation.

The key advantages of Siafu for us are the open Java source code, the fast start up time, and the good documentation and examples, but Siafu besides has some disadvantages. It is not possible to start more than one simulation at the same time only the opportunity exists to start the entire tool more than one time. In addition, the last commit in the GitHub repository was two years ago which means it is not under development anymore. As well, Siafu uses Eclipse SWT as user interface environment which supports Linux and Microsoft but only Mac Os cocoa as operating systems.

For our evaluation, it was important to run multiple simulations in parallel, which mean that each simulation runs in one thread. Therefore, we extend Siafu with a simulation configuration file as input to create as much simulations in parallel as configured.

2.2 Concepts

The central concept of the SAKE framework is depicted in Fig. 1. All system constituents have a physical and a virtual part. The physical part comprises an ensemble of hardware-components (e.g., computers or engines). The virtual part of the system comprises its roles, goals, and behaviors. Each agent plays different roles aiming to reach specified goals using available behaviors, which, in turn, are determined by the present hardware-components.

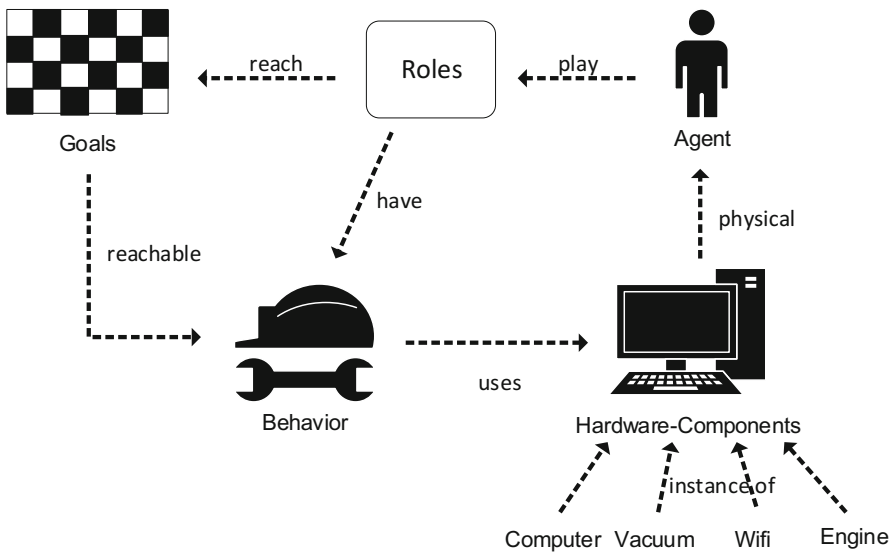


Fig. 1. Concept of the agent representation

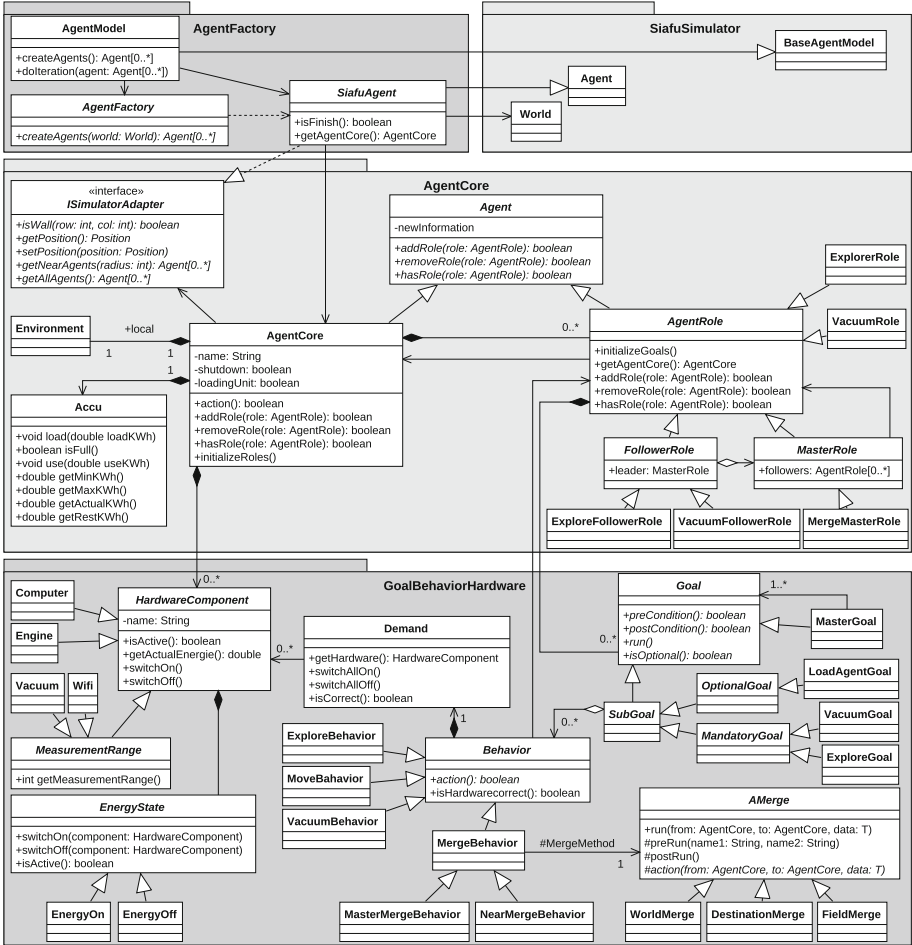


Fig. 2. Package class diagram of the simulator

Figure 2 depicts an overview of the framework as UML class diagram. At the top level, the framework is comprised of four packages: *SifafuSimulator*, *AgentFactory*, *AgentCore*, and *GoalBehaviorHardware*.

The *SifafuSimulator* package shows the reused and specialized classes of the Sifafu simulator to focus on evaluating knowledge exchange strategies in SAKE. We reuse the `World` class with the world model of Sifafu, i.e., it contains the common global environment of all system constituents and is the same for all `SifafuAgents`. We specialize the `Agent` and `BaseAgentModel` classes, where the second comprises a method to create the former and specifies how the simulation proceeds iteration-wise.

The *AgentFactory* package is represented by three classes: `AgentModel`, `AgentFactory`, and `SifafuAgent`. The first is a specialization of Sifafu's `Base-`

`AgentModel` and realizes the creation of agents using the factory method design pattern [7, pp. 107].

Next, the *AgentCore* package applies the role-object pattern [1] to enable a dynamic management of agent goals and related behaviors. The pattern comprises the abstract `Agent` class and the two classes `AgentCore` and `AgentRole`, where the first one contains the implementations of management methods to add and remove roles, whereas the second one only delegates to the former. Besides such management methods, the `Agent` class defines a property `newInformation` to indicate, whether the agent has collected information since its last exchange. The `AgentCore` class specifies, e.g., a property `name`, `localModel`, and `shutdown` and contains the only accumulator component of an agent.

The Simulator works with location data relying on maps and computes distance values between all agents. The *AgentCore* package describes the main structure and the collaborations of all agents, where each agent plays different roles, e.g., `MasterRole` or `FollowerRole`. These two roles and thereof specialized roles create a hierarchical collaboration structure between the agents (agents can be master and follower at the same time).

The *GoalBehaviorHardware* package is the last one and contains the modeled `Goals`, `Behaviours`, and `HardwareComponents`. The different goals are added to the roles of each agent, if it has appropriated hardware-components. In every time step, the action method of each goal is executed once. This method runs all behaviors of the goal once. When the postcondition is met, the goal is achieved and deleted. The composite pattern makes it possible to create a hierarchical structure of goals for each role. For the representation of a loading unit, it must be possible to have goals, which will never be achieved, but also are not hindering the system as a whole to finish. Therefore, we distinguish between `OptionalGoals` and `MandatoryGoals`. With this structure, we create elements which are relevant for the end of a scenario and elements which are only relevant for the continuous execution of the agent. The structure with some example goals is illustrated in Fig. 2 and further described in [17]. This technical introduction shows the extension points of the framework to use it in different scenarios.

2.3 Measurements

For every running example, we collect four different measurements to evaluate each strategy: time, data-exchange, memory consumption, and energy. The time is measured on the one hand as the computing time for each agent. On the other hand, the simulator saves the number of time steps an agent performs to reach a specific goal. Energy consumption is measured for each hardware-component an agent consists of. Each component has different energy states which represents working states. In Fig. 3, two exemplary energy states are depicted. Another example is shown in [6], where a bigger energy state model was used w.r.t. the accumulator state representing the probability of working success of the hardware-component. In our example, we only use the *on* and *off* state to represent a working component. Notably, besides energy states, the transitions

between states are also annotated with their respective energy demand. This strategy can represent every hardware-component in the simulator.

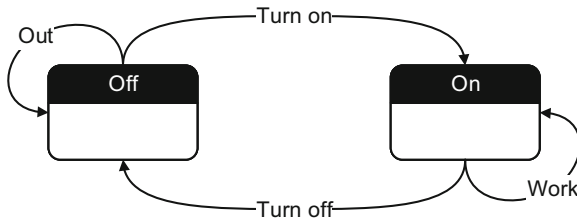


Fig. 3. Energy states for hardware-components

For the data-exchange measurement, the whole communication is monitored for an abstraction of the network load. To create a representative view of the data-exchange, the number of exchanged elements and the data-stream itself are stored in an optimal, save format. The memory consumption is monitored at the end of each simulation for every agent. This concept works under the assumption that each agent has its maximal local knowledge at the end of a simulation and did not delete anything during a simulation, as this would invalidate the results.

The measurements can be used to identify the cost quality properties as defined in Sect. 1. The **Performance (Q1)** is connected with the execution time and data-exchange measurements. Furthermore, the **Real-Time (Q2)** arises from the number of time steps measured for each complete simulation. The **Energy Consumption (Q3)** and **Memory Consumption (Q4)** result from their corresponding energy and memory consumption measurements. The **Privacy & Risk (Q5)** property is not measurable by the SAKE simulator, but the more data an agent exchanges the more of his privacy is sacrificed. In the evaluation section, we only show the results of the measurement values and not the connected quality properties, because the weighting between two measurement values to create one quality property is different in every use case.

2.4 Extensibility

To enable reuse, it must be possible to easily extend the simulator and test different novel exchange strategies. Our framework provides different modification and connection points for new strategies. The first point is the factory package, which creates every agent with its own hardware-components and roles. In the *AgentCore* package, new agent roles can be implemented to specify different collaboration structures, goals, and responsibilities for the agents. The goals, behaviors, and hardware-components are modified like the agent roles, so it is possible to add new functions and components. The energy states contain currently only the **EnergyOn** and **EnergyOff** states and are modeled with the

State-pattern. The State-pattern offers an extension point to create new states like idle, busy, or standby.

Each agent saves in every time step the measurement values of the simulation and adds them to an evaluation file. For that, the evaluation function is called every time step and can be modified to measure more values. After the complete simulation run, the simulator creates a JSON string for each evaluation object and saves them in more than one document. These documents can be used to load the value-lists in a new software-tool and to create diagrams or tables for analysis.

3 Evaluation

We illustrate the results of the SAKE simulation based on an office cleaning running example with cleaning robots. Three different strategies are used to test and get comparable values for the evaluation. The easiest way is a complete collaboration between each agent. This means that, if an agent meets another agent, he will give him his complete local model of the world. The introduction of a master agent creates a hierarchical collaboration structure between the agents. Furthermore, it makes global work decisions and handles the communication.

The environment of the test scenario includes different parameters, e.g., size and nature to represent different environmental contexts. Not only the environment also dependencies between agent types influence the results of each test case. To get representative results between single simulation runs, we run each one five times and use the average values for the depicted diagrams. These five runs are necessary, because the agents decide randomly for one destination in the set of nearest new destinations.

3.1 Running Example

The running example is an office cleaning scenario for one floor. To not interfere with office work the cleaning of office spaces has to take place outside working hours. This implies one important requirement for cleaning robots that are used in this scenario, which is to satisfy deadlines in various different working spaces. The easiest way is to use one agent, however he might not finish in time. In this case, it is important that different agents share their work and communicate with each other about the areas they already cleaned. For this reason, we create three different strategies:

- **C1: Complete Collaboration.** Each agent exchanges his complete local information model with a closely located agent, but with a time delay to avoid data-exchange in small cycle time.
- **C2: Communication with Master.** A master coordinates and handles the communication with near agents. The master is located at the loading unit and exchanges the local knowledge when the agents are loading. Every agent computes his drive destinations based on its own information. This approach reduces the locally needed memory and minimizes the knowledge exchange.

- **C3: Communication and Coordination with Master.** The master always communicates with all agents and tells them what to do. The working agents need a communication infrastructure to stay connected with the master. However, they need less memory, because they do not store any information locally.

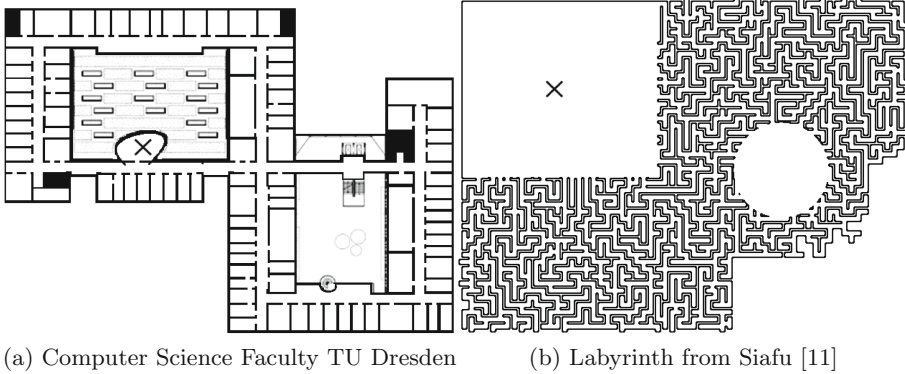


Fig. 4. SAKE evaluation maps

In Fig. 4, we show two of three different maps on which we tested the three-strategies. The black cross in each map represents the loading unit and with it the location of the master agent. Figure 4a represents a floor of the computer science faculty of the TU Dresden. It gives the best real world example with small rooms and big corridors. Henceforth, we only show the results from this map. Figure 4b represents a labyrinth which is predefined in Siau [11]. This is the biggest map we use with a lot of dead ends and narrow ways. At last, we also used a quadratic hall map, which is not depicted in Fig. 4. The results of the other maps and agent types are shown in [17]. In the maps, each white pixel is a point for cleaning and the black pixels show the walls of the map.

To create dependencies between agents, we deployed two different types. Before, e.g., a vacuum agent can clean the world it needs a map of the area. This implies a hierarchical structure of agents. A master agent communicates with exploration agents about the world and exchanges his knowledge with the vacuum agents. The phases of the cleaning process are (a) create the map with explore agents, and (b) clean the area with vacuum agents. This step dependencies mean that every agent needs parts of the information from an agent one step before to start with its work. This structure creates waiting periods for agents from a higher level in the hierarchy and influences the deadline property.

Thus, we first increase the number of explore agents from one to ten and then add vacuum agents and increase them from one to ten, too. This creates a usable dataset for analysis. In the real world, it makes sense to use other numbers of

agents, but for showing them in the diagrams of Figs. 5 and 6 we decide to use this numbers and dependencies.

3.2 Time Measurement

In this subsection, the time measurement results from the faculty map and the three strategies are presented. In Fig. 5, the average number of time steps are shown. The number of agents and the type of each new agent are explained in the previous subsection. A time step is one iteration run in the simulator, where every agent activates each *run* method in its goals once as mentioned in Subject. 2.2. The deviations in the diagram stems from the average values of five runs. Each agent randomly searches the next nearest destination to work. This background strategy fulfills the goals on divergent ways. In Fig. 5 one time step can be seen as one second or millisecond in real time. From the tenth to the eleventh agent a big step arises because of the incoming agent dependency.

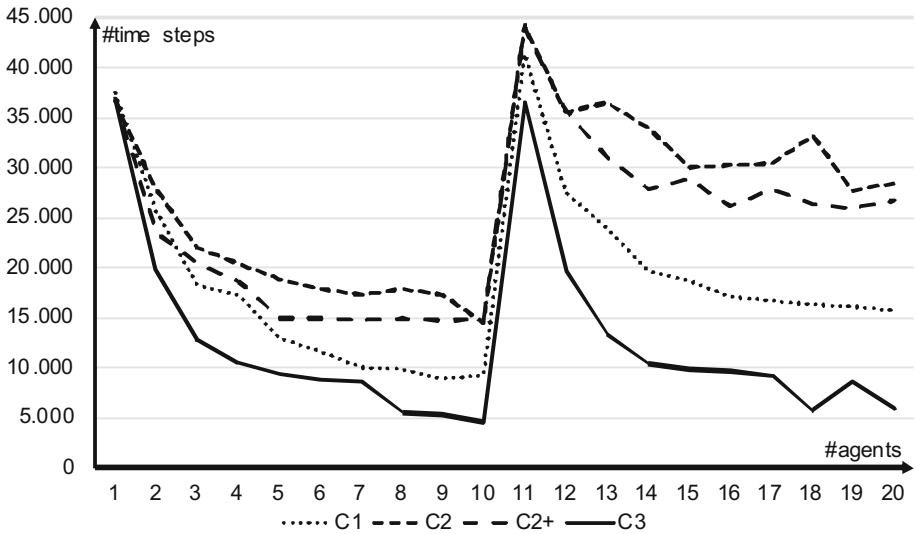


Fig. 5. Average number of time steps from the faculty map

As we can see, C3 is the optimal strategy, because the master always collects all knowledge and makes optimal decisions with its complete knowledge base. C3 makes it possible to remove the big step by adding of new agents. There is no visible difference between the values 7 and 17. In C1 and C2, there is always such a step, which cannot be removed with adding more agents (e.g., between value 6 and 16 or 7 and 17). The master as communication interface at the loading unit (C2) always creates such a step for new types of agents. In the strategy C2+ from the diagram, the master defines the first destination after loading to

spread around all agents, i.e., avoid duplication of cleaning work. This strategy minimizes the step compared to C2, but cannot remove it completely. As a result, after five agents of a type the time savings will get very low in all configurations. In addition, C2+ gets better and C1 worse in bigger maps than in smaller ones, which is a result from comparing all time diagrams from all maps. The earlier one agent gets information from other ones, the better the decisions of the agent will be.

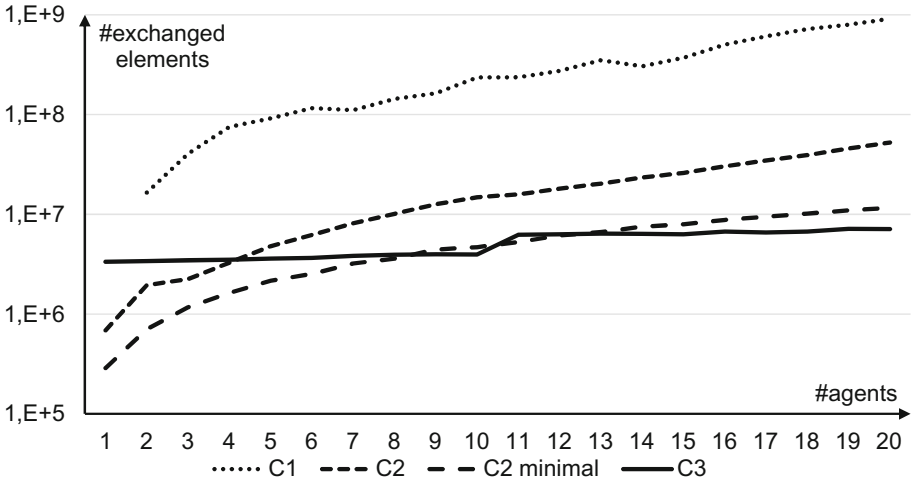


Fig. 6. Complete knowledge exchange values from the faculty map

3.3 Knowledge Exchange

In this part, a short overview on the knowledge exchange results is given. In Fig. 6, the results of test runs are shown for the faculty map. The y-axis shows the number of exchanged elements between all agents in logarithmic scale. The minimum y-Axis value is one hundred thousand to highlight interesting parts of the diagram. In this diagram an exchanged element represents either an integer, string, float, or double value, because all primitive data elements are counted as one element. This approach only takes care of the real exchanged knowledge without any optimization on input and output communication strategies. C1 has the most exchanged elements with exponential rising, because the agents make a total exchange every time they meet. The un-optimized C2 has more than ten times less data-exchange elements than C1 and the optimized variant (C2 minimal) has more than twenty times less data-exchange elements depending on the number of agents. The optimized option uses internal timestamps and states to reduce the redundant exchanged elements, but this optimization raises the memory consumption on all agents. C3 has the least knowledge exchange, because the master gets directly the new information from each agent and only has to

send new destinations. The global decisions from the master minimize redundant work and redundant exchanged elements. With more agents, C3 has the best results w.r.t. the overall knowledge exchange and it is the only configuration with a minimal step in the diagram because of the new incoming agent type.

3.4 Results

The evaluation results show different dependencies between input parameters and the final outcome. For example, the number of agents and the choice of a strategy change the results in different ways. If the number of agents rises, the overall time decreases, but nevertheless the energy consumption rises, too. This means that the influence of reducing time with more agents is smaller than the energy consumption per time step. The diagrams with the energy consumption results are presented in [17].

The three maps with their different characteristics have influence on the number of meetings, because large open spaces increase the probability of meetings. In a labyrinth with narrow ways an agent rarely meets others and cannot exchange his knowledge. In bigger maps the strategy, where the master spreads the agents, is faster because the probability of same working areas decreases.

The evaluation results of the different strategies show, that a master reduces the amount of exchanged data between all agents and creates, e.g., an interface for human interaction. In all test cases, we found, that a complete controlling master returns the best result in a perfect world. However, the requirements for C3 are far away from reality. It is likely, that this configuration will produce different results in a real-world case study because of location and hardware failures. Strategies with a master show, that the more control options the master has, the faster the run is, but the more configurations to test. A master is also a bottle neck in the infrastructure, which could be replaced with more masters or with new combinations of different strategies.

4 Related Work

The problem of a good simulation framework is that there must be a lot of possibilities to modify the simulator and simulation for new test cases and to collect as much information as possible. There are many simulators for different specific scenarios. For example the SUMO [2] simulator simulate traffic in an urban area. SUMO is used to test traffic light control algorithms to get the most efficient light control system for a city. Therefore, it respects traffic tips, which are controlled with characteristic parameters. Similarly, UdelModels [10] is a simulation framework for urban networks. It takes realistic propagation into account and provides a user interface to create cities. The OMNeT++ [16] simulation sweet includes different tools and simulates network protocols in varying areas.

In Sect. 2.1, we introduce the open source context simulator Siafu [11], but there are other similar simulators, for example, JAS [15]. JAS is implemented

in JAVA and gets his functional scope from well tested standard libraries and third party libraries. It represents agents in components and provides a variety of standard collections of components and rules to create simulations. Thus, it appears to be discontinuous, because JAS did not present any new version since 2006.

In the area of knowledge exchange, different protocols are introduced from standardization communities. For example, the FIPA [13] produces software specifications for multi-agent systems (MAS) like communication protocols to maximize the compatibility of MAS. JADE [3] for example is a platform for peer-to-peer agent based applications. It describes a middleware, which uses the FIPA specifications for the communication interface between agents. Thereby, it provides a graphic userinterface and facilitates the troubleshooting and deployment phase of the system. The platform is implemented in JAVA and can be used to realize different kinds of agent architectures. In the background, JADE uses containers for the representation of an agent. The contrary part of JADE is JACK [9], a commercial tool which implements the FIPA communication protocols in Java. JACK is developed from the Agent Oriented Software Pty, Ltd. (AOS) and is a progression of the Procedural Reasoning System (PRS), and the Distributed Multi-Agent Reasoning System (dMARS). Like JADE, it helps to create MAS. Every agent works in JACK in accordance with the BDI (Belief, Desire and Intentions) principle, such that every agent is described with his goals, knowledge, social skills, and acts in reaction on the environmental input.

For the SAKE simulation, we evaluate the running example with knowledge exchange strategies based on real data objects. The specific exchange strategies are introduced by Götz et al. [8], describing three different strategies. They contain the *total-complete* strategy where all agents collaborate with each other and exchange their complete knowledge. Then, the *partial-complete* strategy where each sub-agent exchanges his complete knowledge with his direct collaborators, and the third strategy *partial-subset* where the agents only exchange parts of their own knowledge with their direct collaborators. This strategies are the templates for our strategies and implementation.

Knowledge exchange is important in all kinds of MAS and is often used in different ways. For example, DEECo [5], SeSaMe [4], and DECIDE [6] are frameworks to create MAS. DEECo is an ensemble-based component system where an ensemble represents dynamic bindings of a set of components and thus determines their composition and interaction. The ensemble component describes the collaborations and data connections. Nevertheless, if the ensemble component only mentions small parts as exchanged data, DEECo proactively shares all his information and picks only the important parts out at the end. SeSaMe coordinates distributed components in various self-organizing inter-composed groups based on the types of roles they can play. Thus, it facilitates a direct interaction between the supervisors and followers. DECIDE splits the control-loops to many nodes of a distributed self-adaptive system. This generates more flexibility and mitigates failures in the master node. The goal of DECIDE is to check the system at runtime and guarantee the quality requirements of critical self-adaptive systems.

In this work, we mention the privacy and risk quality properties of a MAS, but do not consider them. In [12], Palomares et al. introduce a framework for risk-aware planning and decision making. In detail, they require a known map from the world with destinations, a number of identical agents, and probabilities for connections between platforms. Therefore, they create the best strategy to reach the global goal of the configuration. In SAKE each agent never break down. Consequently, we do not need such risk-aware planning algorithms at the moment. The advantages of SAKE comparing to other frameworks are the extension points of agent properties, application scenarios, and measurement of evaluation parameters. It is possible to create a completely new scenario or only test and create new exchange strategies.

5 Conclusion and Future Work

In this paper, we presented a simulator for different kinds of multi-agent systems and tested it in one test scenario with three collaboration configurations on three different maps. For the simulator, it was important to change different start parameters to create a wide range of test cases and environments. To analyze the tested strategies, we introduced some easily changeable and extendable predefined measurements. The evaluation results shown basic correlations between the input parameters, the nature of the environment, and the measurement results. Therefore, the SAKE simulation framework is suited for testing the systems behavior before the first real-world run and finding the best system configuration.

As future work, we plan to implement more knowledge exchange strategies and check the results for strategies in the real world. For a real-world implementation and testing, it is important to have changeable environments, which is not yet implemented and realized. In office scenarios, this point arises with new temporary obstacles. In the real world, agents can fall down or crash so the system must change the strategy on the fly. For this, the SAKE simulation should be combined with a probability model. Furthermore, an open question is the automatic selection of the best strategy based on quality properties.

Acknowledgments. This work has been funded by the German Research Foundation within the Collaborative Research Center 912 “Highly Adaptive Energy-Efficient Computing” and within the Research Training Group “Role-based Software Infrastructures for continuous-context-sensitive Systems” (GRK 1907).

References


1. Bäumer, D., Riehle, D., Siberski, W., Wulf, M.: The role-object pattern. In: Proceedings of the 4th Pattern Languages of Programming Conference (PLoP) (1997)
2. Behrisch, M., Bieker, L., Erdmann, J., Krajzewicz, D.: SUMO-simulation of urban mobility. In: The Third International Conference on Advances in System Simulation (SIMUL), Barcelona, Spain (2011)

3. Bellifemine, F., Poggi, A., Rimassa, G.: JADE-A FIPA-compliant agent framework. In: Proceedings of PAAM, London, vol. 99, p. 33 (1999)
4. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: a generic architecture for storing and querying RDF and RDF schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-48005-6_7
5. Bures, T., Gerostathopoulos, I., Hnetyinka, P., Keznikl, J., Kit, M., Plasil, F.: DEECO: an ensemble-based component system. In: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE, pp. 81–90. ACM, New York (2013)
6. Calinescu, R., Gerasimou, S., Banks, A.: Self-adaptive software with decentralised control loops. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 235–251. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_16
7. Gamma, E.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education India, Bengaluru (1995)
8. Götz, S., Gerostathopoulos, I., Krikava, F., Shahzada, A., Spalazzese, R.: Adaptive exchange of distributed partial models@run.time for highly dynamic systems. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE (2015)
9. Howden, N., Rönquist, R., Hodgson, A., Lucas, A.: Jack intelligent agents—summary of an agent infrastructure. In: 5th International conference on autonomous agents (2001)
10. Kim, J., Sridhara, V., Bohacek, S.: Realistic mobility simulation of urban mesh networks. *Ad Hoc Netw.* **7**(2), 411–430 (2009)
11. Martin, M., Nurmi, P.: A generic large scale simulator for ubiquitous computing. In: Third Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous), San Jose. IEEE, July 2006
12. Palomares, I., Killough, R., Bauters, K., Liu, W., Hong, J.: A collaborative multi-agent framework based on online risk-aware planning and decision-making. In: 2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI), pp. 25–32, November 2016
13. Poslad, S.: Specifying protocols for multi-agent systems interaction. *ACM Trans. Auton. Adapt. Syst.* **2**(4), 15 (2007)
14. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.* **4**(2), 14:1–14:42 (2009)
15. Sonnessa, M.: JAS: Java agent-based simulation library, an open framework for algorithm-intensive simulations. In: Industry and Labor Dynamics: The Agent-Based Computational Economics Approach. World Scientific, Singapore (2004)
16. Varga, A., Hornig, R.: An overview of the OMNeT++ simulation environment. In: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops, Simutools, Brussels, pp. 60:1–60:10. ICST (2008)
17. Werner, C.: Adaptive knowledge exchange with distributed partial models@run.time. Master's thesis, Technische Universität Dresden, January 2016

OCL and Textual Modeling (OCL)

Workshop in OCL and Textual Modelling

Report on Recent Trends and Panel Discussions

Robert Bill¹, Achim D. Brucker²(✉) , Jordi Cabot^{3,4}, Martin Gogolla⁵,
Antonio Vallecillo⁶, and Edward D. Willink⁷

¹ TU Vienna, Vienna, Austria
bill@big.tuwien.ac.at

² The University of Sheffield, Sheffield, UK
a.brucker@sheffield.ac.uk

³ ICREA, Barcelona, Spain
jordi.cabot@icrea.cat

⁴ UOC, Barcelona, Spain

⁵ University of Bremen, Bremen, Germany
gogolla@informatik.uni-bremen.de

⁶ Universidad de Málaga, Málaga, Spain
av@lcc.uma.es

⁷ Willink Transformations Ltd., Reading, UK
ed@willink.me.uk

Abstract. This paper reports on the panel session of the 17th Workshop in OCL and Textual Modeling. As in previous years, the panel session featured several lightning talks for presenting recent developments and open questions in the area of OCL and textual modeling. During this session, the OCL community discussed, stimulated through short presentations by OCL experts, proposals for improving OCL to increase the attractiveness of textual modeling.

This paper contains a summary of the workshop from the workshop organisers as well as summaries of two lightning talks provided by their presenters.

Keywords: OCL · Textual modeling

1 Introduction

Textual modeling in general and OCL in particular are well established. This year does not only mark the 17th edition of the OCL workshop, it also marks the twentieth anniversary of the first publication of the OCL standard by the OMG [3]. Nevertheless, textual modeling in general and OCL in particular is an active field of research.

The workshop received seven submissions from which five were selected as full papers. Each paper was reviewed by at least three PC members. The workshop hosted an open session with “Lightning Talks (5 min)” at the end of the day

where speakers were given the opportunity to talk about whatever they wanted, as long as it was related to the topics of the workshop. Three presentations were given. The topics discussed at the workshop covered topics such as the translation of OCL to programming and specification languages, proposals for improving textual modeling languages and their tool support, as well as the development of an OCL benchmark.

The lightning talks at the panel session of the workshop provided a platform for the textual modeling community to discuss and present tools, ideas, and proposals to support textual modeling as well as to shape the future of textual modeling. The following sections, each of them contributed by one expert of the field, discuss the different tools and ideas that were discussed during the panel session.¹

2 Sometimes Postconditions Do Not Suffice

Martin Gogolla and Antonio Vallecillo

2.1 Non-determinateness and Randomness in OCL

Recently there have been proposals for incorporating the option to express randomness in OCL [2, 4]. In many modeling and simulation environments, the use of random numbers and probability distributions are used to combine definite knowledge with an uncertain view on the result or the population of a test case. Thus, there is an interest to express such requirements in UML and OCL.

OCL already has operations that possess a flavor of randomness, like the operation `any()`. One could also consider a new collection operation `random()` that randomly chooses an element from the argument collection. Our understanding of such operations is that they cannot be characterized only by ‘traditional’ postconditions. In particular special attention has to be given in order to express the difference between `any()` and `random()`: A ‘traditional’ postcondition would characterize ‘one’ call to the respective operation (for example, with `Set{1..6}->includes(result)`); but these two operations must be characterized by ‘many’ operation calls and a comparison between their actual and their expected results. We show with a small example how such a ‘non-traditional’ postcondition in form of an invariant could look like.

2.2 Formulating Randomness Quality Criteria as an Invariant

Consider the class diagram in Fig. 1 that is intended to model a dice. Every time the operation `random6()` is called it should return a random number between 1 and 6. Our expectation for the operation `any()` would be that it can also return any number between 1 and 6, but that different calls to `any()` always yield the same result. In contrast, different calls to `random()` should show different results.

¹ There is no summary for the lightning talk of Dimitris Kolovos, entitled “Managing MATLAB Simulink models with Epsilon,” as the author considered the results to be in an too early stage to be summarized in a written report.

The attributes in the class `Dice` (see Listing 1.1) give a simple measure for the quality of the generated random numbers. Basically the attributes say that the number of tests for `random6()` that have to be performed is `numChecks` and that, for example, the difference between (a) the amount of operation calls yielding 2 and (b) the amount of operation calls yielding 5 is at most `deltaMax`. These requirements are formulated as an OCL formula in terms of an invariant of the class `Dice`. The requirement should not be formulated as a `random6()` postcondition because this would lead to a situation where a recursive call to the operation would occur in the postcondition. Much better criteria for the random distribution could be formulated in OCL as well. The purpose of the shown invariant is only to demonstrate that many calls to an operation may be necessary in order to express desired properties.

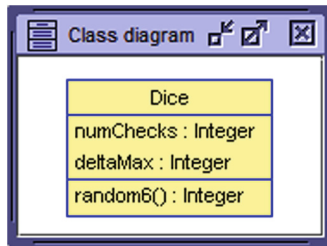


Fig. 1. Class diagram for Dice example.

3 Commutative Short Circuit Operators

Edward D. Willink

OCL's 4-level logic has been a source of much unhappiness and while various solutions have been suggested, none have met with enthusiasm. We look at where the unhappiness comes from and thereby suggest a new solution.

The OCL designers defined an underlying model in which all expressions have types. Consequently the mathematical concept of truth was reified by a `Boolean` type with associated Boolean library operations. The designers chose to avoid exceptions. This in combination with UML conformance required a `null` value for the missing value of properties with optional multiplicity, and an `invalid` value for everything bad that might be evaluated.

Unfortunately `null` and `invalid` pollute the simplicity of truths and so the Amsterdam Manifesto [1] elaborates Boolean operators with short-circuit like functionality for problems such as:

```
a <> null and a.doSomething()
```

```

class Dice
attributes
  numChecks: Integer
  deltaMax: Integer
operations
  random6(): Integer=Set{1..6}->random()
post returns_1_6: Set{1..6}->includes(result)
constraints
  inv manyRandom6CallsResultInNearlyEquallyDistributedValues:
  -- call random6() many times
  -- store resulting amounts in Sequence{A1,A2,A3,A4,A5,A6}
  -- check differences between A1..A6
let amts=Set{1..numChecks}->iterate(i: Integer;
  amts: Sequence(Integer)=Sequence{0,0,0,0,0,0} |
  let r=random6() in
  Sequence{
    if r=1 then amts->at(1)+1 else amts->at(1) endif,
    if r=2 then amts->at(2)+1 else amts->at(2) endif,
    if r=3 then amts->at(3)+1 else amts->at(3) endif,
    if r=4 then amts->at(4)+1 else amts->at(4) endif,
    if r=5 then amts->at(5)+1 else amts->at(5) endif,
    if r=6 then amts->at(6)+1 else amts->at(6) endif}) in
Sequence{1..5}->iterate(i; diffs: Sequence(Integer)=Sequence{} |
  Sequence{i+1..6}->iterate(j; diffs2: Sequence(Integer)=diffs |
    diffs2->including((amts->at(i)-amts->at(j)).abs()))->
    forAll(d | d<=deltaMax)
end

```

Listing 1.1. Specification of the Dice example.

However the operators remain commutative and so it is suggested that all terms are evaluated in parallel until the result is knowable. A Karnaugh Map defines the mapping from the true (T), false (F), null (ϵ) and invalid (\perp) values of Left and Right inputs to the `and` output.

Left	Right	and	requires	'and'
T	T	T	T	T
T	F	F	F	F
T	\perp, ϵ	\perp	\perp	\perp
F	-		F	
F	T, F	F		F
F	\perp, ϵ	F		\perp
\perp, ϵ	-		\perp	
\perp, ϵ	T, F, \perp, ϵ	\perp		\perp

Parallel execution is an implementation nightmare and the intermediate `invalid` results can be inefficient. If we eliminate commutative short circuits, we find that `invalid` results are exceptional rather than normal.

```

a <> null requires a.doSomething()

```

A new `requires` operator imposes a left argument first evaluation order for `and`. This avoids the spurious `invalid` results from the right argument and clearly indicates the intent to handle non-truths. The `and` operator can then be used for truths only. Once static analysis verifies that neither left nor right input of an `and` operator can be `null` or `invalid`, an implementation may implement a regular ‘`and2`’ operation that returns `invalid` for any `null` or `invalid` input.

A new `obviates` operator is also needed to regularize `or` short circuiting.

4 Conclusion

The lively discussions both during the lighting talks as well as for each paper that was presented showed again that the OCL community is a very active community. Moreover, it showed that OCL, even though it is a mature language that is widely used, has still areas in which the language can be improved. We all will look forward to upcoming version of the OCL standard and next year’s edition of the OCL workshop.

Acknowledgments. We would like to thank all participants of this years OCL workshop for their active contributions to the discussions at the workshop. These lively discussions are a significant contribution to the success of the OCL workshop series.

References

1. Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J., Wills, A.: The amsterdam manifesto on OCL. In: Clark, T., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 115–149. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45669-4_7
2. Johnson, P., Ullberg, J., Buschle, M., Franke, U., Shahzad, K.: P²AMF: predictive, probabilistic architecture modeling framework. In: van Sinderen, M., Oude Luttighuis, P., Folmer, E., Bosems, S. (eds.) IWEI 2013. LNBIP, vol. 144, pp. 104–117. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36796-0_10
3. OMG: Object constraint language specification (version 1.1). OMG document (1997). <http://www.omg.org/cgi-bin/doc?ad/97-08-08>
4. Vallecillo, A., Gogolla, M.: Adding random operations to OCL. In: Posse, E., Ratiu, D., Selim, G., Zalila, F. (eds.) Proceedings of the Workshop on Model Driven Engineering, Verification and Validation (MODEVVA 2017). CEUR Proceedings (2017)

Improving Incremental and Bidirectional Evaluation with an Explicit Propagation Graph

Frédéric Jouault^(✉), Olivier Beaudoux, Matthias Brun, Fabien Chhel,
and Mickaël Clavreul

Groupe ESEO, Angers, France

{frederic.jouault,olivier.beaudoux,matthias.brun,fabien.chhel,
mickael.clavreul}@eseo.fr

Abstract. Active operations enable bidirectional incremental evaluation of OCL-like expressions on collections: changing the source (resp. the result) of an expression causes corresponding updates in the result (resp. the source). The current evaluation approach of active operations is based on the observer pattern. Previous work showed how they can be used for model transformation, and that they can scale to processing large models while maintaining collection ordering. However, observation makes the directed acyclic propagation graph implicit, and imposes a depth-first traversal. This sometimes results in unwanted transitory states, which uselessly increase the amount of computations required for propagating some changes.

To address this issue, we propose in this paper to make the propagation graph explicit. This enables separation of the propagation graph from traversal strategies (e.g., breadth-first instead of depth-first). We show how this approach gets rid of unwanted transitory states, and discuss some of its other advantages like enabling more efficient graph visualization and analysis, as well as more compact propagation graph representations. Additionally, incremental algorithms of active operations do not need to be changed, but can actually be better encapsulated, which decreases maintenance cost of the incremental framework.

Keywords: Incremental evaluation · Active operations
Propagation strategies

1 Introduction

Incremental evaluation of model queries or transformations has many applications. One of its advantages is that it reduces the amount of computation required to obtain a new result after only a few changes occurred on source models. In the traditional model-driven development context, it can be used to speed up the development process. After a developer has updated a design model, incremental evaluation makes queries (e.g., to check invariants), or transformations (e.g., to generate concrete models from abstract ones) faster. In a *models at runtime*

context [1], incrementality speed up is potentially even more desirable to make user-facing systems more responsive.

Another advantage of incremental transformation execution is that it updates target models in-place rather than producing whole new models. If the target model is loaded in a model editor, which binds shapes on a canvas to its model elements, incrementality enables updating the target model's visual representation automatically upon source model changes. This property may also be useful in a models at runtime context where a concrete target model may be updated without reloading it from scratch.

Active operations [2] have been defined to provide a basis on which incremental model query and transformation tools can be built. They also provide bidirectional change propagation when possible. For instance, in the case of both source and target models loaded in model editors, bidirectionality enables making changes on the target model, and see those changes propagated to the source model. Active operations work by wrapping every mutable value inside of an observable box, and by providing operations that compute initial values (i.e., perform evaluations), and can then propagate changes. Each operation knows how to change its result boxes when its source boxes change, and *vice versa* where applicable. Complex expressions are represented by composing these operations, which results in an implicit directed propagation graph that connects source boxes to target boxes via active operations and intermediate boxes. Furthermore, this graph is acyclic because a result cannot be used to compute itself. Because its purpose is to propagate changes, which are data, it is a kind of dataflow graph.

Active operations have notably been shown to be applicable to incremental OCL evaluation [3], and to bidirectional model transformation [4]. The Active Operations Framework (AOF) is their current implementation, and it has been shown to scale to large models [5].

However, the implementation approach on which AOF is currently based presents some issues. The whole framework is based on the *Observer* pattern. This approach works by making all intermediate values observable, and making all operations observe their sources, and their results. As a consequence, traversal of the propagation graph follows a depth-first strategy. The main issue is that n-ary operations (i.e., operations with n inputs, such as binary operations with 2 inputs) may be notified several times for a single source change if several of their inputs are directly or indirectly (i.e., via other operations) connected to the same source. This results in unwanted transitory states, which are sometimes inconsistent. It is possible to prevent these inconsistencies from propagating to the target model, but (consistent) transitory states are still observable by a user of the target model. Moreover, all these transitory states must still be processed by all downstream operations (i.e., operations that depend on the results of the misbehaving n-ary operation), which leads to performance degradation.

In this paper, we propose a new approach in which the propagation graph is made explicit. Propagation algorithms are no longer implemented on nodes of the propagation graph, which notably enables decoupling operation-specific

propagation algorithm from the overall propagation graph traversal strategy. Making the graph explicit enables the implementation of different traversal strategies without impacting the way operation-specific algorithms are implemented. Different strategies typically trade slower initial evaluation time for faster propagation times, or *vice versa*. It notably enables traversal strategies that do not have the issues identified in the observation-based approach.

Section 2 presents the observer-based approach used in the current implementation of active operations, and details change propagation issues by illustrating them on a motivating example. Section 3 introduces the notion of explicit propagation graph, and how it can be used to address the identified issues, with illustration on the motivating example. Some related works are presented in Sect. 4. Finally, Sect. 5 gives some concluding remarks.

2 Observer-Based Propagation

2.1 Overview

Observer-based propagation is the original approach for the incremental evaluation of active operations. Its mechanism is illustrated on Fig. 1.

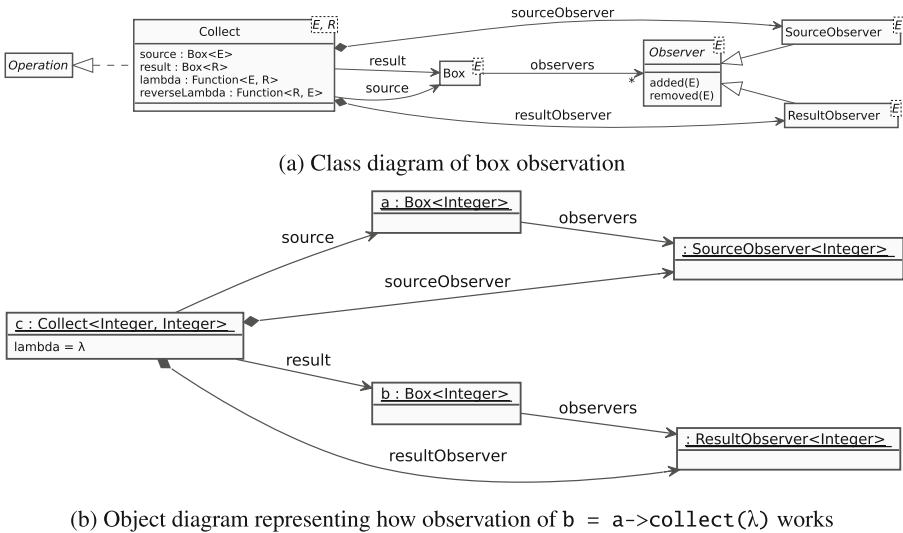


Fig. 1. Diagrams representing implementation of box observation

Firstly, Fig. 1a is a simplified class diagram representing how operation `Collect` is reified as a class with one `source` `Box`, and one `result` `Box`. Each `Box` has `observers`, which it notifies when it changes. For instance, when an element

is added to (resp. removed from) a `Box`, this `Box` calls method `added(E)`¹ (resp. `removed(E)`) on each of its observers. Every operation, like `Collect`, implements specific `Observers` like `SourceObserver`, and `ResultObserver` here. These specific observers react by applying appropriate propagation algorithms when they are notified. Many operations are *unary*: they have a single source `Box`, but in general operations may be n-ary and have several source or result boxes.

Secondly, Fig. 1b represents the instantiation of the classes shown on Fig. 1a to represent expression `b = a->collect(λ)`. There are two instances of the `Box` class called `a` and `b`, which are connected by an instance called `c` of class `Collect`. `Box a` is observed by a `SourceObserver` belonging to `c`, and `box b` is observed by a `ResultObserver` belonging to `c` as well. For instance, when the `SourceObserver` of `c` is notified of addition of element `e` to `a`, it correspondingly adds `lambda.apply(e)` to `b`. We call this *forward propagation*. Conversely, for *reverse propagation* of changes on the result into changes on the source, when the `ResultObserver` of `c` is notified of addition of element `e` to `b`, it correspondingly adds `reverseLambda.apply(e)` to `a`. For simplification purposes, the remainder of this paper will only consider forward propagation, but what we present also applies to reverse propagation.

Applying active operations to models is just a matter of specifying what the boxes are. Modeling frameworks like EMF offer mechanisms to observe changes to the properties of model elements. Therefore, we wrap every model element property by an observable `Box`. For the time being, we suppose that there are no concurrent changes on a model: only one box is modified at a time, and changes happen sequentially. We call this the *no-concurrency hypothesis*. This is how EMF works by default.

Chaining multiple operations results in a directed acyclic graph structure where source boxes are *roots*. With the non-concurrency hypothesis, only one root box may change at a single time. Every box may be consumed by an operation, and every non-source box is produced by exactly one operation. We call this structure a *propagation graph* because changes are propagated along this graph. It is implicit because there is no global representation of it: boxes only know of their immediate observers, and operations only know of their source and result boxes.

2.2 Motivating Example with Propagation Issues

Observation-based propagation as presented above is based on the well-known *Observer* design pattern, and is relatively simple to understand and to implement. However, it has some problems handling non-unary operations when their input depend from the same root box.

¹ Changes on ordered collections such as `Sequence` and `OrderedSet` are handled by `Observer` methods not shown here. Those methods additionally take the index at which the change occurs. Moreover, other kinds of changes like replacing, or moving are also supported.

An example of a binary operation is `zip`², which takes two ordered collections (e.g., OCL's `Sequence` or `OrderedSet`) as input: its source collection, and an other collection passed as an argument. From these two collections, `zip` produces a single collection of pairs consisting of one element (that we will call `left`) from its first source collection, and one element (that we will call `right`) from its argument collection. It traverses both collections in order, and therefore pairs elements having the same index. The resulting collection is as long as the shortest of the two input collections. If one of the input collections is longer, its tail is ignored. The signature of its `Sequence`-based version is as follows (with implicit type parameters `L` and `R`):

```
context Sequence(L) def: zip(b : Sequence(R)) :
    Sequence(Tuple(left:L,right:R))
```

`zip` can be used in the following way:

```
let a = Sequence {1, 2, 3, 4} in
a->collect(e | e * 2)->zip(a->collect(e | e * 3))
```

which results in the following value:

```
Sequence {Tuple {left = 2, right = 3}, Tuple {left = 4, right = 6},
    Tuple {left = 6, right = 9}, Tuple {left = 8, right = 12}}
```

In later examples, we will rather use the Haskell notation because OCL tuples are very verbose³:

```
[(2, 3), (4, 6), (6, 9), (8, 12)]
```

Furthermore, in order to better visualize propagation, we simplify the expression by giving symbolic names to the lambdas given as argument to `collect`:

```
a->collect(λ1)->zip(a->collect(λ2))
and we give names to every intermediate value:
```

```
def: f(a) =
    let b = a->collect(λ1) in
    let c = a->collect(λ2) in
    let d = b->zip(c) in
    d
```

Figure 2 gives several representations of the corresponding implicit propagation graph at various steps of the propagation of a single change. Boxes are represented as rectangles, and operations are represented as ellipses. An arrow from a box to an operation denotes that this box is a source of that operation.

² This operation is well-known from functional programming, but is notably not present in OCL. We use it as an example because it enables illustrating alignment issues. Moreover, it can be leveraged to implement some operations from the OCL standard library instead of implementing each of them separately.

³ OCL tuples are actually more similar to Haskell records than to its tuples.

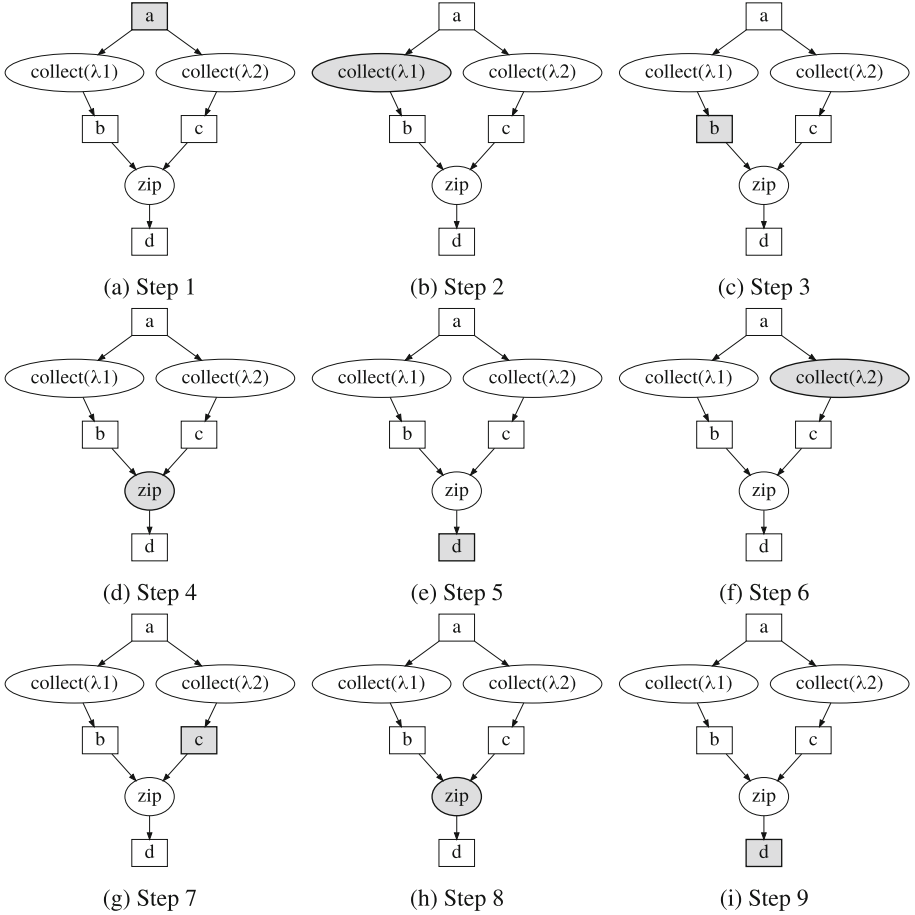


Fig. 2. Depth-first propagation with zip

Similarly, an arrow from an operation to a box denotes that box is a result of this operation. These arrows may be considered as abstractions of **Observers**, which makes this representation more compact than the one from Fig. 1b. Because the list of observers of a box is ordered, we further need to specify that left-most observers are notified first. For instance, box **a** will first notify `collect(λ.1)`, and then `collect(λ.2)`. Propagation happens in a sequence of steps, which Fig. 2 shows as multiple copies of the propagation graph with the current active step (box or operation) highlighted (i.e., filled with gray).

Let us consider that initialization has already been performed, and that boxes **a**, **b**, **c**, and **d** already have the following values:

- a** = [1, 2, 3, 4] **b** = [2, 4, 6, 8]
- c** = [3, 6, 9, 12] **d** = [(2, 3), (4, 6), (6, 9), (8, 12)]

Let us further consider the following change on box **a**: removal of 3. We are now in step 1 shown on Fig. 2a, and the boxes have the following values:

a = [1, 2, 4] b = [2, 4, 6, 8]
 c = [3, 6, 9, 12] d = [(2, 3), (4, 6), (6, 9), (8, 12)]

These values are inconsistent with respect to the expression that relates them, which is normal during change propagation. The whole point of incremental evaluation is to make them consistent again. The source observer of the first `collect` is notified (step 2, Fig. 2b), and it modifies box **b** (step 3, Fig. 2c) such that the boxes now have values:

a = [1, 2, 4] b = [2, 4, 8]
 c = [3, 6, 9, 12] d = [(2, 3), (4, 6), (6, 9), (8, 12)]

Then the left observer of `zip` is notified by box **b** (step 4, Fig. 2d), and it modifies box **d** (step 5, Fig. 2e):

a = [1, 2, 4] b = [2, 4, 8]
 c = [3, 6, 9, 12] d = [(2, 3), (4, 6), (8, 9)]

The reader may already notice that box **d** has been changed, whereas box **c** still contains its initial value. This results in: (1) a transitory state that a user may notice (e.g., if box **d** is displayed in a user interface), and (2) an inconsistent state (8 and 9 are not the double and triple of the same number) that a user may notice as well. We call this specific kind of issue and *alignment issue* because it is due to boxes **b** and **c** becoming unaligned. Then box **a** notifies its second observer, which is the source observer of the second `collect` (step 6, Fig. 2f), which in turn modifies box **c** (step 7, Fig. 2g), resulting in:

a = [1, 2, 4] b = [2, 4, 8]
 c = [3, 6, 12] d = [(2, 3), (4, 6), (8, 9)]

Finally, the right observer of `zip` is notified by box **c** (step 8, Fig. 2h), and it updates box **d** (step 9, Fig. 2i), which concludes the propagation of this change with:

a = [1, 2, 4] b = [2, 4, 8]
 c = [3, 6, 12] d = [(2, 3), (4, 6), (8, 12)]

This final state is finally consistent with the expression that relates the values of the boxes.

It should be noted that such issues as presented here only occur because the two inputs to `zip` are computed from a single root box (**a**). If the two inputs only depended on two distinct root boxes, the no-concurrency hypothesis would prevent such issues from taking place.

2.3 Summary of Propagation Issues

Previous sections have explained how observer-based propagation works, as well as some of its issues. These are mostly about unwanted transitory states caused by changes on source boxes of n-ary operations being processed separately instead of together. For instance, on the previous `zip` example, step 4, Fig. 2d, and step 5, Fig. 2e, do not actually need to happen before box `c` has been updated. These steps could be removed, and the last two steps (step 8, Fig. 2h, and step 9, Fig. 2i) could perform all the required changes to box `d` in one go. Change propagation performed by `zip` should all happen together. These transitory states have the following three consequences:

Issue-1: Extra computations are required for change propagation, which slightly reduces performance.

Issue-2: Change amplification occurs: a single input change is mapped to multiple target changes, which may be user-observable.

Issue-3: Transitory inconsistencies may also become user-observable, as was noted after step 5, Fig. 2e of the previous `zip` example.

About **Issue-3**: we previously stated when describing step 1, Fig. 2a that transitory inconsistencies are necessarily to be expected. This is true for *internal* inconsistencies occurring between intermediate values (such as boxes `b`, or `c`), and input or output values (such as boxes `a`, or `d`). However, externally observable inconsistencies such as when box `d` contains the pair `(8, 9)` are generally undesirable.

2.4 Workarounds

Of the three issues presented previously, **Issue-3** about externally observable transitory inconsistencies is the only one that absolutely requires to be addressed. Consequently, all workarounds presented in this section prevent it from happening. Furthermore, each workaround is accompanied by a statement about how well it handles the two other issues: **Issue-1** about performance, and **Issue-2** about change amplification.

1. **Post-filtering** consists in adding an additional `select` into the expression after the n-ary operation, in order to filter out inconsistencies. For instance, in the previous `zip` example, we could rewrite the expression as:

```
a->collect(e | e * 2)->zip(a.collect(e | e * 3))
  ->select(e | e.right = 3 * e.left / 2)
```

This would not prevent internal inconsistencies, but would prevent them from appearing downstream of the `select`, and therefore from being externally visible. Locally, this will require more computations to evaluate the `select`, and its predicate. Globally, it may increase performance by reducing change amplification, which may reduce the amount of work required of downstream operations, if any. However, this depends on the proportion of inconsistent changes among those resulting from amplification.

2. **Making dependencies explicit** to n-ary operations. For instance, on the previous example, one can notice that every change to box **a** will result into one change on box **b**, and one change on box **c** in that order (because the observer list is ordered that way). Such a situation is common because of how operations are assembled (i.e., only after their source boxes have been created by upstream operations), and of how observation works. Therefore, the `zip` operation offered by AOF takes an additional parameter indicating whether its argument box has such a dependency to its source box. If that is the case, it only reacts to changes once they reach its argument box, which happens after its input box has already been updated. This workaround also reduces change amplification downstream, and therefore increases performance. However, figuring out which input box will get notified first is not always trivial without static analysis.
3. **Reducing arity** of n-ary operations by specializing them, ideally turning them into unary ones. For instance, on the previous `zip` example, we could define a single `collectZip` operation that performs the two collects, and the `zip`. This approach addresses all three issues, but requires the definition of new ad-hoc operations, and expertise from the user who needs to know when to use them (although this could probably be automated by static analysis).

In practice, we have successfully used both post-filtering, and making dependencies explicit with AOF. We have not tried arity reduction because of its cost to the user, but also because of its maintenance cost for the new ad-hoc operations.

3 Explicit Propagation Graph

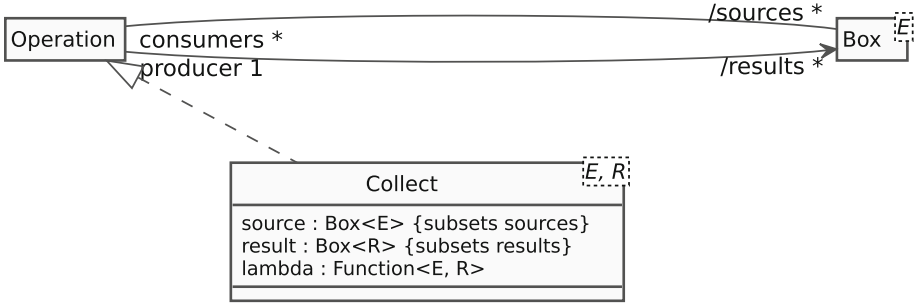
3.1 Overview

Propagation issues presented in the previous section are caused by the use of observation. Indeed, with such an approach every operation only knows about what happens locally, immediately around it (i.e., to the boxes it observes), and not globally. Therefore, a better solution than the presented workarounds would be to improve propagation algorithms to take into account the global structure of the propagation graph. With observation, this propagation graph is implicit, and only results from the local connections between observables and observers. The first step is to make this graph explicit, which will open many algorithm improvement opportunities.

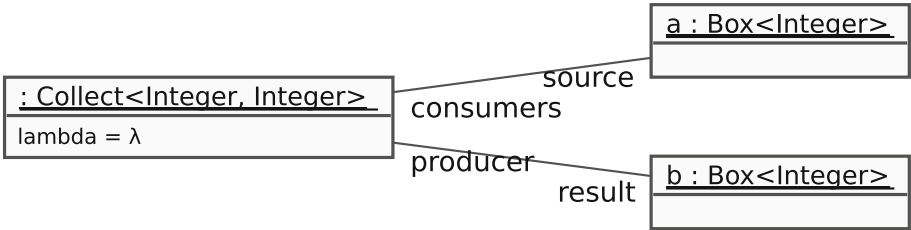
Figure 3a depicts an excerpt of an explicit propagation graph metamodel. Note that, whereas Fig. 1a was merely a class diagram representing a corresponding implementation (e.g., in Java for AOF) of observation, the diagram of Fig. 3a represents a metamodel. A notable change from Fig. 1a is the disappearance of observers. The metamodel only captures the graph structure, and ignores technical artefacts. It is independent of the way propagation will occur. Every operation has `Boxes` as `sources`, for which it is part of their `consumers`,

and has Boxes as **results**, for which it is their **producer**. We have implemented our explicit propagation graph metamodel using EMF.

Figure 3b shows a model conforming to the propagation graph metamodel, and representing expression $a \rightarrow \text{collect}(\lambda)$. Here again, when compared to the observation implementation from Fig. 1b, one can notice the disappearance of observers. Only the essential graph structure remains. Apart from the lambda, which is still opaque (i.e., not detailed in the model), the structure of this model is similar to the structure of OCL abstract syntax.



(a) Propagation graph metamodel



(b) Propagation graph model of $b = a \rightarrow \text{collect}(\lambda)$

Fig. 3. Diagrams for the explicit propagation graph

3.2 Propagation Algorithms

Making the propagation graph explicit separates it from the implementation of propagation algorithms. Moreover, we can now also decouple operation-local propagation algorithms from the propagation strategy, which is linked to how we traverse the graph during propagation. Indeed, observation imposed a depth-first traversal of the propagation graph, but we can now define any graph traversal of our choosing.

Given an explicit propagation graph, a traversal strategy may be computed using two main categories of approaches:

- **Static** approaches precompute traversal strategies ahead of propagation time. This may happen at compile time, but it may also happen during initial

expression evaluation. What distinguishes static approaches from dynamic approaches is that a traversal strategy has already been selected before the first change propagation occurs, and it remains the same for every change propagation. Change propagation can be faster at the cost of a slower initialization.

- **Dynamic** approaches compute a traversal strategy for each change propagation. Initialization can be faster, at the cost of slower change propagation. Such a strategy may be simpler to implement than a static strategy.

An example of a dynamic strategy consists in doing two traversals for each change propagation. The first traversal does not propagate the change, but is only used to discover every box that may be impacted, and to mark it. During the second traversal, each operation waits for all of its marked inputs to be updated before executing its local propagation algorithm. Waiting for all inputs to be updated is not sufficient because a given change propagation may not impact all inputs. Therefore, without the first marking traversal, an operation could be stuck waiting for an input that will not be updated. We have implemented this simple strategy, and have confirmed that it solves the propagation issues mentioned earlier. Actually, it should even be possible to implement such a strategy by extending an observation-based implementation. However, this would have required changes to the implementation of every operation. As mentioned above, having an explicit propagation graph decouples operation-local propagation algorithms from the traversal strategy, which makes it possible to consider and implement various traversal strategies without changing operation-local algorithms.

An example of a static strategy is the following. We remarked in Sect. 2.3 that an n -ary operation should only perform propagation once all its sources have been updated. Consequently, we need to find a traversal that satisfies this condition. Two given operations o_1 and o_2 may only be related in the three following ways because the propagation graph is acyclic:

1. o_1 is upstream from o_2 and must be processed before it,
2. o_2 is upstream from o_1 and must be processed before it, or
3. o_1 and o_2 can be processed in any order.

We can thus define a partial order between operations. Therefore, a valid propagation graph traversal can be computed using a topological sort of the set of operations using this partial order. Such graph traversals are breadth-first, and not depth-first like it is with observation. We are currently implementing this strategy, with initial experiments showing that it also solves the propagation issues mentioned in Sect. 2.3.

3.3 Application to Motivating Example

Figure 4 shows an example of a static strategy based on a topological sort. Because of the graph structure, there are two possible orderings: `[collect(λ 1)`,

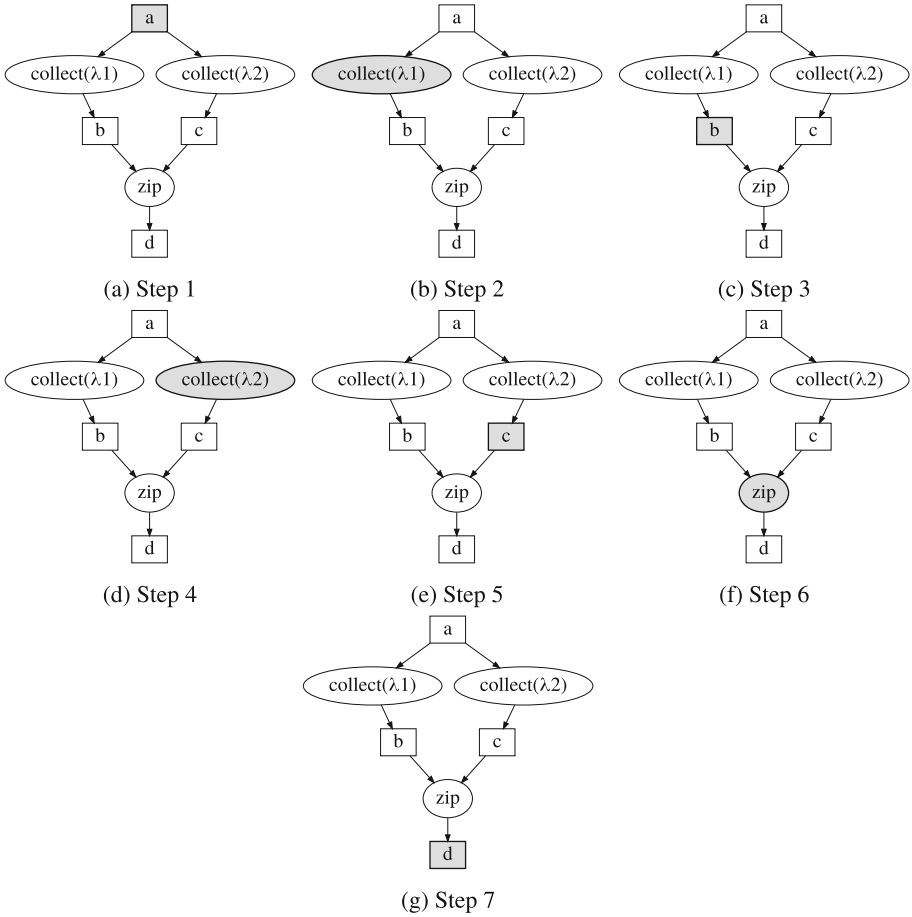


Fig. 4. Breadth-first propagation with `zip` based on a topological sort

`collect(λ.2), zip]`, or `[collect(λ.2), collect(λ.1), zip]`. We arbitrarily chose the first one.

Change propagation starts at step 1, Fig. 4a from the same initial state, and with the same change (i.e., removal of 3 from box a) as for the observation-based approach in Sect. 2.2. Then the first `collect` processes the change (step 2, Fig. 4b), and updates box b (step 3, Fig. 4c). We are now in the same state as step 3, Fig. 2c of the observation-based approach. The next step marks the distinction between the two approaches. The second `collect` processes the change (step 4, Fig. 4d), and updates box c (step 5, Fig. 4e). At this stage, the boxes have the following values:

$$\begin{array}{ll}
 a = [1, 2, 4] & b = [2, 4, 8] \\
 c = [3, 6, 12] & d = [(2, 3), (4, 6), (6, 9), (8, 12)]
 \end{array}$$

Finally, now that both its inputs are updated, `zip` processes the change (step 6, Fig. 4f), and updates box `d` (step 7, Fig. 4g). The final value is the same as for the observation-based approach. However, with the breadth-first traversal enabled by the explicit graph, box `d` has never been in any inconsistent state, even transitorily.

3.4 Discussion

Firstly, another issue with an approach based on an implicit graph is that it is not easy to visualize and debug it. As a matter of fact, having an explicit graph enables the creation of many useful tools, such as visualizers (e.g., via transformation of the graph model to a visualization model), as well as a variety of graph analyzers (e.g., to optimize it).

Secondly, an observation-based implicit graph is necessarily extensional: several calls to the same operation with different arguments will result in duplicate subgraphs, one for each call. It basically inlines all calls. With an explicit graph, it becomes possible to consider having an intensional representation where operation calls are represented explicitly, without forcing to inline them. This results in a more compact representation of the propagation graph. Moreover, some operations like `collect`, and `select` may be given lambdas that return mutable values: the result of the lambda may change even if the argument does not (for instance because the lambda accesses a mutable field of a model element). In such a case, the lambda itself must also be represented as a subgraph. If such lambda subgraphs are represented extensionally, there must be a copy of it for every element on which it is applied. The propagation graph then grows linearly in size with the length of the source box. Having an intensional representation enables saving some memory, although there still needs to be a node that represents the subgraph call.

4 Related Work

The VIATRA [6] approach offers incremental query evaluation, and model transformation. VIATRA scales well, as has been shown on several benchmarks including [7], developed by the VIATRA team, and on which active operations have been shown to scale as well [5]. It is based on RETE [8], which works with a pattern matching network/graph. It therefore has an explicit graph representation like our proposal. However, its incrementality is based on a different algorithmic approach: a pattern matching network, whereas our propagation graph is basically a data flow graph structured similarly to the OCL metamodel. Although they can translate a subset of OCL to their pattern language [9], the dissimilar structure is probably one reason why they are limited to this subset of OCL, which is not an issue for our approach. Also, as already explained in [5], VIATRA cannot preserve collection order, whereas active operations can.

Current work in progress on the implementation of QVT using micro-mappings [10, 11] exhibits some promising initial results. However, although it

scales particularly well on simple benchmarks, it has not been shown to work on closer to real-world benchmarks like [7] yet. The micro-mappings approach also works using a graph representation of a transformation, which is a kind of dataflow between actionable pattern matching mappings (i.e., patterns with a matching part and a creation part). But the structure of this graph is neither close to the OCL abstract syntax nor to the QVT abstract syntax. Furthermore, the optimization of micro-mappings is typically done ahead of time.

The work presented in [12] has some similarities with active operations in that it considers operation-local algorithms, and builds complex expressions by composing these operations. However, it is implemented in Haskell, which is a purely functional language, whereas our framework AOF is implemented in Java, and works with EMF. The performance and scalability of such an approach is unclear at this time. Their approach does not have an explicit graph representation, and appears to follow a depth-first propagation strategy. Therefore, we expect that it will have similar problems to the ones we encountered with observation.

5 Conclusion

The observation-based approach used in the current implementation of active operations for incremental evaluation has been presented and criticized. Its main flaws have been presented, and illustrated on a simple motivating example. Methods to work around these flaws have been presented, but getting rid of them required to rethink the approach.

A novel approach based on an explicit propagation graph has been presented, and applied to the motivating example. This new approach cleanly decouples the propagation graph from, on one hand, its traversal strategy, and on the other hand, the local operation-specific propagation algorithms. Two possible traversal strategies have been presented, one dynamic, and one static.

Given the advantages of having an explicit propagation graph, we plan to further our implementation effort in this direction, in order to support all features supported by the current observation-based implementation but with more efficient foundations.

References

1. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. *Computer* **42**(10), 22–27 (2009)
2. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.-M.: Active operations on collections. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) *MODELS 2010*. LNCS, vol. 6394, pp. 91–105. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_7
3. Jouault, F., Beaudoux, O.: On the use of active operations for incremental bidirectional evaluation of OCL. In: *15th International Workshop on OCL and Textual Modeling (OCL 2015)*. *OCL and Textual Modeling: Tools and Textual Model Transformations*, vol. 1512, Ottawa, Canada, pp. 35–45, September 2015

4. Beaudoux, O., Jouault, F.: Bidirectional incremental transformations with active operation framework - application to facades. In: 1st Papyrus Workshop - DSML Technologies (CEA), Toulouse, France, June 2015
5. Jouault, F., Beaudoux, O.: Efficient OCL-based incremental transformations. In: 16th International Workshop in OCL and Textual Modeling, Saint-Malo, France, pp. 121–136, October 2016
6. 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)
7. IncQuery Labs Ltd.: Performance benchmark using the VIATRA CPS demonstrator. <https://github.com/viatra/viatra-cps-benchmark>
8. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artif. Intell.* **19**(1), 17–37 (1982)
9. Bergmann, G.: Translating OCL to graph patterns. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Infran, E. (eds.) *MODELS 2014*. LNCS, vol. 8767, pp. 670–686. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11653-2_41
10. Willink, E.: Optimized declarative transformation - first eclipse QVTc results. In: *Proceedings of BigMDE 2016: Workshop on Scalability in Model Driven Engineering*, July 2016
11. Willink, E.D.: Local optimizations in eclipse QVTc and QVTr using the micro-mapping model of computation. In: 2nd International Workshop on Executable Modeling, Saint-Malo, France, pp. 26–32, October 2016
12. Firsov, D., Jeltsch, W.: Purely functional incremental computing. In: Castor, F., Liu, Y.D. (eds.) *SBLP 2016*. LNCS, vol. 9889, pp. 62–77. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45279-1_5

Translating UML-RSDS OCL to ANSI C

Kevin Lano¹(✉), Sobhan Yassipour-Tehrani¹, Hessa Alfraihi¹,
and Shekoufeh Kollahdouz-Rahimi²

¹ Department of Informatics, King's College London, London, UK
`kevin.lano@kcl.ac.uk`

² Department of Software Engineering, University of Isfahan, Isfahan, Iran

Abstract. In this paper we describe a transformation from a subset of OCL to ANSI C code, and we show that the generated code provides improved efficiency compared to OCL execution using Java, C# or C++. The transformation is itself formally specified using OCL.

1 Introduction

In this paper we describe a transformation that maps a substantial subset of OCL 2.4 [7] to ANSI C. C has limited expressiveness compared to more modern languages such as Java or C#, but it has the benefits of high efficiency and small code size. The OCL to C translator is a subtransformation of a UML to C code generator, UML2C, for the UML-RSDS MDE language [6]. UML2C maps instance models of the UML-RSDS class diagram (Fig. 1), OCL (Fig. 3) and activities metamodels to C language metamodels (Fig. 2 and C expression and statement metamodels). We target the 1989 ANSI C standard [4].

The mapping of OCL expressions depends upon the prior mapping of types and class structures, however it is relatively independent of the strategy chosen for representing classes in C (eg., how inheritance and dynamic dispatch is expressed), since all access to objects and their features is via an interface of creators, lookup operations and getters/setters which have a standard signature independent of their implementation details. For example, any instance-scope attribute f of class C is accessed via operations $getC_f$ and $setC_f$, for both owned and inherited attributes (cf., Table 3, case F1.2.5). The application API is defined in the header file `app.h` for each application. A library `ocl.h` of C functions for OCL operators is also defined, and evaluation/execution of particular OCL expressions is based upon `app.h` and `ocl.h`.

1.1 UML-RSDS

UML-RSDS enables applications to be defined using class diagrams, use cases, constraints and activities (pseudocode). It is similar to fUML [8] in being a subset of UML, however, unlike fUML, it is oriented to declarative specification, with OCL constraints being used to define use cases and operations by pre and post conditions, instead of activities. The UML-RSDS tools can synthesise a

procedural platform-independent design from such specifications, and this design is then mapped to program code by code generators (3 generators exist for Java versions, and there are C++ and C# generators in the latest UML-RSDS version 1.7 at nms.kcl.ac.uk/kevin.lano/uml2web).

Specifiers are recommended to optimise their application functionality *at the specification level*, eg., by using let-variables to avoid duplicated expression evaluations. These optimisations then apply regardless of the eventual target platform. Optimisation is also performed during the design synthesis stage, eg., to use bounded loops instead of fixpoint iteration where possible [6].

The tools have been extensively used since 2006, particularly in the financial domain and for defining transformations. There are a number of restrictions and variations in the language compared to full UML and OCL (Table 1). We have found these variations helpful in simplifying specifications and improving the capability for verifying specifications.

Table 1. Differences between UML-RSDS and UML

<i>UML/OCL</i>	<i>UML-RSDS subset/variant</i>
Ternary associations, Multiple inheritance	Omitted
General n..m multiplicities on association end	Only 1, 0..1, or * multiplicities permitted
Integer type	int, long computational numeric types
Real type	double computational type
null, invalid	Omitted
OclMessage, Tuple	Omitted
Implicit conversion of single elements to collections	Omitted. 0..1 association ends are treated as collections
4-valued logic	Classical 2-valued logic
General <i>iterate</i>	Omitted
OclAny, oclType()	Omitted
	Lookup of objects by primary key value E[value] Additional collection operators $\rightarrow sort()$, $\rightarrow front()$, $\rightarrow tail()$, etc

Collections are assumed not to contain null elements. String-valued attributes can be declared as *identity* attributes, i.e., as primary keys for a class. Classes with a subclass must be abstract.

Minor syntactic variations are the use of \implies for OCL *implies*, $\rightarrow exists1$ for $\rightarrow one$, and $\&$ for *and*. *E.allInstances* is abbreviated to *E* when used as the LHS of a \rightarrow operator. $s \rightarrow includes(x)$ can also be written as $x : s$, $s \rightarrow includesAll(x)$ as $x <: s$, and $s \rightarrow excludes(x)$ as $x / : s$.

1.2 Paper Structure

Section 2 describes the mapping of types and class structures to C, Sect. 3 describes the OCL expression mapping, Sect. 4 gives an evaluation, Sect. 5 describes related and future work, and Sect. 6 gives conclusions.

2 Mapping of Types and Classes

Figure 1 shows part of the UML-RSDS class diagram metamodel. This is closely based upon UML 2.2. Instances of this metamodel are mapped to instances of a C metamodel by UML2C. The base target language is a simplified version of the abstract syntax of C programs (Fig. 2).

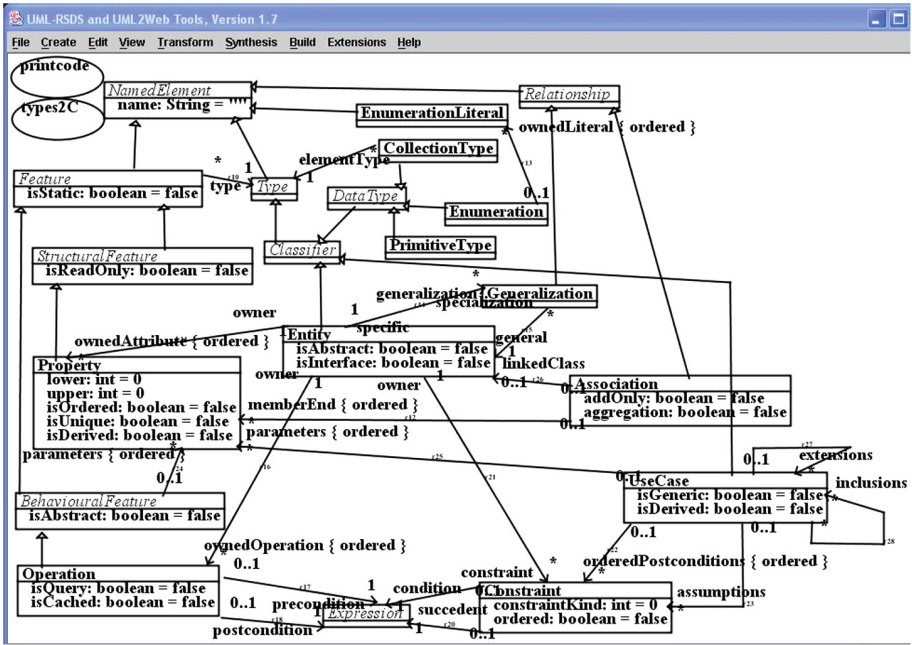


Fig. 1. UML class diagram metamodel (subset)

Table 2 shows the informal mapping of UML *Types* to C. The T* operator directly interprets Collection (T), for sequences and ordered sets of string and entity types T. Collections of collections can be mapped down to 2 levels (eg., Sequence (Sequence(double)) for matrices is mapped to double**). Unordered sets and bags are implemented as binary search trees.

To achieve bidirectionality and traceability of the transformation, a new identity attribute `typeId : String` was introduced into *Type*, and `ctypeId` into *CType*. This enables *Type* and *CType* instances to be looked-up by key value:

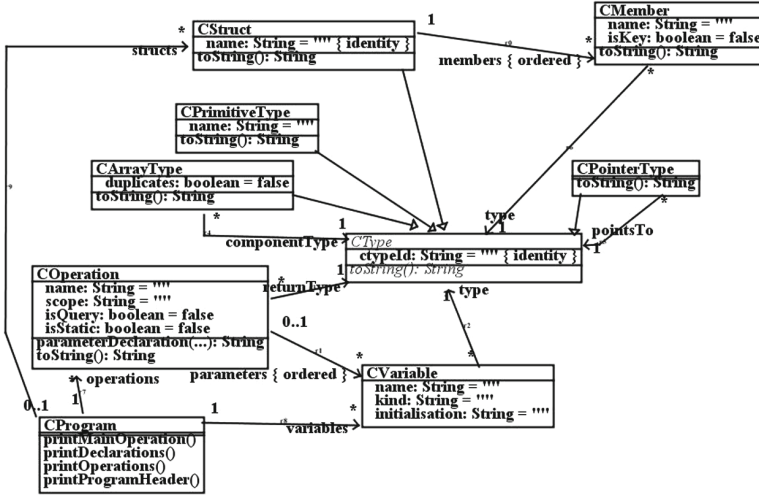


Fig. 2. C language metamodel

Table 2. Informal mappings of types to C

Case	UML/OCL element	C representation e'
F1.1.1.1	String type	char*
F1.1.1.2	int, long, double types	same-named C types
F1.1.1.3	boolean type	unsigned char
F1.1.3	Entity type E	struct E* type
F1.1.4.1	OrderedSet(T) type	T' * (NULL-terminated array of T', without duplicates)
F1.1.4.2	Sequence(T) type	T' * (NULL-terminated array of T', possibly with duplicates)
F1.1.4.3	Set(T) type	BSTs of T' elements without duplicates
F1.1.4.4	Bag(T) type	BSTs of T' elements possibly with duplicates

$CType[id]$ is the C type instance t with $t.ctypeId = id$. An instance $t : Type$ corresponds to an instance $c : CType$ if $t.typeId = c.ctypeId$.

An example transformation rule of the UML2C transformation, formalising case F1.1.1.1 from Table 2, is:

```

PrimitiveType::
  name = "String" =>
    CPointerType->exists( p | p.ctypeId = typeId &
      CPrimitiveType->exists( c | c.name = "char" & p.pointsTo = c ) )
    
```

This rule applies to objects $self : PrimitiveType$. Whenever the lhs of the rule is true, the rhs is made true, i.e., the relevant C types are looked-up or created if they do not already exist. The semantics of $E \rightarrow exists(e | e.eId = v \ \& \ P)$ in the case that eId is an

identity attribute of E is that the E object $E[v]$ with eId value equal to v is looked up, if it exists, and is then modified according to P . If the object does not exist, it is created and then modified.

Classes, features and inheritances are mapped to C as shown in Table 3.

Table 3. Informal mapping of UML class diagrams to C

<i>Case</i>	<i>UML element e</i>	<i>C representation e'</i>
F1.2.1	Class diagram D	C program with D 's name
F1.2.2	Class E	struct E { ... }; Global variable struct E** e_instances; struct E* createE(void) operation
F1.2.3.1	Instance property $p : T$ (not principal identity attribute)	Member T' p; of the struct for p 's owner, E where T' represents T Operations T' getE_p(E' self) and setE_p(E' self, T' px)
F1.2.3.2	Principal identity attribute $p : String$ of class E	Operations getE_p, setE_p, struct E* getEByPK(char* v) Key member char* p; of the struct for E
F1.2.4	Operation $op(p : P) : T$ of E (non-static)	C operation T' op_E(E' self, P' p) with scope = "entity"
F1.2.5	Inheritance of A by B	Member struct A* super; of struct B Operations getB_att(x) for inherited att invoke getA_att(x→super) Operations op_B(x, p) for inherited op invoke op_A(x→super, p) unless redefined in B
F1.2.6	Operation $op(p : P) : T$ of E (static)	C operation T' op(P' p) with scope = "entity"

For each entity type E , getters and setters for each feature of E are produced, together with creation and deletion operations createE and killE, and lookup operations getEByPK, getEByPKs in the case that E has a principal primary key (identity attribute). These form the object API for E. Operations for OCL collection operators acting on collections of E instances are also generated: collectE, selectE, rejectE, intersectionE, unionE, reverseE, frontE, tailE, asSetE, concatenateE, removeE, removeAllE, subrangeE, isUniqueE, insertAtE. An operation opE is only generated for OCL operator op if there is an occurrence of $\rightarrow op$ applied to a collection of E elements in the source UML/OCL specification model.

2.1 Mapping of Associations and Polymorphic Operations

We have found that the most complex parts of UML to code mappings are typically: (i) managing object deletion; (ii) maintaining the consistency of opposite association ends. Additionally for C, expressing inheritance and dynamic dispatch are further complex aspects. Deletion and association management operations are created during design

synthesis. If an association has both ends named, then these ends need to be maintained in consistency. For example, a $*-*$ association between classes A and B , with ends ar , br will have synthesised design operations

```
A::
static addA_br(ax : A, bx : B)
activity:
    ax.br := ax.br->including(bx) ;
    bx.ar := bx.ar->including(ax)
```

```
A::
static removeA_br(ax : A, bx : B)
activity:
    ax.br := ax.br->excluding(bx) ;
    bx.ar := bx.ar->excluding(ax)
```

and similarly for other association multiplicities. Deletion operators $killE$ for concrete E are also inserted into the design, these manage the deletion of aggregation part objects linked to the deleted object, and the removal of the object from all association ends. The UML2C generator therefore generates C declarations and code for these operations.

General schemes for representing inheritance in C include an embedded superclass struct instance in each subclass struct, and function pointers for each supported method, or the use of vtables for function pointers. We use a pointer member `struct E* super`; referring from a subclass F to its superclass E .

Dynamic dispatch of an abstract operation $op(p : P) : Rt$ of class E with leaf subclasses A, B, \dots is carried out by a C operation op_E with the schematic definition

```
Rt' op_E(struct E* self, P' p)
{ if (oclIncludes((void**) a_instances, (void*) self))
  { return op_A((struct A*) self, p); }
  else if (oclIncludes((void**) b_instances, (void*) self))
  { return op_B((struct B*) self, p); }
  else ...
}
```

This explicit selection of the correct implementing operation corresponds to the semantic model of polymorphic operations used by the UML-RSDS verification tools¹.

3 Mapping of UML-RSDS OCL Expressions to C

Figure 3 shows the UML-RSDS OCL metamodel, which is the source language for the transformation. A similar metamodel defines the corresponding C expression language abstract syntax. New identity attributes $expId$ and $cexpId$ are added to $Expression$ and $CExpression$, respectively, to support bidirectionality and traceability requirements. $variable : String$ represents iterator variables x for the cases of $s \rightarrow forAll(x|P)$, etc. A $*-*$ association $context$ from $Expression$ to $Entity$ is used to record the context(s) of use of the expression.

¹ The operation versions should have the same signatures, overloading is not supported for this translation.

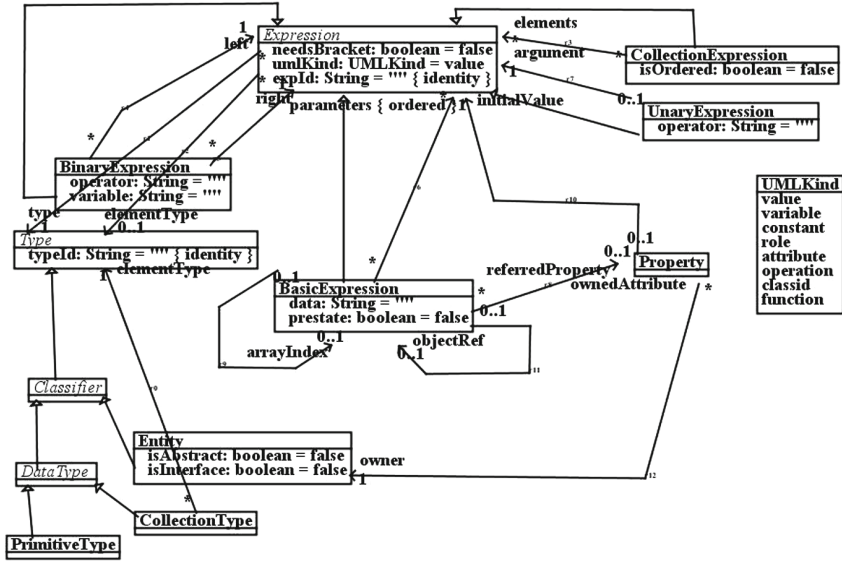


Fig. 3. UML-RSDS OCL metamodel

The mappings are divided into four subgroups: (i) mapping of basic expressions; (ii) mapping of logical expressions; (iii) mapping of comparator, numeric and string expressions; (iv) mapping of collection expressions.

The basic expressions of OCL generally map directly to corresponding C basic expressions. Table 4 shows examples of the mapping for these.

Table 4. Mapping specifications for basic expressions

UML-RSDS OCL expression e	C representation e'
<i>self</i>	<i>self</i> as an operation parameter
Data feature f of context E with no objectRef	$getE_f(self)$
E data feature f of instance ex	$getE_f(ex')$
Operation call $op(e1, \dots, en)$ or $obj.op(e1, \dots, en)$ of instance entity scope op of E	$op_E((struct E^*) self, e1', \dots, en')$ $op_E((struct E^*) obj', e1', \dots, en')$
Call $op(e1, \dots, en)$ of static/application scope op	$op(e1', \dots, en')$
E attribute/role f of collection exs	$getAllE_f(exs')$ (duplicate values preserved)
$E[v]$ v single-valued	$getEByPK(v')$
$E[vs]$ vs collection-valued	$getEByPKs(vs')$
$E.allInstances$ concrete E	$e_instances$
boolean true, false	TRUE, FALSE

Table 5 shows the mapping of logical expressions and operators to C.

Table 5. Mapping specifications for logical expressions

<i>UML-RSDS OCL expression e</i>	<i>C expression e'</i>
A & B	A' && B'
A or B	A' B'
not(A)	!A'
E->exists(P)	existsE(e_instances, fP) fP evaluates P'
e->exists(P)	existsE(e', fP)
E->forall(P)	forallE(e_instances, fP) fP evaluates P'
e->forall(P)	forallE(e', fP)

The auxiliary operations fP are constructed to only have a single parameter, this means that mapping of forAll, select, etc. is only supported where the rhs expressions depend on a single variable. The alternative (used in the UML-RSDS Java, C#, and C++ translators) is to create a specialised iterator implementation for each different use of an iterator operation.

Tables 6 and 7 show examples of the values and operators that apply to ordered sets and sequences, and their C translations. Some UML-RSDS OCL operators (union-All, intersectAll, symmetricDifference, subcollections) were considered a low priority, because these are infrequently used, and were not translated.

Table 6. Translation of collection unary operators

<i>Expression e</i>	<i>C translation e'</i>
x->size()	oclSize((void**) x')
x->reverse()	reverseE(x')
x->sort()	(struct E**) oclSort((void**) x', compareTo_E) x of entity element type E (char**) oclSort((void**) x', compareTo_String) x of String element type
x->sum()	sumString(x',n), sumint(x',n), sumlong(x',n), sumdouble(x',n) n is the size of x
x->max()	maxint(x',n), maxlong(x',n), maxdouble(x',n), or maxString(x',n) n is the size of x.

A common form of OCL expression is the evaluation of a reduce operation (min, max, sum, prd) applied to the result of a collect, eg.: $s \rightarrow collect(e) \rightarrow sum()$ where e is double-valued. This is mapped to:

$$sumdouble(collectE_double(s', fe), oclSize((void **) s'))$$

Table 7. Translation of binary collection operators (s of entity element type E)

<i>Expression e</i>	<i>C translation e'</i>
<code>s->including(x)</code>	<code>insertE(s',x')</code> or <code>appendE(s',x')</code>
<code>s->excluding(x)</code>	<code>removeE(s',x')</code>
<code>s - t</code>	<code>removeAllE(s',t')</code>
<code>s->append(x)</code>	<code>appendE(s',x')</code>
<code>s->count(x)</code>	<code>oclCount((void**) s', (void*) x')</code>
<code>s->at(i)</code>	<code>(struct E*) (s')[i'-1]</code>
<code>s->indexOf(x)</code>	<code>oclIndexOf((void**) s', (void*) x')</code>
<code>s->union(t)</code>	<code>unionE(s',t')</code>
<code>s->intersection(t)</code>	<code>intersectionE(s', t')</code>
<code>s->sortedBy(e)</code>	<code>(struct E**) oclSort((void**) s', comparee)</code> comparee defines e-order on E objects

because it is not possible to find the length of a collection of primitive values. Likewise, *s.att.sum* is mapped to *sumdouble(getAllE_att(s'), oclSize((void**) s'))*. For a literal sequence *s*, *oclSize(s')* can be directly determined and used.

Table 8 shows the translation of select and collect operators on sequential collections. *selectMaximals* and *selectMinimals* are not currently mapped to C.

Table 8. Mapping of selection and collection expressions

<i>OCL expression e</i>	<i>C translation e'</i>
<code>s->select(P)</code>	<code>selectE(s', fP)</code> where E is entity element type of s, fP evaluates P': <code>unsigned char fP(struct E* self) { return P'; }</code>
<code>s->select(x P)</code>	as above, fP is: <code>unsigned char fP(struct E* x) { return P'; }</code>
<code>s->collect(e)</code>	<code>collectE_et(s', fe)</code> e of primitive type et fe evaluates e'
<code>s->collect(x e)</code>	(et'*) <code>collectE(s', fe)</code>
Non-primitive type et	as above

Unlike the types and class diagram mappings, a recursive functional style of specification is needed for the expressions mapping (and for activities). This is because the subordinate parts of an expression are themselves expressions. For each category of expression, the mapping is decomposed into cases, for example:

```

BasicExpression::
query mapBasicExpression(ob : Set(CExpression),
                          aind : Set(CExpression),
                          pars : Sequence(CExpression)) : CExpression
pre:
ob = CExpression[objectRef.expId] &
aind = CExpression[arrayIndex.expId] &
pars = CExpression[parameters.expId]
    
```

```

post:
  (umlKind = value =>
    result = mapValueExpression(ob,aind,pars)) &
  (umlKind = variable =>
    result = mapVariableExpression(ob,aind,pars)) &
  (umlKind = attribute =>
    result = mapAttributeExpression(ob,aind,pars)) &
  (umlKind = role =>
    result = mapRoleExpression(ob,aind,pars)) &
  (umlKind = operation =>
    result = mapOperationExpression(ob,aind,pars)) &
  (umlKind = classid =>
    result = mapClassExpression(ob,aind,pars)) &
  (umlKind = function =>
    result = mapFunctionExpression(ob,aind,pars))

```

The operation precondition of *mapBasicExpression* asserts that the parameters correspond to the sub-parts of the basic expression. This approach enables inverse mappings to be systematically defined using the 1-1 correspondence of OCL and C expressions by identity.

The mapping transformation consists of 92 operations and 33 transformation rules. The expression mapping is then further used by the mappings of UML activities and use cases to C code. The efficiency of the expression translator has been tested on a range of UML/OCL models, and found to be of practical efficiency. For models with 100 classes, each with 100 attributes, the code generation took 1.7 s.

The semantic correctness of the mapping was checked by reasoning inductively on expression structure that $Sem_C(e')$ is equivalent to $Sem_{OCL}(e)$ for OCL expressions e , if $e' = CExpression[e.expId]$, where Sem_C is a mathematical semantics for C programs, and Sem_{OCL} is the UML-RSDS semantics for expressions, defined by a mapping from OCL to the B AMN formalism [6]. We assume that *malloc* and *calloc* always succeed, and that equivalent numeric types are used in the specification and implementation.

4 Evaluation

In this section we evaluate the effectiveness of the translation approach. The Visual Studio (2012) and lcc² (2016) C compilers were used to evaluate the generated C code. All tests were carried out on a standard Windows 7 laptop with Intel i3 2.53 GHz processor using 25% of processing capacity.

In order to test the efficiency and compactness of generated code, we considered different UML specifications with different computational characteristics. The first was a small-scale application involving a fixed-point computation of the maximum-value node in a graph of nodes. This application has one entity type A , with an attribute $x : int$ and a self-association $neighbours : A \rightarrow Sequence(A)$. There is a use case *maxnode* with the postcondition

```

A::
  n : neighbours & n.x > x@pre => x = n.x

```

² www.cs.virginia.edu/~lcc-win32.

This updates a node to have the maximum x value of its neighbours. Because this constraint reads and writes $A :: x$, a fixed-point design is generated by the UML-RSDS tools. It is an example of object-oriented specification with intensive use of navigation from object to object.

The generated C code of the use case and its auxiliary functions is:

```
void maxnode1(struct A* self, struct A* n)
{ setA_x(self, getA_x(n)); }

unsigned char maxnode1test(struct A* self, struct A* n)
{ if (getA_x(n) > getA_x(self))
  { return TRUE; }
  return FALSE;
}

unsigned char maxnode1search(void)
{ int ind_boundedloopstatement_80 = 0;
  int size_boundedloopstatement_80 = oclSize((void**) a_instances);
  for ( ; ind_boundedloopstatement_80 < size_boundedloopstatement_80;
        ind_boundedloopstatement_80++)
  { struct A* ax = (a_instances)[ind_boundedloopstatement_80];
    int ind_boundedloopstatement_85 = 0;
    int size_boundedloopstatement_85 = oclSize((void**) getA_neighbours(ax));

    for ( ; ind_boundedloopstatement_85 < size_boundedloopstatement_85;
          ind_boundedloopstatement_85++)
    { struct A* n = (getA_neighbours(ax))[ind_boundedloopstatement_85];
      if (maxnode1test((struct A*) ax, n))
      { maxnode1((struct A*) ax, n);
        return TRUE;
      }
    }
  }
  return FALSE;
}

void maxnode(void)
{ unsigned char maxnode1_running = TRUE;
  while (maxnode1_running)
  { maxnode1_running = maxnode1search(); }
}
```

Table 9 compares the code size (for the complete applications, including OCL library code) and the efficiency of the C code with the Java code produced by the UML-RSDS Java code generator. These show that code size is halved by using C, and that efficiency is improved.

Table 9. Generated C code versus Java code, case 1

	<i>C version</i>	<i>Java version</i>
<i>Code size</i>	17Kb	35Kb
<i>Execution time</i>		
A.size = 20	0	30ms
A.size = 50	15ms	70ms
A.size = 100	240ms	330ms
A.size = 200	1750ms	2500ms

In a second case, the efficiency test from [5] was used. This computes prime numbers in a given range using a double iteration. Table 10 compares the generated code in Java, C, C# and C++ on this case. In this purely numerical example, C is significantly more efficient than the alternative implementations for larger cases.

Table 10. Generated C code versus Java, C#, C++ code, case 2

Testing primes up to	<i>C version</i>	<i>Java version</i>	<i>C# version</i>	<i>C++ version</i>
10000	5ms	7ms	3ms	8ms
20000	9ms	15ms	8ms	16ms
50000	22ms	47ms	27ms	31ms
100000	47ms	63ms	54ms	62ms
200000	109ms	125ms	274ms	112ms
500000	143ms	374ms	472ms	405ms

The main causes of inefficiency in generated C code are (i) repeated linear traversals of collections to calculate the sizes of collections; (ii) the cost of allocating and reallocating large contiguous blocks of memory for array-based collections. An alternative array collection representation could use the first element of an array to store the collection length. This also has the advantage that C and OCL indexing of collections would coincide. However it would hinder the compatibility of the generated code with standard C code. For sets and bags non-contiguous memory blocks can be used, and this reduces the memory allocation costs.

We also compared the C and Java implementations using the OCL benchmarks of [1]. Table 11 shows the execution time for adding n elements to a collection, using $\rightarrow including$.

Table 12 shows the execution time for testing the membership of 2000 elements in a collection of size n , using $\rightarrow includes$.

There are the following restrictions on the UML-RSDS input specification for UML2C: (i) no overloading of operation names within a class; (ii) quantifiers, collect, select/reject predicates can only depend on one context object; (iii) no static attributes; (iv) collection values and types can only be nested to 2 levels (collections of collections of non-collection types); (v) root classes must contain at least one property, and only single inheritance is represented; (vi) there are no interfaces, association classes or qualified associations.

Restriction (iii) will be removed in release 1.8 of UML-RSDS.

Table 11. C and Java efficiency results for \rightarrow including

n	<i>Sequence</i>	<i>OrderedSet</i>	<i>Bag</i>	<i>Set</i>	<i>Java Sequence</i>	<i>Java OrderedSet</i>
4000	65ms	150ms	16ms	26ms	4ms	180ms
8000	220ms	486ms	47ms	49ms	5ms	720ms
16000	660ms	895ms	99ms	101ms	10ms	2.5s
32000	2.1s	8.9s	202ms	231ms	10ms	10s

Table 12. C/Java efficiency results for \rightarrow includes

n	<i>Sequence/OrderedSet</i>	<i>Bag/Set</i>	<i>Java Sequence/OrderedSet</i>
1000	8ms	5ms	46ms
2000	21ms	10ms	62ms
4000	46ms	11ms	140ms
8000	67ms	12ms	312ms
16000	109ms	12ms	710ms
32000	169ms	16ms	1.4s

5 Related Work

Code generation from UML to ANSI C is an unusual topic, with only one recent publication describing such a translator [3]. This code generator is described in a high-level manner, and it is not clear how OCL expressions or UML activities are mapped to C using the transformation. In contrast, we have implemented mappings for all elements of a substantial subset of UML, including a large subset of OCL. Formal specification approaches for MT are described in [2, 9]. The constructive logic approach of [9] does not appear to have been applied to large scale transformations. The approach of [2] is focussed on the specification of architectural choices. Our approach enables large-scale transformations to be specified using OCL, with their implementations being verified as correct-by-construction.

A Java VM is the usual target for OCL execution [10]. Compared to [10] we consider a subset of OCL which (i) omits OclAny, null and invalid values, (ii) uses classical logic, (iii) uses computational numeric types. These modifications make the correspondence between a (UML-RSDS) OCL specification and a Java/C#/C++/C implementation more direct and also simplify specification verification, eg., using the B formal method or other classical logic theorem prover.

6 Conclusions

The UML to C translator is the largest transformation which has been developed using UML-RSDS, in terms of the number of rules (of the order of 250 OCL rules/operations in 5 subtransformations). The translator provides efficient implementation of OCL using a direct translation approach which supports traceability and bidirectionality. The translator has been incorporated into the UML-RSDS tools version 1.7 at nms.kcl.ac.uk/kevin.lano/uml2web. UML-RSDS specifications are type-checked and converted to designs prior to export for code generation. The translator is itself defined using UML class diagrams and the UML-RSDS subset of OCL, demonstrating that

purely declarative OCL specifications can be sufficient for large and complex applications: no activities or other procedural elements were needed in the specification. We found substantial benefits in reduced development time and improved correctness and flexibility compared to the manually-coded translators for Java, C# and C++. The UML2C OCL code is less than 25% of the size of the Java code of the manually-coded C++ translator, and required half the development effort.

References

1. Cuadrado, J., Jouault, F., Molina, J., Bezivin, J.: Deriving OCL optimisation patterns from benchmarks. In: OCL 2008 (2008)
2. Dieumegard, A., Toon, A., Pantel, M.: Model-based formal specification of a DSL library for a qualified code generator. In: OCL 2012 (2012)
3. Funk, M., Nysen, A., Lichter, H.: From UML to ANSI-C: an Eclipse-based code generation framework. RWTH (2007)
4. Kernighan, B., Ritchie, D.: The C Programming Language. Prentice Hall, Upper Saddle River (1988)
5. Kuhlmann, M., Hamann, L., Gogolla, M., Buttner, F.: A benchmark for OCL engine accuracy, determinateness and efficiency. *SoSyM* **11**, 165–182 (2012)
6. Lano, K.: Agile Model-Based Development Using UML-RSDS. Taylor and Francis, Milton Park (2016)
7. OMG: OCL Version 2.4 (2014)
8. OMG: Semantics of a Foundational Subset for Executable UML Models (FUML), v1.1 (2015)
9. Zschaler, S., Poernomo, I., Terrell, J.: Towards using constructive type theory for verifiable modular transformations. In: FREECO 2011 (2011)
10. Willink, E.: An extensible OCL virtual machine and code generator. In: OCL 2012 (2012)

Mapping USE Specifications into Spec#

Jagadeeswaran Thangaraj^(✉) and SenthilKumaran Ulaganathan

School of Information Technology and Engineering,
VIT University, Vellore, TN, India
jagadeest@gmail.com

Abstract. The UML model is easy to describe the object oriented program components clearly in graphical notation. OCL allows users to express textual constraints about the UML model. The USE tool allows specification to be expressed in a textual format for all features of the UML model with OCL constraints. Spec# is a formal language, which extends C# with constructs for non-null types, preconditions, post conditions, and object invariants. It allows programmers to document their design decisions in the code. Spec# has run time verifier to verify the specification constraints over the C# code. This paper describes the mapping of USE specifications into Spec# which helps to improve the quality of both UML/OCL and Spec#.

Keywords: USE · UML · OCL · Spec#

1 Background and Motivation

The Unified Modelling Language (UML) model is easy to describe the object oriented program components clearly at the system design stage. The UML's class diagram depicts the details of a class of the model in an object oriented system [9]. The relationship restrictions with other classes can be described by associations which are called UML constraints. Association multiplicities define the connection relation of classes to each other. Object Constraint Language (OCL) allows users to express textual constraints about the UML model [8]. So the UML class diagram with OCL constraints can describe all the elements of object program constructs with their specification.

The UML-based Specification Environment (USE) tool describes the program's specification at the specification level. The USE tool is based on a subset of UML and OCL. The USE tool allows specification to be expressed in a textual format for all features of a model, e.g., classes, attributes in the UML class diagrams. Additional constraints are written using OCL expressions [6]. The USE specification can easily convert to corresponding graphical representations using textual editor: Class diagram, Object diagram. Also it performs the verification of OCL constraint structures easily.

Spec# has run time verifier to verify the specification constraints over the C# code. Spec#'s specifications are not just comments, but those are executable [10].

In recent years, model based transformation is getting more popular [2] i.e. code generation from system design. At the moment, there is no explicit tool to generate Spec# code from UML/OCL.

In this paper, we map the UML/OCL properties of USE specification in order to generate Spec# code. Motivation behind this mapping is to find out what properties can add at the design phase in order to improve the quality of UML/OCL. In the same manner, this paper helps improving Spec# to support full UML/OCL properties. Remainder of the paper is organised as follows. Next section maps the properties between USE specification and Spec#. Class specification mapping is illustrated in Sect. 2.1 and constraints in Sect. 2.2. Unmapped properties are explained in Sect. 2.3. Finally Sect. 3 explains the conclusion of the mapping and recommended future works.

2 Mapping UML/OCL Properties Between USE and Spec#

For code generation, we need the corresponding references to the elements of both the source and the target languages [3]. This section presents the structural correspondence of USE (UML/OCL) and Spec#.

2.1 Mapping Class Diagrams to Spec# Classes

This section explains the mapping of the class properties between the USE specification and Spec#.

Primitive Types: Integer, Real, Boolean and String [8] are primitive types in USE. The USE primitive types are directly mapped on predefined Spec# types. Thus the primitive USE types *Boolean*, *Integer*, *Real* and *String* are, respectively, mapped into *Boolean*, *Integer*, *Float* and *String* of Spec#.

Collection Types: Collection types are used to group the elements together in some formal manner: *Set*, *Bag*, *OrderedSet* and *Sequence* [8]. The Spec# generic class, 'System.Collections.Generic.List' stores the elements in the format of *Sequence*. The OCL constraints are constructed on a UML diagram using these collection types, but, Spec# only supports the collection type *List*. The collection operations for all USE types are mapped into the corresponding operations of Spec#'s list.

Class and Enumeration: In an USE specification, class diagrams define the static characteristics of the system by specifying all classes, attributes and methods of each class and interrelations between the objects of these classes. In Spec#, whole program implementations mirror the role of the class diagram. The class construct is used to define all aspects of the class model with attributes, method

definitions, inheritance and association relations. In USE, operations of a class are represented after attribute declaration using the keyword *operations* as an example shown in Table 1. In Spec#, operations are written as standard method definitions.

Enumeration is used to hold the predefined constants to declared variables. The UML supports enumeration types using keyword *enum* [8]. The Spec# also supports the *enum* keyword as shown in Table 1.

Associations and Aggregations: An *association* describes the static relationship between the classes. In USE, associations are represented using the keyword *association* followed by the association name representing the link between the classes with role names. In Spec#, the associations are represented by constructing the objects of the association in the class definition. An association with multiplicity ‘1’ is represented as a single object, and the association with multiplicity ‘*’ is represented as a list of object declarations. References to other objects are represented with the ownership type annotations ([Rep] & [Peer]).

Table 1. Class and enumeration representation

<i>USE</i>	<i>Spec#</i>
<pre>enum Color{silver, gold} class Customer attributes name : String; title : String; isMale : Boolean; age : Integer; operations age():Integer; birthdayHappens(); end class CustomerCard end association holds between Customer[1] role owner CustomerCard [0..*] role cards end</pre>	<pre>public enum Color{silver, gold}; public class Customer { [Rep][ElementsRep] List<CustomerCard> cards = new List<CustomerCard>(); protected String name; protected String title; protected bool isMale; protected int age; public int age() { ---- } public void birthdayHappens() { ---- } } public class CustomerCard { [Rep]protected Customer owner; ... }</pre>

Any object can refer to other objects. Aliasing occurs when one object is reachable through multiple paths, i.e. more than one reference is referred by the same object. Ownership helps to control aliasing and assists in structuring object relationships in a program. By using this ownership representation, an owner object can access the reference objects. Ownership types help the programmer track information about object aliasing. Ownership types representation mainly specified in two types: Rep & Peer. Same ownership objects are represented ‘peers’ or ‘siblings’ [1]. Some objects are represented as reference of an owner object, are called ‘reference’ objects, i.e. an object can referred by owner. Sometimes multiple references can exist to an object. A [Rep] attribute which stands for representation [5]. [ElementsRep] specifies the ‘*’ multiplicity as list of objects. An example is shown in Table 1.

Inheritance: Inheritance is an important concept in object-oriented design, which allows identical functionality of a class to be inherited into another class. New functionality can then be added to the class which inherits [7]. For example, **Burning** and **Earning** are subclasses of the **Transaction** class. Subclass is that may inherit the properties of superclass. In USE, inheritance is represented by the ‘<’ operator.

In Spec#, inheritance is represented by the ‘:’ operator. Subclasses attributes which need to access superclass attributes must be declared with [Additive] keyword [5]. If an object of a subclass needs to access attributes of its superclass, then those attributes must be annotated with the keyword [Additive]. In the example shown in Table 2, an attribute *points* is overridden in the `earnPoints()` method of the subclass **Earning**. Therefore, it needs to access the superclass **Transaction**’s attribute *points*. Therefore it is annotated as [Additive].

Table 2. Inheritance representation

<i>UML</i>	<i>Spec#</i>
<pre>class Transaction attributes points : Integer; operations earnPoints(points); end class Burning < Transaction end class Earning < Transaction earnPoints(points); end</pre>	<pre>public class Transaction { [Additive] public int points; [Additive] public void earnPoints(int points) { additive expose (this){ } } } public class Burning:Transaction { } public class Earning:Transaction { [Additive] public void earnPoints(int points) { additive expose (this) { } } }</pre>

USE supports multiple inheritance by comma as an example follows:

```
Earning < Transaction, Burning
```

Here, **Earning** class inherits from classes **Transaction** and **Burning**. As C#, Spec# does not support multiple inheritance.

2.2 Mapping OCL Constraints to Spec#

Constraints are conditions or restrictions over a model or state. In USE, constraints are specified by Boolean expressions which must be side effect free. That means, the constraint must be evaluated to true or false and it must not change the state over the system. In a correct system, constraints must be evaluated to true. In USE, constraints are defined over various elements of the class diagram. *Invariants*, *preconditions* and *postconditions* are major constraints [8] which are specified by the operators *inv*, *pre* and *post*. These are checked via a validation process. These constraints are represented in Spec# as assertions. This section explains the mapping of these properties between the USE specification and Spec#.

Preconditions: A precondition is a condition that must be true before calling a method in a context in order to get the expected behaviour from the method. In Design by Contract (DBC), the method's client must meet the precondition. In a university, a student must be older than 23 years to enroll into a course as a mature applicant. This is described as a constraint as follows in Table 3.

Table 3. Precondition representation

<i>OCL</i>	<i>Spec#</i>
<pre>context MatureProgram ::enroll(stu : Student) pre: stu.age >23</pre>	<pre>public class MatureProgram { public void enroll(Student stu) requires stu.age >23; { } }</pre>

The precondition declares that for any Student **stu**, who will be enrolled into a course as a mature student, his age must be greater than 23. The keyword **requires** is used to represent preconditions in Spec#.

Postconditions: A postcondition is a condition that should be true after executing a method in a context if the method behaves as expected when executed with a true precondition. In DBC, a designer establishes the postcondition. For example, as shown in Table 4, the method postcondition declares that the result of the method `enroll()` must add the student `stu` to the `MatureProgram` if he/she has already not enrolled into the program. The keyword `ensures` is used to represent the postcondition in `Spec#`.

Table 4. Postcondition representation

<i>OCL</i>	<i>Spec#</i>
<pre>context MatureProgram ::enroll(stu : Student) post: numStudents = numStudents@pre + 1 context MatureProgram ::allocate() post: self.numPlaceAvail = self.numPlaceAvail@pre - 1</pre>	<pre>public class MatureProgram { public void enroll(Student stu) ensures numStudents.Count == old(numStudents.Count) +1 { } public void allocate() ensures this.numPlaceAvail == old(this.numPlaceAvail)-1; }</pre>

The Special Property '@pre': In USE, `@pre` is used to hold previous value of an element before methods execution. The keyword `old` performs the same function in `Spec#`. An example follows in Table 4. In this, if a student enrolled into a course, the number of available places must be reduced by one.

Keyword 'self': In USE, the keyword `self` is used to refer the current instance of certain object of a class. The keyword `this` is used for the same function in `Spec#`.

Class Invariants: A class invariant is a condition that should be true during the entire life cycle of the class instances. That means, the class invariants must be hold for entire life of objects created. For example, a class invariant could be that a student must be more than 18 years old to enter into 3rd level education as shown in Table 5. The keyword `invariant` is used to represent the invariants in `Spec#` as shown in Table 5. During inheritance in `Spec#`, an overriding method may add additional postconditions with the superclass's preconditions and postconditions but cannot add new preconditions in order to keep the property of strengthening postconditions and weakening preconditions. A subclass may also strengthen the invariant.

Table 5. Class invariant representation

<i>OCL</i>	<i>Spec#</i>
<pre>context Student inv : age > 18</pre>	<pre>Class Student { invariant age>18; }</pre>

Table 6. Correspondence of UML/OCL properties between USE and Spec# ✓: Support but no specific keyword and ✗: No support

UML/OCL properties	USE	Spec#
Precondition	pre	requires
Postcondition	post	ensures
Invariant	inv	invariant
Attributes	attributes	✓
Collection	Set, Bag, Sequence	List
Old	@pre	old
Quantifiers	✓	forall, exists
Multiple inheritance	✓	✗
OCL types	✓	✗
Metatypes	✓	✗
Initial	✗	constructor
Derived	✗	✓
Non null	✗	✓
Termination	✗	✗

2.3 Unmapped Properties

Collection Types: Table 6 shows the correspondence of UML/OCL properties between USE and Spec#. Based on this comparison, Spec# needs to define the generic collection types such as Set, Bag, Sequence. Also it needs to define corresponding ‘Collection’ Operations.

Special OCL Types: Spec# does not support OCL Meta types such as OclAny, OclType, OclUndefined, OclVoid and OclInvalid.

Special OCL Operations: Spec# does not have separate keywords for OCL operations such as `init`, `derive`, `define` and `body`. These operations support by method definitions in Spec#.

Ownership Types: On the other hand, Spec# provides ownership type constraints (`Rep`, `Peer`) in association relations and Inheritance properties from one class to another to specify conditions using [Additive].

Non-null Reference: Spec# has special feature, Non null-reference, that eradicates all non null dereference errors. That is, references in Spec# can be declared Null or Non-null. In Spec#, type `T!` contains only references to objects of type `T`, which cannot be null.

For example, `'string! S;'` specifies Non-null string. `'string? S;'` specifies the String which may be a null reference. In `'string! []? a;'`, `'a'` is either null or a string array and all elements in the array are non-null.

3 Conclusion

This paper has presented the mapping of USE properties with Spec# for generating the Spec# code skeletons. It gives an idea to introduce some properties in software design and implementation towards to support the verification. Based on our study, USE does not allow the addition of ownership type constraints (`Rep`, `Peer`) in software design phase. We have introduced these ownership type information to UML/OCL [4]. In this paper, we developed an approach to introduce ownership type constraints to USE specifications.

3.1 Future Work

To support OCL directly, Spec# needs the collection operations. So our next aim is to generate a library to support the generic collection data types (`Set`, `Bag` and `Sequence`) and the different operations on the collection types (`size`, `isEmpty`, `notEmpty`, `sum`, `count`, `includes` and `includesAll`). Also our work will support Meta types like `OclAny`, `OclType` and OCL statements (`OCLKindof`, `OCLTypeof`).

References

1. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: a survey. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. LNCS, vol. 7850, pp. 15–58. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36946-9_3. ISBN 978-3-642-36946-9

2. Hilken, F., Niemann, P., Gogolla, M., Wille, R.: From UML/OCL to base models: transformation concepts for generic validation and verification. In: Kolovos, D., Wimmer, M. (eds.) ICMT 2015. LNCS, vol. 9152, pp. 149–165. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21155-8_12
3. Shimba, H., Hanada, K., Okano, K., Kusumoto, S.: Bidirectional translation between OCL and JML for round-trip engineering. In: 20th Asia-Pacific Software Engineering Conference (APSEC 2013), pp. 49–54. IEEE (2013)
4. Thangaraj, J., SenthilKumaran, U.: Introducing ownership type constraints to UML/OCL. In: International Workshop on Aliasing, Capabilities and Ownership, IWACO17 co located with the 31st European Conference on Object-Oriented Programming, ECOOP 2017, Barcelona, Spain, June 2017
5. Leino, K.R.M., Müller, P.: Using the Spec# language, methodology, and tools to write bug-free programs. In: Müller, P. (ed.) LASER 2007-2008. LNCS, vol. 6029, pp. 91–139. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13010-6_4
6. Gogolla, M., Buttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.* **69**, 27–34 (2007). Elsevier
7. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3. ISBN 978-3-540-30569-9
8. OMG: Object Constraint Language (OCL): Version 2.3.1., Object Management Group (2012). <http://www.omg.org/spec/OCL/2.3.1>
9. OMG: Unified Modeling Language (UML): Version 2.4.1., Object Management Group (2011). <http://www.omg.org/spec/UML/2.4.1>
10. Monahan, R., Leino, K.R.M.: Program verification using the Spec# programming system. In: ECOOP Tutorial (2009)

Deterministic Lazy Mutable OCL Collections

Edward D. Willink^(✉)

Willink Transformations Ltd., Reading, England
ed@willink.me.uk

Abstract. The Collection iterations and operations are perhaps the most important part of OCL. It is therefore important for an OCL evaluation tool to provide efficient support for Collections. Unfortunately, some clauses of the OCL specification appear to inhibit efficient or deterministic support. We review the inhibitions and demonstrate a new deterministic and lazy implementation that avoids them.

Keywords: OCL · Collection · Deterministic · Lazy · Mutable

1 Introduction

The OCL specification [11] defines an executable specification language suitable for use with models. OCL's power comes from its ability to evaluate characteristics of multiple model elements using iterations and operations over collections.

The side-effect free functional characteristics of OCL should provide excellent opportunities for optimized evaluation, but sadly the optimization in typical OCL tools is poor. Collection evaluation is an area that should be particularly good, however it is very easy for the efficiency and/or memory usage to be outstandingly bad.

Deterministic execution is a desirable property of any language; very desirable if you are attempting to debug an obscure failure. Unfortunately today's OCL tools are not deterministic and so OCL-based Model-to-Model transformations tools also lack determinism.

In Sect. 2, we review the problems that the OCL specification appears to pose. In Sect. 3 we revisit these problems to identify over-enthusiastic or inappropriate reading of the OCL specification. Then in Sect. 4 we introduce our new Collection implementation that solves the problems. The new solution is still work in progress and so in Sect. 5 we describe what remains to do to integrate it effectively. In Sect. 6 we look at related work and conclude in Sect. 7.

2 The Problems

We briefly review some implementation challenges that the OCL specification provides.

Collection types: Four concrete derivations of the abstract *Collection* type are specified to support the four permutations of ordered/not-ordered, unique/not-unique content. These derived collections types are

- *Bag* - not-ordered, not-unique
- *OrderedSet* - ordered, unique
- *Sequence* - ordered, not-unique
- *Set* - not-ordered, unique

Java implementations may use a custom class, `LinkedHashSet`, `ArrayList` and `HashSet` respectively to implement these four `Collection` kinds.

Problem: Four distinct collection types.

Immutability: OCL is a functional language free from side effects. It is therefore impossible to modify an OCL *Collection*. There are no operations such as `Set::add(element)` that modify the receiver. Rather there are operations such as `Set::including(element)` that return a new `Set` based on the receiver and including the additional `element`. The obvious implementation of a cascade of operations such as `a->including(b)->including(c)->including(d)` creates a new intermediate collection between each operation.

Problem: Immutability implies inefficient collection churning.

Eagerness: OCL operations are defined as a computation of an output from some inputs. A cascade of operations such as `a->including(b)->excludes(c)` is therefore evaluated in three steps as `get-a`, then `create a+b`, and finally test `a+b` for `c` content. There is no mechanism for early discovery of a `c` to bypass redundant computations.

Problem: Specification implies eager evaluation.

Invalidity: A malfunctioning OCL evaluation does not throw an exception, rather it returns the `invalid` value, which will normally be propagated through invoking computations back to the caller. However OCL has a strict Boolean algebra that allows the `invalid` value to be ‘caught’ when, for instance, ANDed with the `false` value. The presence of the `invalid` value in a collection is prohibited, or rather the whole collection that ‘contains’ the `invalid` value is replaced by the `invalid` value. The result of a collection evaluation cannot therefore be determined until every element is present and checked for validity.

Problem: Invalidity implies full evaluation.

Determinism: Each collection type has distinct useful capabilities and so conversions between collection types are specified to facilitate their use. However, when the `asOrderedSet()` and `asSequence()` operations are applied to not-ordered collections, the operations must create an ordering without any clue as to what a sensible ordering criterion might be. This is obviously impossible and so typical Java implementations use the indeterminate order provided by a Java iteration over an underlying Java `Set`.

Problem: `asOrderedSet()` and `asSequence()` imply indeterminacy.

OCL equality: OCL is a specification language and when dealing with numbers, OCL uses unbounded numbers. Consequently the following OCL expressions are true:

```
1 = 1.0 Set{1,1.0}->size() = 1 Set{Set{1},Set{1.0}}->size() = 1
```

When using Java to implement OCL, the numeric equality is satisfied by the primitive types `int` and `double` but not by the object types `Integer` and `Double`. Since Java sets use object equality to establish uniqueness, a naive implementation may malfunction if it assumes that OCL and Java equality are the same.

Problem: OCL and Java equality semantics are different.

3 The Problems Revisited

The foregoing problems lead to poor and even inaccurate OCL implementations. We will therefore examine them in more detail to distinguish myth and truth before we introduce our new solution.

3.1 Immutability

While OCL may provide no operations to modify Collections, it does not prohibit modification by underlying tooling. A modification that does not affect OCL execution is permissible.

An evaluation of `a->including(b)->including(c)` may therefore re-use the intermediate collection created by `a->including(b)` and modify it to create the final result. This is safe since the intermediate result cannot be accessed in any other way than by the subsequent `->including(c)`. If there are no other accesses to `a`, it is permissible to modify `a` twice and avoid all intermediates.

3.2 Eagerness

While the specification may imply that evaluations should be performed eagerly, this is just the way specifications are written to ease understanding. An implementation is permitted to do something different so long as the difference is not observable. Lazy evaluation is a tactic that has been used with many languages. OCL has a strong functional discipline and so laziness has much to offer in an OCL evaluator. Unfortunately OCL development teams have been slow to exploit this tactic.

3.3 Invalidity

The OCL specification is far from perfect. In OCL 2.0, there were the three overlapping concepts of *null*, *undefined* and *invalid*. OCL 2.2 clarified the concepts by eliminating *undefined* and so distinguished *null* and *invalid*, but *invalid* is still inadequate to represent real execution phenomenon.

There is currently no distinction between program failures such as

- divide by zero
- *Sequence/OrderedSet* index out of range
- *null* navigation

and machine failures such as

- stack overflow
- network failure

Since machine failures are not mentioned by the specification, it would seem that they must be *invalid*, but only very specialized applications such as the OCL specification of a debugger can be expected to handle machine failures. Consequently the treatment of machine failures as *invalid* for the purposes of 4-valued (*true*, *false*, *null*, *invalid*) strict logic evaluation seems misguided. Rather a further fifth *failure* value for machine failure should be non-strict so that machine failures are not catchable by logic guards. The fourth strict *invalid* value should apply only to program failures.

Program failures are amenable to program analysis that can prove that no program failure will occur. When analysis is insufficiently powerful, the programmer can add a redundant guard to handle e.g. an ‘impossible’ divide-by-zero. With 5-valued logic we can prove that the partial result of a collection evaluation will remain valid if fully evaluated and so avoid the redundant full calculation when the partial calculation is sufficient.

Proving that null navigations do not occur is harder but an analysis of null safety is necessary anyway to avoid run-time surprises [5].

Once machine failures are irrelevant and the absence of program failures has been proved, a partial collection result may be sufficient; the redundant evaluations can be omitted.

3.4 Determinism

Determinism is a very desirable characteristic of any program evaluation, particularly a specification program. Is OCL really non-deterministic?

`Collection::asSequence()` is defined as returning elements in a collection kind-specific order.

The `Set::asSequence()` override refines the order to *unknown*, which is not the same as *indeterminate*.

The `Collection::any()` iteration specifies an *indeterminate* choice between alternatives.

The foregoing appears in the normative part of the specification. Only the non-normative annex mentions a lack of determinism for order discovery.

It is therefore unclear from the specification text whether an OCL implementation of order discovery may be non-deterministic. A clarified OCL specification could reasonably take either alternative. If order discovery is deterministic, it is easy for `Collection::any()`’s choice to be consistent with that discovery.

In practice, typical OCL implementations use a Java `Set` to realize OCL *Set* functionality. The iteration order over a Java `Set` depends on hash codes, which depend on memory addresses, which depend on the unpredictable timing of garbage collection activities. It is therefore not possible for typical OCL implementations to be deterministic.

It would appear that implementation pragmatics are driving the specification or at least the user perception of the specification. But indeterminacy is so bad that it would be good to find a way to make OCL deterministic.

3.5 Four Collection Types

The four permutations of unique and ordered provide four collection behaviors and four specification types, but do we really need four implementation types? With four types we may have the wrong one and so we need conversions. UML [10] has no collection types at all. What if an implementation realized all four behaviors with just one implementation type? One benefit is obvious; no redundant conversions.

4 New Collection Solution

Our new solution has only one *Collection* implementation type that exhibits all four *Collection* behaviors, but only one at a time. To avoid confusion between our new *Collection* implementation and the OCL abstract *Collection* or the Java *Collection* classes, we will use `NewCollection` in this paper¹.

4.1 Deterministic Collection Representation

A `NewCollection<T>` instance uses two Java collection instances internally.

- `ArrayList<T>` of ordered elements.
- `HashMap<T, Integer>` of unique elements and their repeat counts.

For a *Sequence*, the `ArrayList` serializes the required elements; the `HashMap` is unused and may be `null`.

For a *Set*, the keys of the `HashMap` provide the unique elements each mapped to a unit `Integer` repeat count; the `ArrayList` serializes the unique elements in a deterministic order.

For an *OrderedSet*, the keys of the `HashMap` provide the unique elements each mapped to a unit `Integer` repeat count; the `ArrayList` serializes the unique elements in the required order.

For a *Bag*, the keys of the `HashMap` provide the unique elements each mapped to a repeat count of that element; the `ArrayList` serializes the unique elements in a deterministic order.

¹ The Eclipse OCL class name is currently `LazyCollectionValueImpl`.

The Java implementation of a `HashSet` uses a `HashMap` and so using a `HashMap` for `Set` and `OrderedSet` incurs no additional costs. On a 64 bit machine, each `HashMap` element incurs a 44 byte cost per `Node` and typically two 8 byte costs for pointers. Using an `ArrayList` as well as a `HashMap` increases the cost per entry from 60 to 68 bytes; a 13% overhead for non-*Sequences*.

Use of an `ArrayList` to sequence the unique elements allows an efficient deterministic iterator to be provided for all kinds of *Collection*.

Since a *Set* now has a deterministic order, there is no implementation difference between a *Set* and an *OrderedSet*.

The deterministic order maintained by the `ArrayList` is based on insertion order. New elements are therefore added at the end or not at all, which avoids significant costs for `ArrayList` maintenance.

For a *Bag*, there is a choice as to whether an element iteration is over all elements, repeating repeated elements, or just the unique elements. The `NewCollection` therefore provides a regular `iterator()` over each element, and an alternative API that skips repeats but allows the repeat count to be accessed by the iterator. *Bag*-aware implementations of *Collection* operations can therefore offer a useful speed-up.

The `NewCollection` supports all *Collection* behaviors, but only one at a time. Non-destructive conversion between behaviors can be performed as no-operations. A *Set* converts to a *Sequence* by continuing to use the `ArrayList` and ignoring the `HashMap`. However the conversion from a *Sequence* to a *Bag* or *Set* requires the `HashMap` to be created and non-unique content of the `ArrayList` to be pruned; a new `NewCollection` is therefore created to avoid modifying the original `NewCollection`.

The `NewCollection` does not inherit inappropriate Java behavior. The problems with inconsistent OCL/Java equality semantics can therefore be resolved as `NewCollection` delegates to the internal `HashMap`.

4.2 Performance Graphs

The performances reported in the following figures use log-log axes to demonstrate the relative linear/quadratic execution time behaviors over a 6 decade range of collection sizes. The measurements come from manually coded test harnesses that instrument calls to the specific support routines of interest. Considerable care is taken to ensure that the 64 bit default Oracle Java 8 VM has warmed up and is garbage free. Curves are ‘plotted’ backwards i.e. largest collection size first to further reduce warm up distortions. Each plotted point comes from a single measurement without any averaging. Consequently the occasional ‘rogue’ point is probably due to an unwanted concurrent activity and demonstrates the probable accuracy of surrounding points even at the sub-millisecond level. Genuine deviations from smooth monotonic behavior may arise from fortuitous uses of L1 and L2 caches. Garbage collection may lead to inconsistent results for huge collection sizes.

4.3 Deterministic Collection Cost

Figure 1 shows the time to create a *Set* from a *Sequence* of distinct integers, contrasting the ‘old’ Eclipse OCL *Set* with the ‘new’ *NewCollection Set*. Overall the ‘new’ design is about 2 times slower corresponding to the use of two rather than one underlying Java collection.

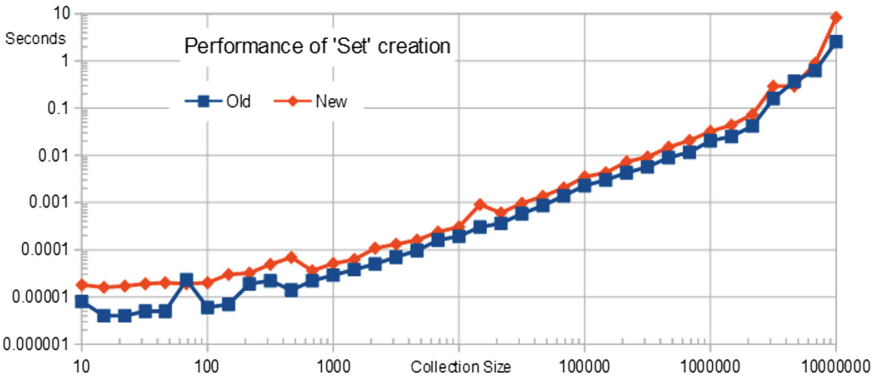


Fig. 1. ‘Set’ creation performance

A corresponding contrast of iteration speed is shown in Fig. 2. The ‘new’ design is now about three times faster since the iteration just traverses adjacent entries in the deterministic *ArrayList* rather than the sparse tree hierarchy of non-deterministic *HashMap* nodes.



Fig. 2. ‘Set’ iteration performance

Iteration is faster than creation and so it depends how often the *Set* is used as to whether ‘new’ or ‘old’ is faster overall. More than three uses and the

‘new’ design is faster as well as deterministic. Even when used only once the speed penalty is less than a factor of two. Determinism is therefore practical and incurs acceptable size and speed costs.

4.4 Lazy Usage

The ‘eager’ exposition of `NewCollection`’s `ArrayList` solves the problem of indeterminacy. The lazy use of a `HashMap` as well as the `ArrayList` supports conversions and non-*Sequence* collections.

The `NewCollection` may also be used for lazy evaluation by providing careful support for Java’s `Iterator` and `Iterable` interfaces.

When a `NewCollection` has a single consumer, its `Iterator` may be used directly by invoking `iterator()` to acquire an output iterator that delegates directly to the input.

When a `NewCollection` has multiple consumers, it must be used as an `Iterable` to provide a distinct `Iterator` for each consumer. `iterable()` is invoked to activate the caching that then uses an internal iterator to iterate over the input at most once.

Considering: `a->including(b)->including(c)`

An eager implementation of `Collection::including` might be implemented by the `IncludingOperation.evaluate` method as shown in Fig. 3.

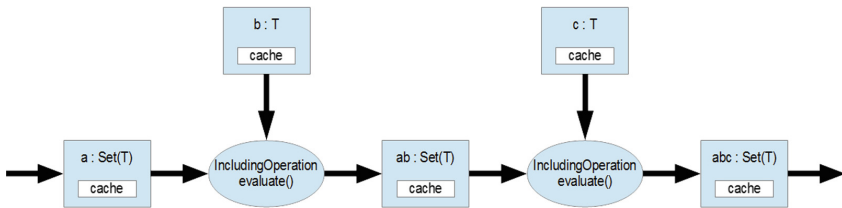


Fig. 3. Example eager evaluation data flow

The stateless `IncludingOperation::evaluate()` eagerly accesses the `a` and `b` values cached by their `Variable` objects and creates the intermediate `ab`. A second `IncludingOperation::evaluate()` similarly produces the result `abc`. Three collection caches are fully populated for each of `a`, `ab` and `abc`.

The lazy implementation shown in Fig. 4 uses an `IncludingIterator` object that has a `current` iteration context. The iterator iterates to produce the required output, one element at a time by fetching the inputs one element at a time and interleaving the additional value at the correct position. No computation is performed until an attempt is made to access the `abc` result. Since the result cache is missing, the `abc` access invokes `IncludingIterator::next()` to provide each element of `abc` that is required. `IncludingIterator::next()` provides its result from `c` or by invoking `next()` on `ab`, which in turn acquires its values from `a` or `b`. No input, intermediate or output collection caches are

required; **a** can read its source one element at a time, **ab** relays its values one at a time, and the **abc** output may be accessed one element at time. This is a major size improvement, three uncached `NewCollections` that relay one element at a time, rather than three fully-cached `NewCollections`.

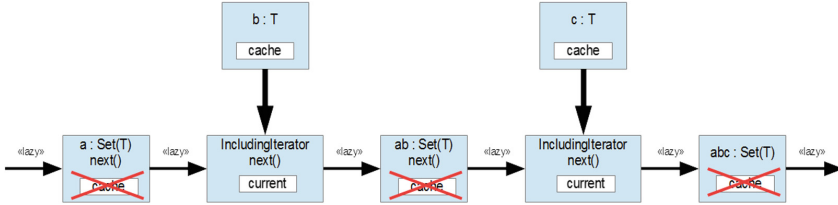


Fig. 4. Example lazy evaluation data flow

If **a** or **c** has multiple consumers, as shown in Fig. 5, the undesirable repetition of the lazy including computations is avoided by activating caches where the multi-use occurs. This is slightly awkward to implement since the first consumer must invoke `NewCollection.iterable()` to activate the cache before any consumer invokes `NewCollection.iterator()` to make use of the collection content. As part of a general purpose library used by manual programmers this programming discipline could cause many inefficiencies. However as part of an OCL tool, an OCL expression is easily analyzed to determine whether a collection variable is subject to multiple access. If analysis fails, `iterable()` can be invoked just in case.

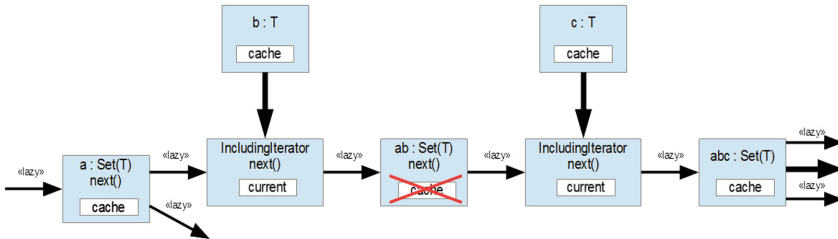


Fig. 5. Example lazy cached evaluation data flow

Eager, lazy and cached evaluations share the same structure of operation and variable interconnections. The correct behavior is determined by analysis of the OCL expression. For a singly accessed collection, a transparent behavior is configured. For multiple access, a cached behavior is configured in which the source iteration is lazily cached for multiple use by the multiple accesses. Unfortunately, collection operations, such as `Collection::size()`, are unable to return a result until the source has been fully traversed and so an eager evaluation is sometimes unavoidable.

4.5 Lazy Cost

In Fig. 6 we contrast the performance of eager and lazy OCL evaluation of the inclusion of two values into a *Sequence of Integers*.

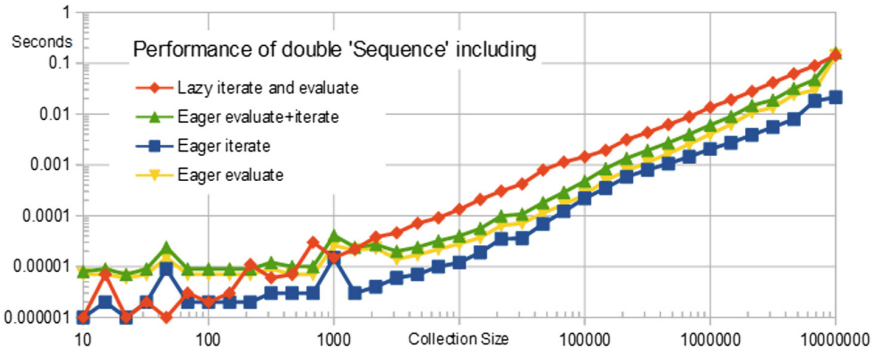


Fig. 6. Double including ‘sequence’ performance

For more than 1000 elements, the top curve shows the lazy approach scaling proportionately. The next curve shows the aggregate performance of eager evaluation also scaling proportionately until garbage collection affects results at 10,000,000 elements. The bottom two curves show the contributions to the aggregate from the eager evaluation, and the final result traversal.

For small *Sequences* with fewer than 1000 elements, the higher constant costs of the eager approach dominate and the lazy approach is perhaps five times faster.

For larger *Sequences*, the lazy approach is about two times slower since an outer element loop traverses iterations for each partial computation whereas the eager approach has tighter inner loops for each partial computation.

For the largest 10,000,000 element result, garbage collection has started to affect the eager evaluation with its three full size collection values for input, intermediate and output. In contrast, the lazy approach only uses a few hundred bytes regardless of model size and so is much less affected by huge models.

The lazy approach is clearly superior with respect to memory consumption, and also faster for up to about 1000 elements. For larger sequences, lazy evaluation may be two times slower. Since lazy evaluation offers the opportunity to skip redundant computations, we may conclude that in the absence of application-specific profiling measurements, lazy evaluation should be used.

4.6 Mutable Collections

As suggested above, lazy evaluation is not always better. The simple example in Fig. 4 replaced three fully-cached by three uncached `NewCollections` but also introduced two intervening `IncludingIterator` objects. Invocation of `next()`

to return an output object traverses the lazy sources incurring four nested invocations of `next()`. For an iteration such as

```
aCollection->iterate(e; acc : Set(String) |
    acc->including(e.name))
```

the overall iterate of an N -element `aCollection` evaluates using a chain of N interleaved `NewCollection` and `IncludingIterator` objects. The overall evaluation incurs a quadratic $2 * N * N$ cost in `next()` calls.

Of course the traditional approach of creating a new `Collection` for each invocation of `including` also incurs a quadratic cost through creating and copying N collections of approximately N -element size.

In order to achieve a more reasonable cost we can use a non-OCL mutable operation behind the scenes:

```
aCollection->iterate(e; acc : Set(String) |
    acc->mutableIncluding(e.name))
```

This exploits the invisibility of the intermediate values of `acc`. The evaluation should therefore analyze the OCL expression to detect that the single use of `acc` allows the immutable `including()` operation to be evaluated safely and more efficiently using the internal `mutableIncluding()` operation.

In Fig. 7 we contrast the performance of the accumulation that computes `S->iterate(i; acc : C(Integer) = C{} | acc->including(i))`

using `Set` or `Sequence` as the `C` collection type and `C{1..N}` as the `S` source value for an N collection size.

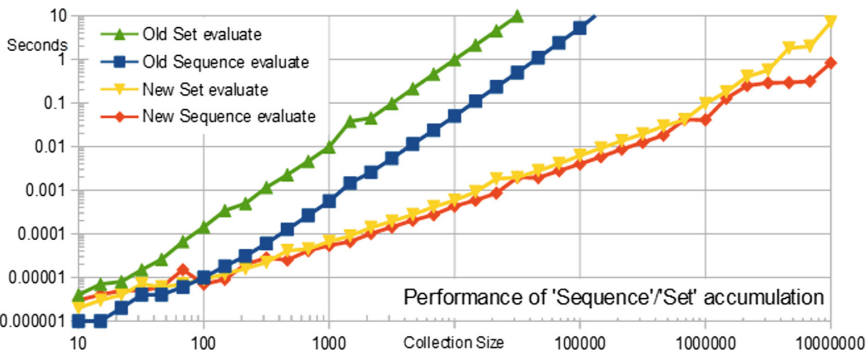


Fig. 7. ‘Sequence’ and ‘Set’ accumulation performance

The new approach uses mutable evaluation to re-use `acc` and so avoid churning. The old approach uses the one new `Set` churn per `Set::including` execution as currently practiced by Eclipse OCL [7] and USE [12] (Dresden OCL [6] creates two `Sets`). The new approach scales linearly and so is clearly superior to

the traditional quadratic cost. The new approach has a two-fold cost for using *Sets* rather than *Sequences*; much less than when churning occurs.

Note that this optimization relies on a ‘compile-time’ OCL expression analysis that replaces `including` by `mutableIncluding`.

4.7 Lazy Limitations

Some operations such as `aCollection->size()` cannot be executed lazily since the size cannot be known without the whole collection. But in a suitable context such as `aCollection->size() > 3`, it is obvious that the full collection is not necessary after all. Even for `aCollection->size()`, `aCollection` does not need to be fully evaluated since we are only interested in the number of elements. If the computation of `aCollection` can be aware that only its size is required, a more efficient existence rather than value of each element might be computed.

4.8 Operation Caches

As well as using ‘lazy’ evaluation to defer computation in the hope that it may prove redundant, performance may be improved by caching what has already been computed in the hope that it can be re-used.

As a side-effect free language, OCL is very well suited to caching the results of iteration or operation calls. However for simple arithmetic, short strings and small collections, the cost of caching and re-use may easily exceed the cost of re-computation. For larger collections, the cache size may be unattractive and the probability of re-use too low. Such dubious benefits perhaps explain the reticence of implementations to provide result caching.

Model to model transformations depend on re-use of created output elements and so the Eclipse QVTd tooling [8] pragmatically provides caches for Functions and Mappings but not Operations or Iterations.

Empirical observation suggests that for object operations and derived properties, the re-use benefits and statistics are much more favorable and so such caching should be part of an OCL evaluator. We will shortly see another example where operation caching can be helpful.

4.9 Smart Select

The *select* iteration applies a Boolean predicate to filter a source collection.

```
sourceCollection->select(booleanPredicate)
```

In practice there are two common idioms associated with *select*.

Conformance Selection: It is very common to use

```
S->select(oclIsKindOf(MyType)).oclAsType(MyType)
```

This selects those elements of `S` that conform to `MyType`. This clumsy test and cast idiom was recognized in OCL 2.4 and a `selectByKind()` operation added to improve readability.

In practice each source collection is partitioned into a very small number of types that can be identified by compile-time analysis of the OCL expressions. A naive implementation may recategorize the type of each element in each invocation. A more efficient implementation should re-use the type categorization to partition into all types of interest on the first invocation and cache the partitions for re-use by subsequent invocations for any of the types of interest. This should of course only be performed after Common Sub Expression or Loop Hoisting has eliminated redundant invocations, and only if there is more than one residual invocation.

Content Selection: It is also common to use

```
S->select(element | element.name = wantedName)
```

This locates a matching content of `S` by choosing the appropriately named elements. This idiom treats the `S` as a `Map` with a `name` key, but whereas a `Map` returns the value in constant time, naive implementation of `select` incurs linear search cost.

For a single matching lookup, building the `Map` incurs a linear cost and so there is no benefit in an optimization. However in a larger application it is likely that the name lookup may occur a few times for the same name and many times for different names. Providing an underlying `Map` may be very beneficial, converting a quadratic performance to linear.

We will contrast the performance, with and without a `Map`, of the accumulation that computes

```
let S = Sequence{1..N} in
let t = S->collect(i|Tuple{x = i}) in
S->collect(i | t->select(x = i))
```

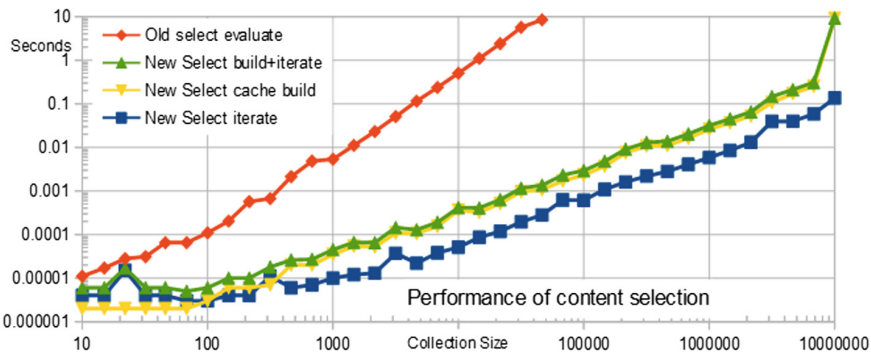


Fig. 8. ‘select’ Performance

The first two *let* lines build the table all of whose entries are looked up by the final line.

The top line of Fig. 8 shows the traditional naive full search for each lookup. The lower lines show the time to build the cache, the time to perform all lookups and their sum. The `Map` is clearly helpful for anything more than one lookup. As expected, it scales linearly rather than quadratically.

5 Context and Status

The OCL tooling must perform OCL expression analyses to use the foregoing `NewCollection` capabilities effectively

- Identify mutable collections - use alternative mutable operation
- Identify single/multiple use collections - configure shared laziness
- Identify content selects - configure lookup tables

However since OCL by itself is useless, OCL tooling cannot know whether or how to optimize. It is only when OCL is embedded in a larger application that provides the models and the related OCL expressions that OCL becomes useful.

For the simplest OCL application in which an interactive OCL expression is evaluated with respect to a model, the costs of the model and expression analyses may easily outweigh the benefits. No optimization may well give the snappiest interactive response.

For a more complex OCL application such as the OCL definition of model constraints, operations and properties supported by `OCLinEcore`, Eclipse OCL provides a code generator [3] that embeds the Java for the OCL within the Java for the `Ecore` model.

The code generator performs a variety of compile-time analyses and syntheses:

- Common SubExpression/Loop hoisting
- Constant Folding
- Inlining
- Dispatch tables

The code generator also prepares tables and structures that cannot be fully analyzed until the actual run-time models are available:

- Run-Time Type Information (e.g. `oclIsKindOf` support)
- Run-Time Navigability Information (unnavigable opposites)
- Run-Time Instances Information (`allInstances`)

Adding a few additional activities is structurally easy, and only a minor compile-time degradation. The results presented earlier use a manual emulation of what the automated analysis and synthesis should achieve ².

² Unifying the four concrete eager `Collection` types by a single lazy replacement is an API breakage that requires Eclipse OCL to make a major version number change. The code for lazy evaluations is therefore only available on the `ewillink/509670` branch in the Eclipse OCL GIT repository.

For OCL-based applications such as QVTc or QVTr [9], the Eclipse OCL code generator has been extended and appears to provide a twenty-fold speed-up compared to less optimized interpreted execution [4]. A smaller speed-up is to be expected for intensive *Collection* computations where most of the execution occurs in shared run-time support such as `Set::intersection()`.

6 Related Work

Lack of determinism in Model-to-Model transformation tools has been a regular irritation. e.g. https://bugs.eclipse.org/bugs/show_bug.cgi?id=358814.

Gogolla and Hilken [1] identify the lack of determinism for OCL collection conversions and suggested that certain combinations should be deterministic so that the following is true:

$$\text{SET} \rightarrow \text{asBag}() \rightarrow \text{asSequence}() = \text{SET} \rightarrow \text{asSequence}()$$

In this paper we make OCL collections fully deterministic and so all the suggested combinations are deterministic. The only open question is whether the deterministic order is *unknown*. If known, two different OCL implementations should yield the same deterministic result.

Lazy OCL evaluation is used by Tisi et al. [2] to support infinite collections. The authors consider their work as a variant semantics for OCL. Our alternative reading of the OCL specification allows infinite collections to be supported by regular OCL tooling provided eager operations such as `Collection::size()` are avoided. The default Bag-aware iteration provided by the `NewCollection` is incompatible with lazy Bags, however an alternative but less efficient approach could remedy this limitation.

Discomfort with the prevailing state of the art highlighted by these papers inspired the solution provided in this paper. The unified `Collection` implementation type is new. The deterministic `Collection` type is new. ‘Lazy’ OCL is not new, but the OCL expression analysis to exploit the lazy unified `Collection` type is new.

7 Conclusions

We have introduced a new underlying representation for a `Collection` implementation that unifies all four types and eliminates redundant conversion costs.

The new representation is deterministic allowing OCL and OCL-based model-to-model transformation tools to be deterministic too.

We have distinguished between program and machine failures so that the new representation can provide effective lazy evaluation capabilities.

We have used lazy evaluation to significantly reduce memory costs and to avoid redundant computations by allowing favorable algorithms to terminate prematurely.

We have linearized some quadratic costs by using mutable collections and a content cache for `select()`.

References

1. Gogolla, M., Hilken, F.: Making OCL collection operations more deterministic with restricting equations. In: 16th International Workshop in OCL and Textual Modeling, Saint-Malo, France, 2 October 2016. <http://www.db.informatik.uni-bremen.de/publications/intern/ocl2016-talk-lightning-mg-fh.pdf>
2. Tisi, M., Douence, R., Wagelaar, D.: Lazy evaluation for OCL. In: 15th International Workshop on OCL and Textual Modeling, Ottawa, Canada, 8 September 2015. https://ocl2015.lri.fr/OCL_2015_paper_1111_1115.pdf
3. Willink, E.: An extensible OCL virtual machine and code generator. In: 2012 Workshop on OCL and Textual Modelling (OCL 2012), Innsbruck, Austria, 30 September 2012. <http://st.inf.tu-dresden.de/OCL2012/preproceedings/14.pdf>
4. Willink, E.: Local optimizations in eclipse QVTc and QVTr using the micro-mapping model of computation. In: 2nd International Workshop on Executable Modeling, Exe 2016, Saint-Malo, October 2016. <http://eclipse.org/mmt/qvt/docs/EXE2016/MicroMappings.pdf>
5. Willink, E.: Safe navigation in OCL. In: 15th International Workshop on OCL and Textual Modeling, Ottawa, Canada, 8 September 2015. https://ocl2015.lri.fr/OCL_2015_paper_1111_1400.pdf
6. Dresden OCL Project. <http://www.dresden-ocl.org/index.php/DresdenOCL>
7. Eclipse OCL Project. <https://projects.eclipse.org/projects/modeling.mdt.ocl>
8. Eclipse QVT Declarative Project. <https://projects.eclipse.org/projects/modeling.mmt.qvtd>
9. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. OMG Document Number: ptc/16-06-03, June 2016
10. OMG: Unified Modeling Language (OMG UML), Version 2.5., OMG Document Number: formal/15-03-01, Object Management Group (2015). <http://www.omg.org/spec/UML/2.5>
11. Object Constraint Language. Version 2.4., OMG Document Number: formal/2014-02-03, Object Management Group (2009). <http://www.omg.org/spec/OCL/2.4>
12. USE: The UML-based Specification Environment. http://useocl.sourceforge.net/w/index.php/Main_Page

Step 0: An Idea for Automatic OCL Benchmark Generation

Hao Wu^(✉)

Department of Computer Science, National University of Ireland, Maynooth, Ireland
haowu@cs.nuim.ie

Abstract. Model Driven Engineering (MDE) is an important software development paradigm. Within this paradigm, models and constraints are essential components for expressing specifications of a software artefact. Object Constraint Language (OCL), a specification language that allows users to freely express constraints over different model features. However, one major issue is that the lack of OCL benchmarks makes difficult to evaluate existing and newly created OCL tools. In this paper, we present our initial idea about automatic OCL benchmark generation. The purpose of this paper is to show a developing idea rather than presenting a more formal and complete approach. Our idea is to use an OCL metamodel to sketch abstract syntax trees for OCL expressions, and solve generated typing constraints to produce the concrete OCL expressions. We illustrate this idea by using an example, discuss our work-in-progress and outline challenges to be tackled in the future.

1 Introduction and Related Work

Object Constraint Language (OCL), as a specification language in Model Driven Engineering (MDE), is *formally* used for writing rules that are not expressible by using models [1]. It plays a central role in many model-based engineering domains such as language engineering, model transformation and business process modelling. One particular example is ATL, a model transformation language that is built on top of OCL and it allows users to specify precise transformation rules for a set of model features. On the other hand, users can use OCL for different purposes including writing constraints/invariants for specific entities, specifying pre/post conditions over operations or methods and running queries over a set of features.

Recently, many approaches and techniques have been proposed for analysing or verifying models annotated with OCL [2–13]. These approaches either provide comprehensive case studies or tool support [6, 14–16] for analysing OCL constraints. However, a major issue is the lack of OCL benchmark. This is difficult for users to evaluate or choose suitable OCL tools for their own projects. This issue has recently been addressed by Gogolla and Cabot [17, 18]. Forming a collection of OCL benchmarks is necessary for OCL communities. Typically, there are two ways of forming such collections: (1) Extensively collecting existing models that are annotated with OCL constraints from different locations

such as code repositories and modelling zoos [32]. (2) Automatically generating a collection of OCL constraints with respect to user’s requirements. For example, users may be interested in evaluating scalability of their own tools. Thus, they need a large number of OCL expressions. Further, users may also focus on evaluating a particular aspect of a tool such as conflict detection. In this scenario, it would be very useful to automatically generate a large number of conflicted OCL expressions.

In this paper, we propose an idea of automatic OCL benchmark generation. We consider this idea as a complement to the idea of forming a benchmark via manually collecting existing models annotated with OCL. By exploiting this idea, users could create customised benchmarks to accommodate their own purposes such as generating property-specific OCL expressions.

2 The Proposed Idea

Our idea for automatic generating OCL benchmark is visualised in Fig. 1. Given a number of OCL constraints to be generated, users first define the properties for each OCL constraint. For example, a property call with a logic operator over an attribute. Here, we consider these properties are described in a standard OCL metamodel [1]. Second, we use a tree generator to generate the shape of an abstract syntax tree (AST) for each OCL constraint. This tree generator consults both the OCL metamodel and OCL concrete syntax to produce the ideal size of an AST, and generates a set of typing constraints for each AST. These constraints restrict possible types on each node in an AST. We then use an SMT solver to solve these constraints to derive a precise type for each node. Finally, we traverse the AST and instantiate each node with a concrete value. To form a OCL benchmark, we repeat these steps until the number of OCL constraints a user asked for is met.

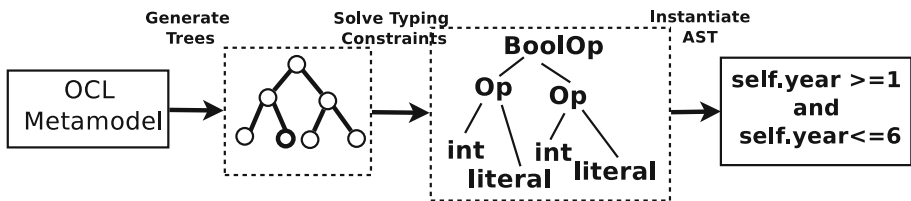


Fig. 1. The overview of an idea for generating a OCL benchmark.

2.1 An Example

In this section, we describe a scenario to illustrate our idea of automatic OCL benchmark generation. This scenario is based on our recent experience in evaluating a newly created OCL tool [19].

Figure 2 shows a UML class diagram that captures a relationship between a doctor and accident & emergency department in a hospital. Now consider a scenario where a user has already designed a tool for verifying OCL constraints, and would like to evaluate the performance and scalability of this tool on the OCL logical expressions with the model shown in Fig. 2. In this case, existing collected OCL examples such as those are in [17, 18] are no longer suitable for this scenario since they use different models and contain less number of constraints. Typically, measuring the performance and scalability of a tool involves running against a large number of OCL constraints. Further, this user requires a specific criteria that models must contain a large number of expressions using logical operators. Therefore, it would be very useful to generate a customised OCL benchmark for this specific scenario.

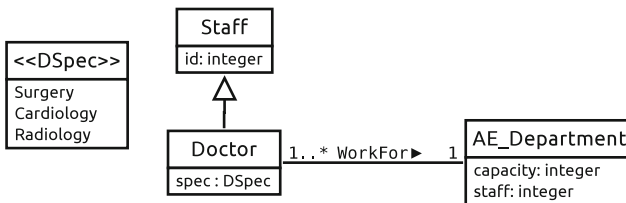


Fig. 2. A UML class diagram that represents a relationship between a doctor and Accident & Emergency department in a hospital.

To generate OCL logical expressions for this model, we first allow users to specify a type for each OCL constraint to be generated. To ensure the chosen types are valid, we use the standard OCL metamodel as a reference. For example, a user may select a property constraint for *id* attribute defined in the *Staff* class. The property call of an OCL constraint corresponds to the *PropertyCallExp* in the OCL metamodel that is shown in Fig. 3. Note that a user may select the same constraint type for multiple model features. For the reason of simplicity, we assume that users only choose a constraint type involving a single model feature.

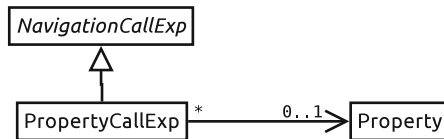


Fig. 3. A part of an OCL metamodel representing the relationship between two classes: *PropertyCallExp* and *Property*.

Once the type of an OCL constraint has been fixed, we then use a tree generator to sketch the shape of an abstract syntax tree based on consulting

the OCL concrete syntax. At this stage, users may specify a particular type expression and tree size. For example, a user may select a binary expression for a property constraint over the attribute *id*. The tree generator then tries to generate a tree that has the specified size. However, the size may vary and depends on OCL concrete syntax. For example, Fig. 4 shows an example of a generated abstract syntax tree for a binary expression. This tree has a size of 9, the root *R* produces two other binary expressions: *n1* and *n2*.

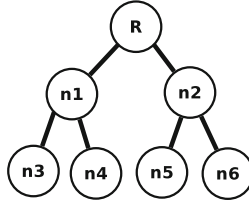


Fig. 4. An abstract syntax tree for a binary expression.

Now, we have the shape of an AST and the goal here is to work out correct types. More importantly, we need to ensure the type information preserved in an AST is consistent. For example, two boolean expressions cannot be connected by an arithmetic operator such as $+$ and $-$. In order to work out type information for each node, we generate a set of typing constraints for an AST and solve these constraints by using an SMT solver. To illustrate these typing constraints, we use Fig. 4 as an example.

Assume the AST in Fig. 4 represents a binary expression that captures an OCL property constraint for the attribute *id* in the class *Doctor* from Fig. 2. Since this tree represents a binary expression, the root *R* must be a binary operator such as $>$ or *and*. Node *n1* and *n2* could be another two OCL expressions containing two children nodes respectively. One of the possible kinds of expressions is that *n1* and *n2* are two binary expressions as well. For the reason of simplicity, let us assume that this is the case. If *n1* is a binary expression over *id*, then either *n3* or *n4* must be the attribute *id*¹. Similarly, this is the same for *n5* and *n6*.

Thus, we now can generate the following typing constraints for the AST in Fig. 4.

$$\begin{aligned}
 &(R \in OP_l) \wedge (n1 \in OP_c) \wedge (n2 \in OP_c) \wedge \\
 &(T(n3) = INT) \oplus (T(n3) = INT_LITERAL) \wedge \\
 &(T(n4) = INT) \oplus (T(n4) = INT_LITERAL) \wedge \\
 &(T(n5) = INT) \oplus (T(n5) = INT_LITERAL) \wedge \\
 &(T(n6) = INT) \oplus (T(n6) = INT_LITERAL) \wedge
 \end{aligned}$$

Here, *T* is a function that returns a particular OCL type. Sets OP_l and OP_c represent all possible binary operators. For the sub-tree that contains nodes *n3*

¹ In a more complex scenario, either *n3* or *n4* could also be an integer or an attribute.

and $n4$, exactly one of the nodes has an *INT* type. This is because the attribute *id* is an integer type. Since we consider a scenario that a constraint over a single attribute, the other node must be an integer literal (*INT_LITERAL*) type². Since each OCL constraint is a boolean expression and the tree represents a binary expression, R must be a logical operator. This implies that nodes $n1$ and $n2$ must be the operators that apply to two integer types and return a boolean type. For example, comparison operators: $>$ and $<$. Hence, we now can define the following operators for OP_l and OP_c .

$$OP_l = \{and, or, xor, implies\}$$

$$OP_c = \{>, >=, <, <=, <>, =\}$$

To generate constraints for OP_l and OP_c , we use an integer variable to encode each operator and constrain this integer variable to cover all possibilities. We then use an SMT solver to solve generated typing constraints and interpret the successful assignment for each node in the AST [20]. For example, Fig. 5(a) shows an example of solved typing constraints for the AST in Fig. 4.

Finally, we instantiate an AST with concrete values. Currently, we use a random value generation for each OCL literal type *string*, *int* and *boolean*. In this example, we use attribute *id* for each *INT* and randomly choose two integers for both *INT_LITERAL*. The final resulting OCL constraint for the attribute *id* in the class *Doctor* is shown in Fig. 5(b).

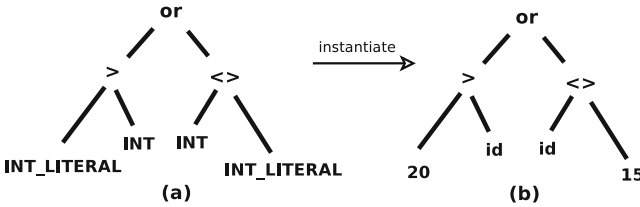


Fig. 5. (a) An example of solved typing constraints. (b) An abstract syntax tree with concrete values.

3 Work in Progress

We have implemented this idea into a prototype tool: OCLGen. We use OCLGen in our most recent work for generating a customised OCL benchmark to evaluate a technique for finding achievable features and OCL constraint conflicts [19, 31]. OCLGen uses the examples presented by Gogolla and Cabot as candidate models and further generates a much larger number OCL constraints based on the calculated configuration [17]. The configuration contains a set of different parameters including number of the quantifiers, logical operators and navigations. These

² In a multiple attributes scenario, the node could be either an integer literal or another integer type attribute.

generated OCL constraints cover a variety of features such as constraints over multiple inheritances, the nested quantified OCL expressions and random constraint conflicts. At the moment, OCLGen is able to handle the generation of simple binary and quantified OCL expressions containing arithmetic, navigation and logical operators.³

4 Challenges and Future Work

Though we have a working prototype for automatically generating OCL benchmarks, there are quite a few much more challenging problems remain.

1. Choosing/Designing an appropriate domain-specific language for describing benchmarks. Formally, users would be able to use a well-defined language to describe the kinds of benchmarks to be generated. For example, allowing users to quantify the number of operators in an OCL expression or specify the type of constraints to be generated such as navigations. Recently, a large number of OCL analysis and verification tools have been developed [6, 16, 21, 22]. However, not many of them evaluated their tools on a large number of inconsistent OCL constraints. The challenge here is that this language not only allows users to specify valid number of OCL constraints to be generated but also constraints cause inconsistencies. The generated benchmarks thus can be used for the purpose of evaluating the soundness of an OCL analysis tool.
2. Measuring the generated computational complexity of OCL benchmarks using a set of metrics. Users may use different or the same OCL benchmarks for evaluating existing, or their own OCL tools for different purposes. In this context, a set of suitable metrics for a benchmark is necessary. Those metrics can be used as a standard way of measuring the computational complexity of an OCL benchmark so that researchers and users in the community could have a clear idea of what tools are capable of. Even if the evaluation is not performed on the same benchmark [23]. For example, the metrics may include the measurement of the number of OCL data types, the maximum/minimum (AST) size of generated OCL expressions, the depths of quantifiers, etc. Further, a much more challenging problem here is that to automatically generate a benchmark meeting those metrics so that users can use it for focusing on a particular aspect of an evaluation.
3. Generating OCL benchmarks efficiently and effectively. Typically, the generation process should be completed within a reasonable time frame. As it can be seen from the example in Sect. 2.1, the shape of an AST and its type information can be naturally and formally tackled by constraints. The properties of an OCL expression such as the number of quantifiers can also be expressed as SAT/SMT constraints. The use of constraint solvers (SAT/SMT) have been proven to be successful in many domains [7, 24–27]. However, one problem

³ The fully generated benchmark is available at https://github.com/classicwuhao/maxuse/tree/master/maxuse_examples/benchmark.

of those solvers is that they usually do not scale very well. Based on our recent experience, we discover that sometimes those solvers may lose accuracy when the problem size is too big [19]. This is probably caused by the heuristic algorithms used within solvers. For this reason, the predication of how those solvers' will performance on a particular problem could be helpful to tell users what to expect [28]. Additionally, a benchmark formed by a mixture of manually created examples with generated ones could be a practical way for determining where a numerous number of OCL constraints needed.

In this paper, we have presented our initial idea of automatically generating OCL benchmark by producing skeletons of OCL abstract syntax trees based on an OCL metamodel and solving generated typing constraints for each AST. The experience of using our prototype tool OCLGen is the very first step towards proposing a complete framework for automatic OCL benchmark generation.

In the long term, we plan to tackle the above challenges individually and continue extending our work in OCLGen. This involves investigating the design of a domain-specific language for generating metrics-oriented OCL benchmarks. Though we have done preliminary work on generating graph-oriented instances, OCL constraint generation is much more challenging since we need to take many aspects into account such as tree shapes and typing constraints [29,30]. Further, we will also enhance our tree generator to generate more complex structures such as queries over a collection data type. Our ultimate goal is to solve these challenges listed above and build a framework for automatically generating customised OCL benchmarks that can be used for evaluating OCL analysis and verification tools to accommodate different user requirements.

References

1. Object Management Group: Object Constraint Language Version 2.4 (2014)
2. Beckert, B., Keller, U., Schmitt, P.H.: Translating the Object Constraint Language into first-order predicate logic. In: Verify Workshop at FLoC, Copenhagen, Denmark (2002)
3. Maraee, A., Balaban, M.: Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA 2007. LNCS, vol. 4530, pp. 17–31. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72901-3_2
4. Brucker, A.D., Wolff, B.: HOL-OCL: a formal proof environment for UML/OCL. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 97–100. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78743-3_8
5. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. ENTCS **115**, 39–47 (2005)
6. Clavel, M., Egea, M., de Dios, M.A.G.: Checking unsatisfiability for OCL constraints. ECEASST **24** (2009)
7. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using ‘off-the-shelf’ SMT solvers. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 432–448. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_28

8. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: DATE, Dresden, Germany, pp. 1341–1344 (2010)
9. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.* **73**, 1–22 (2012)
10. Dania, C., Clavel, M.: Ocl2fol+: coping with undefinedness. In: OCL@MoDELS (2013)
11. Semeráth, O., Vörös, A., Varró, D.: Iterative and incremental model generation by logic solvers. In: Stevens, P., Wasowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 87–103. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_6
12. Przigoda, N., Wille, R., Drechsler, R.: Ground setting properties for an efficient translation of OCL in SMT-based model finding. In: 19th MoDELS, pp. 261–271. ACM (2016)
13. Dania, C., Clavel, M.: OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In: 19th MoDELS, pp. 65–75. ACM (2016)
14. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL data types for SAT-based verification of UML/OCL models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21768-5_12
15. Wu, H., Monahan, R., Power, J.F.: Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In: 7th TASE, Birmingham, UK (2013)
16. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. *J. Syst. Softw.* **93**, 1–23 (2014)
17. Gogolla, M., Büttner, F., Cabot, J.: Initiating a benchmark for UML and OCL analysis tools. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 115–132. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38916-0_7
18. Gogolla, M., Cabot, J.: Continuing a benchmark for UML and OCL design and analysis tools. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 289–302. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_22
19. Wu, H.: Finding achievable features and constraint conflicts for inconsistent meta-models. In: Anjorin, A., Espinoza, H. (eds.) ECMFA 2017. LNCS, vol. 10376, pp. 179–196. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_11
20. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
21. Wille, R., Soeken, M., Drechsler, R.: Debugging of inconsistent UML/OCL models. In: 2012 DATE, pp. 1078–1083 (2012)
22. Balaban, M., Maraee, A.: Finite satisfiability of UML class diagrams with constrained class hierarchy. *ACM Trans. SEM* **22**(3), 24:1–24:42 (2013)
23. Cabot, J., Teniente, E.: A metric for measuring the complexity of OCL expressions. In: Model Size Metrics Workshop@MODELS 2006 (2006)
24. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Conference on Operating Systems Design and Implementation, pp. 209–224 (2008)

25. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 298–312. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_22
26. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: The 32nd PLDI, pp. 62–73. ACM (2011)
27. Phothisilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: Greenthumb: super-optimizer construction framework. In: 25th CC, pp. 261–262. ACM (2016)
28. Healy, A., Monahan, R., Power, J.F.: Predicting SMT solver performance for software verification. In: 3rd Workshop on FIDE, pp. 20–37 (2016)
29. Wu, H.: Generating metamodel instances satisfying coverage criteria via SMT solving. In: The 4th MODELSWARD, pp. 40–51 (2016)
30. Wu, H.: An SMT-based approach for generating coverage oriented metamodel instances. *IJISMD* **7**, 23–50 (2016)
31. Wu, H.: MaxUSE: a tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 348–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_23
32. Noten, J., Mengerink, J.G.M., Serebrenik, A.: A data set of OCL expressions on GitHub. In: 14th International Conference on Mining Software Repositories

Projects Showcase

SICOMORo-CM: Development of Trustworthy Systems via Models and Advanced Tools

Elvira Albert³, Pablo C. Cañizares¹, Esther Guerra², Juan de Lara^{2(✉)},
Esperanza Marcos⁵, Manuel Núñez¹, Guillermo Román-Díez⁴,
Juan Manuel Vara⁵, and Damiano Zanardini⁴

¹ UCM-TER, Universidad Complutense de Madrid, Madrid, Spain
pablocc@ucm.es, mn@sip.ucm.es

² UAM-miso, Universidad Autónoma de Madrid, Madrid, Spain
{Esther.Guerra, Juan.deLara}@uam.es

³ UCM-COSTA, Universidad Complutense de Madrid, Madrid, Spain
elvira@sip.ucm.es

⁴ UPM-COSTA, Universidad Politécnica de Madrid, Madrid, Spain
{groman,damiano}@fi.upm.es

⁵ URJC-Kybele Research Group, Universidad Rey Juan Carlos, Madrid, Spain
{esperanza.marcos, juanmanuel.vara}@urjc.es

Abstract. In this paper we present the SICOMORo-CM project. Its main aim is to advance the state of the art in the development of reliable and trustworthy systems by combining formal and model-based approaches. The project started on October 1st, 2014 and will last four years. The project consortium is integrated by five research groups based in Madrid (Spain) and it has been funded by the Regional Government of Madrid and the European Social Fund of the European Commission with a total of 635.088,65€.

1 Introduction

The main objective of SICOMORo-CM (Spanish acronym for Development of Trustworthy Systems via Models and Advanced Tools) consists in introducing methodologies, supported by tools, that allow the development of trustworthy and high quality software using a rigorous process that covers all its development phases. Therefore, SICOMORo-CM goes beyond partial steps that focus on isolated phases with the risk of wasting the results if there is no integrated framework for software development. SICOMORo-CM offers a joint scientific program organized into 9 scientific-technological objectives. These objectives include work focused on every phase of the software development cycle (modelling, model verification, validation, and system verification); work in transversal lines that require all phases (in particular, we highlight the design of the

Research partially supported by the Comunidad de Madrid project *desarrollo de Sistemas CONfiables mediante MOdelos y herRamientas avanzadas* SICOMORo-CM (S2013/ICE-3006). The project web site is <http://sicomoro-cm.es/>.

SICOMORo-CM workflow and the implementation of a virtual colaboratory as our main expected results); and work, in cooperation with 9 partner companies, on the application of the developed methodologies and tools in industrial environments (with emphasis in transportation, automotive and cloud systems).

SICOMORo-CM is a program of high relevance since the methodology and tools developed in the project will allow software to be delivered in a more effective, efficient, and reliable manner than today, accelerating the development cycle and lowering the operational costs. This has the potential to significantly improve the competitiveness of companies using our developed technologies. In particular, it will be especially relevant for SICOMORo-CM industrial partners. It is a priority of SICOMORo-CM to show the applicability of the methodologies developed by its use, in cooperation with the industrial partners, both in the development of software systems for major industries (e.g. transportation and automotive industry) and in the definition of service and modelling operations, and of cloud systems. We also expect that SICOMORo-CM will have a relevant impact in academia since we propose an ambitious reach-out and dissemination program of the results, that comprises publications and presentations in the most relevant international events and the organization of summer schools and specialized workshops.

The SICOMORo-CM program brings together leading national research groups in the areas of formal modelling and analysis of complex software systems. The principal investigators of the five academic groups that form SICOMORo-CM consortium, in spite of their relative youth have a broad expertise in research, with very relevant publications and remarkable experience in project management, both at national and international levels. These groups work on different but complementary research areas, providing an interdisciplinary background to this challenging research agenda. Furthermore, the project's interdisciplinarity is reflected in the application fields of the partner companies, which tackle divergent areas of software development, like rail signaling systems, and infrastructure security and protection. Finally, it is worth mentioning that only a program like SICOMORo-CM gathers under the same umbrella groups that work in complementary areas but with a common objective: creating high quality software that can be more useful to society.

2 Consortium of the Project

SICOMORo-CM is being implemented by five research groups located in the Madrid Region (Spain). Next, we briefly describe the main activities of each research group participating in the consortium.

The **UCM-TER** *Testing and Performance Evaluation* research group at Universidad Complutense Madrid (<http://antares.sip.ucm.es/testing/>) was founded by Manuel Núñez, Coordinator of the SICOMORo-CM project. The group has 17 members, with a good balance between senior researchers, fresh doctors and doctoral students. Although an important part of the work of the group concentrates on the Mathematical Foundations of Computer Science, the

applicability of the results is also a priority, with an important focus on the development of tools to support the theory.

The **UAM-miso** *Modelling and Software Engineering Research Group* at the Universidad Autónoma de Madrid (<http://miso.es>) was founded by Juan de Lara in 2013. Its current members include 3 professors, 4 PhD students, and 3 research associates. The main focus of the group is on the development of methods and tools for Model-Driven Engineering (MDE) and Domain-Specific Languages (DSLs).

The **COSTA** research group (<http://costa.ls.fi.upm.es>) is split between Universidad Complutense de Madrid (UCM-COSTA) and Universidad Politécnica de Madrid (UPM-COSTA). **UCM-COSTA** has 9 members, spanning from well-known experienced researchers to students. Elvira Albert is the co-founder and coordinator of the group. Group members have their Master and PhD degrees in Mathematics or Computer Science. The main focus of the group from its beginning has been to bring to practice theoretical results in program analysis. This has been obtained by developing a number of techniques with the goal of applying them to large-scale problems, and implement tools working on state-of-the-art programming languages and systems.

The **UPM-COSTA** research group is an emerging research group coordinated by Damiano Zanardini. Currently, this group has 3 members, two of them being staff researchers with a PhD Degree in Computer Science. Apart from the research lines shared with the UCM part of the COSTA group, research interest has been devoted to analysis of heap data structures (e.g. reachability and cyclicity) in Java and termination analysis of multithreaded Java.

The **URJC-Kybele** *Service Science, Management and Engineering and Software Engineering Research Group* at Universidad Rey Juan Carlos in Madrid (<http://www.kybele.es/>) was founded by Esperanza Marcos in 1998. The group has now 14 researchers (11 of them doctors) who collaborate also in Kybele Consulting, the group's spin-off. Modelling has been one of the main areas of interest for Kybele from its inception, even before the advent of Model-Driven Engineering. In fact, the most relevant projects run by the group since 1999 to date have been related with the provision of methods, tools and techniques based on models for different engineering purposes, like the development of information systems or the evolution of services. Indeed, Kybele was one of the first Spanish groups working in the area of service science management and engineering, which has become later the main area of interest of the group.

3 Objectives of the Project and Current Achievements

The project is structured around 9 objectives, which we briefly describe next, summarizing the results obtained so far.

Objective 1. Executable and trustworthy models. *UAM-miso.*

A first objective is to be able to specify DSLs (both syntax and semantics) in a cost-effective way, with the possibility of analysing the DSL semantics.

In the project, we consider both denotational and operational semantics. The former are specified using (model-to-model) transformations into a semantic domain, while the latter are specified using in-place model transformations. For this purpose, we are currently investigating processes to facilitate the creation of (graphical) DSLs. These are based on deriving the DSL concrete and abstract syntax based on examples [20], and include techniques to evaluate the effectiveness of the concrete syntax [15]. We have developed reusability mechanisms for model-to-model transformations and in-place transformations so that we can map families of DSLs (e.g., workflow languages) into semantic domains (Petri nets) [12, 29], or reuse in-place model transformations describing the execution semantics of the DSL [11]. This will allow the construction of libraries of reusable DSL semantics.

Objective 2. Verification of models and transformations. *UPM-COSTA.*

In this objective, we will analyse properties of both models and transformations, including model-to-model, model-to-text and in-place transformations. We will consider a case study in the verification of railway controllers provided by an industry partner.

Regarding model-to-model transformations, we are developing several techniques, for example based on static analysis of transformation definitions (using ATL) [10] and traceability analysis [18]. Regarding in-place model transformations, we are developing techniques based on backwards reasoning [8] to verify whether different model executions can violate given properties. Regarding models, we are working on analysing constraints [16], deriving techniques for slicing [22], and developing DSLs for an integral validation and verification of meta-models [21].

Objective 3. Transformations as a service. *UAM-miso.*

Based on our previous experience [9], our goal is to create a system able to optimize and execute transformations-as-a-service in the cloud, and its use in advanced scenarios (e.g., distributed and streaming transformations). We foresee integrating this system with the virtual collaborative environment of Objective 6, and the use of the modelling and verification techniques for cloud systems of Objective 8. We have currently developed a DSL to describe and generate infrastructure for MDE services [5], and we are collaborating with external groups to define transformation services for verification [26] and distributed transformations [3].

Objective 4. Verification and validation of systems. *UCM-COSTA.*

While the previous objectives dealt with models, the project also considers the verification of systems. This includes the verification of complex properties on sequential systems and concurrent programs, the development of scalable techniques for systems validation and the validation of concurrent programs. In particular, we are developing new techniques and tools to reason automatically on the behaviour of concurrent systems and understand all potential task interleavings that may arise along the execution. This is essential to prove both liveness and safety properties of the concurrent systems, like absence of deadlocks and absence of data races, or the termination

of all loops in the program. Among the contributions of the project to this objective we can mention [1,2,30].

Objective 5. Model-Based systems validation. *UCM-TER.*

To complement formal verification techniques, the project considers validation based on formal testing. This includes the definition of implementation relations, the design of algorithms for automatic generation of test cases and the proposal of passive testing techniques.

In the medium term, we will have new implementation relations and a tool to ensure its ease of use. At the end of the project, we expect to achieve all objectives. In particular, we will derive a full set of implementation relations, test case selection criteria, a formal methodology to perform passive testing for synchronous and asynchronous systems and tools that support these frameworks. We have already contributed to a state-of-the-art paper on formal testing [6] and developed new techniques for passive testing of systems with time information [23] and asynchronous communications [17].

Objective 6. Virtual Collaborative Environment. *UCM-COSTA.*

As the project aims at producing practical tools that can be used in combination along the development process, an objective is to develop supporting infrastructure for the flexible combination of tools, and their cloud-based execution. For this purpose, we have describe generic interfaces for tool integration and developed a prototype [13]. In the rest of the project, this prototype will be used to integrate the developed tools.

Objective 7. Modelling Service Operations. *URJC-Kybele.*

This objective has started in the second phase of the project. In order to support the development and analysis of service-oriented applications, we are currently developing a tool to support families of notations to model service operations, and we will support their formal analysis.

The first release of the tool has been delivered and can be downloaded at <http://kybele.es/innovaserv/>. It supports several notations for business modelling like Canvas [25], e3value [14], Service Blueprint [4] and Process Chain Network [28]. Since the tool has just been delivered only preliminary results are available [15] but some publications have already been submitted for consideration to high-impact conferences.

Objective 8. Cloud systems: model, verification and validation. *UCM-TER.*

We will apply the developed tools and techniques to model and analyse cloud systems. In particular, we have proposed specification techniques based on multi-level modelling [27], and we are developing an environment for the model-based analysis of cloud systems, including both expert rules and simulation for performance prediction [7]. We are also developing a methodology based on metamorphic testing for the validation of cloud systems [24].

Objective 9. Dissemination and exploitation. *UCM-TER.*

We are disseminating the project results primarily in academic conferences and journals, but we aim at disseminate and evaluate results in the software development community. For this purpose, during the second year we organized an “industry day” with the industrial partners and invited software

companies in the Madrid region, showcasing the different developed tools. We have also organized seminars around the project topics.

4 SICOMORO-CM: The Road Ahead

While we have obtained promising scientific and technical results – which have been published in international journal and conferences – there are still different remaining challenges, which will be tackled until the end of the project.

We expect to produce a methodology for the systematic and formal development of *all phases* of the *software-development* process. In addition to the underlying theoretical framework, we will provide tools that allow a smooth transition between the different phases of the development and the technologies used in them. While several individual tools have been developed, we aim at integrating them, using the virtual collaborative environment described in Objective 6. In particular, the environment will serve to *cloudify the different tools* and deliver their functionality adopting a software-as-a-service approach. This will serve to enable the use of MDE techniques and future integration with other services and tools. Techniques for the efficient and distributed execution of model transformations, as well as the development of streaming transformation techniques will be developed as well.

Regarding Objective 7, we will work on the integration of DSLs to support the modelling of service operations, bundled into a (collaborative and virtual) modelling environment, supporting formal verification of properties, value analysis and processes, as well as import/export operations of service operations models from/to other process modelling notations.

Another project goal is to develop *verification and validation techniques* applicable to several domains – like cloud, services, concurrent applications – to ensure that the verified systems satisfy some quality guarantees (e.g., deadlock-freeness, termination of all processes, existence of upper-bounds on resource consumption, etc.). We will continue working on techniques and tools in this direction. In particular, we will provide an *MDE framework* to support the verification of both models and model transformations. The framework is being developed as an open-source framework atop of Eclipse/EMF. An official Eclipse project proposal will be elaborated around the framework in order to enhance its visibility. This will contribute also to foster adoption by the industry due to the popularity of Eclipse among professional developers.

SICOMORO-CM is aimed to attract interested companies and organizations in order to enable technology transfer. The adoption of the techniques and tools delivered by the project is expected to have a verifiable impact in terms of improving the quality of the software developed and production cost-cutting. In-depth studies with program partners on the benefits provided by the methodologies and tools developed in SICOMORO-CM are also planned.

5 References to Related Projects

There are several projects, both at the national and international level, related to SICOMORo-CM. Next, we mention two of the FP7 European projects where members of SICOMORo-CM took part. Both projects have recently finished and some of their results have been used as inputs for SICOMORo-CM. MONDO [19] (<http://www.mondo-project.org/>) (*Scalable Modelling and Model Management on the Cloud*) focused on a very relevant research line of SICOMORo-CM, modelling, and on techniques to make modelling scalable. Instead, our focus in SICOMORo-CM is more on developing trustworthy systems. Envisage (<http://www.envisage-project.eu/>) (*Engineering Virtualized Services*) focused on applying formal approaches to services, having in mind that virtualized services can be used in the cloud. Again, SICOMORo-CM shares research interest with this project, in particular concerning services and the cloud.

References

1. Albert, E., Arenas, P., Gómez-Zamalloa, M.: Testing of concurrent and imperative software using CLP. In: 18th International Symposium on Principles and Practice of Declarative Programming, PPDP 2016, pp. 1–8. ACM Press (2016)
2. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: May-happen-in-parallel analysis for actor-based concurrency. *ACM Trans. Comput. Logic* **17**(2), 11:1–11:39 (2016)
3. Benelallam, A., Tisi, M., Sánchez Cuadrado, J., de Lara, J., Cabot, J.: Efficient model partitioning for distributed model transformations. In: Proceedings of SLE, pp. 226–238. ACM (2016)
4. Bitner, M.J., Ostrom, A.L., Morgan, F.N.: Service blueprinting: a practical technique for service innovation. *California Manag. Rev.* **50**(3), 66–94 (2008)
5. Carrascal, C., Sánchez, J., de Lara, J.: Building MDE cloud services with distil. In: CloudMDE@MODELS, CEUR Workshop Proceedings, vol. 1563, pp. 19–24 (2015)
6. Cavalli, A.R., Higashino, T., Núñez, M.: A survey on formal active and passive testing with applications to the cloud. *Ann. Telecom.* **70**(3–4), 85–93 (2015)
7. Cerro-Cañizares, P., Nuñez, A., de Lara, J.: MAGICIAN: model-based design for optimizing the configuration of data-centers. In: Proceedings of SEKE, pp. 602–607 (2017)
8. Clarisó, R., Cabot, J., Guerra, E., de Lara, J.: Backwards reasoning for model transformations: method and applications. *J. Syst. Softw.* **116**, 113–132 (2016)
9. Sánchez Cuadrado, J., de Lara, J.: Streaming model transformations: scenarios, challenges and initial solutions. In: Duddy, K., Kappel, G. (eds.) ICMT 2013. LNCS, vol. 7909, pp. 1–16. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38883-5_1
10. Cuadrado, J.S., Guerra, E., de Lara, J.: Static analysis of model transformations. *IEEE Trans. Softw. Eng.* **43**(9), 868–897 (2017)
11. de Lara, J., Guerra, E.: A posteriori typing for model-driven engineering: concepts, analysis, and applications. *ACM Trans. Softw. Eng. Methodol.* **25**(4), 31:1–31:60 (2017)
12. de Lara, J., Di Rocco, J., Di Ruscio, D., Guerra, E., Iovino, L., Pierantonio, A., Cuadrado, J.S.: Reusing model transformations through typing requirements models. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 264–282. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_15

13. Doménech, J., Genaim, S., Johnsen, E.B., Schlatte, R.: EASYINTERFACE: a toolkit for rapid development of GUIs for research prototype tools. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 379–383. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_22
14. Gordijn, J., Akkermans, H., Van Vliet, J.: Designing and evaluating e-business models. *IEEE Intell. Syst.* **16**(4), 11–17 (2001)
15. Granada, D., Vara, J.M., Brambilla, M., Bollati, V., Marcos, E.: Analysing the cognitive effectiveness of the webml visual notation. *Softw. Syst. Model.* **16**(1), 195–227 (2017)
16. Guerra, E., de Lara, J.: Automated analysis of integrity constraints in multi-level models. *Data Knowl. Eng.* **107**, 1–23 (2017)
17. Hierons, R.M., Merayo, M.G., Núñez, M.: An extended framework for passive asynchronous testing. *J. Logical Algebraic Methods Program.* **86**(1), 408–424 (2017)
18. Jiménez, Á., Vara, J.M., Bollati, V.A., Marcos, E.: Metagem-trace: improving trace generation in model transformation by leveraging the role of transformation models. *Sci. Comput. Program.* **98**, 3–27 (2015)
19. Kolovos, D., Rose, L., Paige, R., Guerra, E., Cuadrado, J., De Lara, J., Ráth, I., Varró, D., Sunyé, G., Tisi, M.: MONDO: scalable modelling and model management on the cloud. In: STAF Projects Showcase, CEUR Workshop Proceedings, vol. 1400, pp. 44–53. CEUR-WS.org (2015)
20. López-Fernández, J.J., Garmendia, A., Guerra, E., de Lara, J.: Example-based generation of graphical modelling environments. In: Wasowski, A., Lönn, H. (eds.) ECMFA 2016. LNCS, vol. 9764, pp. 101–117. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42061-5_7
21. López-Fernández, J.J., Guerra, E., de Lara, J.: Combining unit and specification-based testing for meta-model validation and verification. *Inf. Syst.* **62**, 104–135 (2016)
22. Mastroeni, I., Zanardini, D.: Abstract program slicing: an abstract interpretation-based approach to program slicing. *ACM Trans. Comput. Logic* **18**(1), 7:1–7:58 (2017)
23. Merayo, M.G., Núñez, A.: Passive testing of communicating systems with timeouts. *Inf. Softw. Technol.* **64**, 19–35 (2015)
24. Núñez, A., Hierons, R.M.: A methodology for validating cloud models using metamorphic testing. *Ann. Telecommun.* **70**(3–4), 127–135 (2015)
25. Osterwalder, A., Pigneur, Y.: *Business Model Generation: a Handbook for Visionaries, Game Changers, and Challengers*. Wiley, Hoboken (2010)
26. Di Rocco, J., Di Ruscio, D., Pierantonio, A., Cuadrado, J.S., de Lara, J., Guerra, E.: Using ATL transformation services in the MDEFForge collaborative modeling platform. In: Van Van Gorp, P., Engels, G. (eds.) ICMT 2016. LNCS, vol. 9765, pp. 70–78. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42064-6_5
27. Rossini, A., de Lara, J., Guerra, E., Nikolov, N.: A comparison of two-level and multi-level modelling for cloud-based applications. In: Taentzer, G., Bordeleau, F. (eds.) ECMFA 2015. LNCS, vol. 9153, pp. 18–32. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21151-0_2
28. Sampson, S.E.: Visualizing service operations. *J. Serv. Res.* **15**(2), 182–198 (2012)
29. 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
30. Zanardini, D., Albert, E., Villela, K.: Resource-usage-aware configuration in software product lines. *J. Logic Algebraic Methods Program.* **85**(1), 173–199 (2016)

Developer-Centric Knowledge Mining from Large Open-Source Software Repositories (CROSSMINER)

Alessandra Bagnato³, Konstantinos Barmpis⁵, Nik Bessis⁷,
Luis Adrián Cabrera-Diego⁷, Juri Di Rocco¹, Davide Di Ruscio¹(✉),
Tamás Gergely⁴, Scott Hansen¹¹, Dimitris Kolovos⁵, Philippe Krief⁸,
Ioannis Korkontzelos⁷, Stéphane Laurière⁹,
Jose Manrique Lopez de la Fuente¹⁰, Pedro Maló⁶, Richard F. Paige⁵,
Diomidis Spinellis², Cedric Thomas⁹, and Jurgen Vinju¹²

¹ University of L'Aquila, L'Aquila, Italy
{[juri.dirocco](mailto:juri.dirocco@univaq.it),[davide.diruscio](mailto:davide.diruscio@univaq.it)}@univaq.it

² Department of Management Science and Technology, Athens University
of Economics and Business, Athens, Greece
dds@aueb.gr

³ Softeam R&D Department, Paris, France
alessandra.bagnato@softeam.fr

⁴ FrontEndART Ltd., Szeged, Hungary
tamas.gergely@frontendart.com

⁵ Department of Computer Science, University of York, York, UK
{[konstantinos.barmpis](mailto:konstantinos.barmpis@york.ac.uk),[dimitris.kolovos](mailto:dimitris.kolovos@york.ac.uk),[richard.paige](mailto:richard.paige@york.ac.uk)}@york.ac.uk

⁶ Unparallel Innovation, Lda, Almada, Portugal
pedro.malo@unparallel.pt

⁷ Department of Computer Science, Edge Hill University, Ormskirk, UK
{[nik.bessis](mailto:nik.bessis@edgehill.ac.uk),[diegol](mailto:diegol@edgehill.ac.uk),[yannis.korkontzelos](mailto:yannis.korkontzelos@edgehill.ac.uk)}@edgehill.ac.uk

⁸ Eclipse Foundation, Toulouse, France
philippe.krief@eclipse.org

⁹ OW2 Consortium, Paris, France
{[stephane.lauriere](mailto:stephane.lauriere@ow2.org),[cedric.thomas](mailto:cedric.thomas@ow2.org)}@ow2.org

¹⁰ Bitergia, Madrid, Spain
jmanrique@bitergia.com

¹¹ The Open Group, Brussels, Belgium
s.hansen@opengroup.org

¹² Centrum Wiskunde and Informatica, Amsterdam, Netherlands
Jurgen.Vinju@cwi.nl

Abstract. Deciding if an OSS project meets the required standards for adoption is hard, and keeping up-to-date with a rapidly evolving project is even harder. Making decisions about quality and adoption involves analysing code, documentation, online discussions, and issue trackers. There is too much information to process manually and it is common that uninformed decisions have to be made with detrimental effects.

The research described has been carried out as part of the CROSSMINER Project, EU Horizon 2020 Research and Innovation Programme, grant agreement No. 732223.

CROSSMINER aims to remedy this by automatically extracting the required knowledge and injecting it into the developers' Integrated Development Environments (IDE), at the time they need it to make design decisions. This allows them to reduce their effort in knowledge acquisition and to increase the quality of their code. CROSSMINER uniquely combines advanced software project analyses with online IDE monitoring. Developers will be monitored to infer which information is timely, based on readily available knowledge stored earlier by a set of advanced offline deep analyses of related OSS projects.

1 Project Data

- **Acronym:** CROSSMINER (<http://www.crossminer.org>)
- **Title:** Developer-Centric Knowledge Mining from Large Open-Source Software Repositories
- **Partners:** The Open Group—*Project Coordinator*, University of York, University of L'Aquila—*Technical Coordinator*, Edge Hill University, Centrum Wiskunde & Informatica, Athens University of Economics and Business, UNPARALLEL, Softeam, Frontendart, Bitergia, OW2 consortium, Eclipse Foundation Europe GmbH
- **Start date:** 1 January 2017, **Duration:** 36 months.

2 Introduction

Open-source software (OSS) is computer software distributed with a license that allows access to its source code, free redistribution, the creation of derived works, and unrestricted use [5]. Unlike commercial software which is typically developed within the context of a particular organisation with a well-established business plan and commitment to the maintenance, documentation and support of the software, OSS is very often developed in a public, collaborative, and loosely-coordinated manner. This has several implications to the level of quality of OSS software as well as to the level of support that OSS communities provide to users of the software they produce. Consequently, *developing new software systems by reusing existing open source components* raises challenges related to at least the following activities [13]: (i) searching for candidate components, (ii) evaluating a set of retrieved candidate components to find the most suitable one, and (iii) adapting the selected components to fit the specific requirements.

Dependence on OSS projects can either be a blessing or a curse. The ability to accurately assess the risks and benefits of adopting particular OSS projects as components is essential to the software development community at large. The EU OSSMETER FP7 [4] project developed a distributed and horizontally-scalable platform for incremental analysis of multiple dimensions of open-source software projects including their source code, communication channels, and bug tracking systems. The aim of CROSSMINER is to extend the outcomes of the

OSSMETER project and to deliver an integrated open-source platform that will support the *development of complex software systems* by (1) enabling *monitoring, in-depth analysis and evidence-based selection* of open source components, and (2) facilitating *knowledge extraction* from large open-source software repositories.

The paper is structured as follows: Sect. 3 gives an overview of the CROSSMINER project. Section 4 outlines the planned evaluation process and concludes the paper.

3 The CROSSMINER Approach

Figure 1 shows a high-level overview of the CROSSMINER approach. It shows two major use cases and two minor user channels which are implemented using two architectural stages: online and offline. We describe the two major use cases here in some detail to clarify what CROSSMINER entails as a whole.

In step ❶ the tool engineers of Use case II use a domain-specific (graphical) editor in their IDE to compose new workflows of data sources and computations. This functionality is commonly available in big data analytics suites; here we specialise this functionality for typical OSS project analysis tasks. Mining and analysis tools will run incrementally in step ❷, and possibly on a remote server, to extract relevant information from a pre-configured set of projects and a list of projects configured by the software engineers of Use case I. The software engineers of Use case I have a wizard to configure CROSSMINER with a rich set of requirements (step ❸), which includes not only registering a set of projects of interest but also expressing preferences regarding the algorithms and processes used to project the mined information into the IDE. This configuration is an important step to make meaningful assessment possible later, since it makes the context and preferences of the engineer explicit to the platform in terms of technological, quality, configuration, and licensing aspects. Finally, step ❹ is when the acquired information is put to action, actively supporting the engineers via the IDE, managers via the web site, and the open-source community via GitHub integration. Typical examples of IDE services, which may be introduced or enhanced using this architecture are: *code assist*, proposing relevant code snippets, ranked by relevance and quality and informed by the earlier configuration; *infer/Fix project setup* to retrieve a list of ranked relevant reusable components, then set up relevant projects in the IDE and configure dependent projects to use them; *monitoring of development activities* of the engineers who will be notified of relevant facts pertaining to their current task context.

In the following the scientific and technological objectives to be achieved for realizing the approach shown in Fig. 1 are summarized.

3.1 Development of Source Code Analysis Tools

State of the art: Source code analysis has its firm fundamentals in compiler (front-end) construction [2] and reverse engineering [20]. Based on this theory and

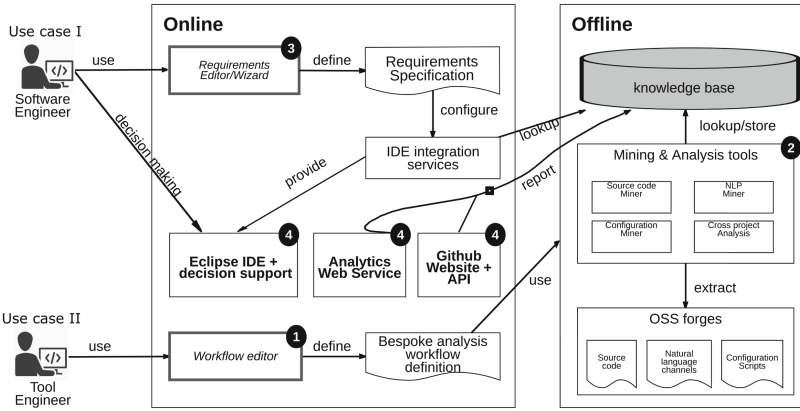


Fig. 1. CROSSMINER approach at a glance

technology, to extract meaningful and accurate metrics, we developed reusable front-ends generating informative reusable intermediate models with the OSS-METER project (for the Java and PHP languages). Recently, examples of source code analyses have been scaled up to acquire information over large source code and software install bases [8, 15]. However, mining information from source code at scale, made available in an integrated platform, including the acquisition, extraction, and querying of source code from groups of arbitrarily selected projects is just beyond the current state-of-the-art. Especially when the platform should cater for bespoke analyses based on the intermediate models there exists few related work in this regard [8].

Innovation: Mining source code artefacts to actively support decision making by software engineers *inside* their IDE requires scaling the technology for source code analysis to a level where we can mine in much larger corpora on the one hand, and on the other hand can enable much more context-specific (bespoke) analyses. At the same time the non-functional requirement of scalability must not imply a lower expected level of accuracy of the (bespoke) analyses. To scope this challenge in balancing trade-offs, and making it manageable for the current project we reason back from example decisions and the information required to make them. The main focus will be on *dependency management*: to help software engineers which (parts of) open-source components to depend on and how to manage these dependencies.

3.2 Development of Natural Language Analysis Tools

State of the art: Text mining tools to automatically extract, analyse, summarise and assess information found in communication channels and bug trackers related to OSS are valuable for supporting OSS development. Although there is a significant amount of literature analysing code repositories and communication channels, there are only very few attempts to use these sources to help

programmers as they program or to improve their output. For example, similarity methods have been proposed to identify the most relevant Stack Overflow discussions to the code that a developer is working on in an IDE and recommend them to improve developing performance [18]. Microsoft has just released Bing Developer Assistant for Visual Studio¹, which searches GitHub repositories, locates and presents examples of API usage relevant to the code being developed in Visual Studio.

Innovation: In CROSSMINER we plan to provide software developers with text analysis components integrating three innovative aspects: (i) a *user-oriented platform*, allowing users to tailor analysis to their needs by synthesising components into workflows. We will design and implement text mining components to identify the types of bugs and discussions in communication channels associated with an OSS project. Developers will be able to select the components of interest and synthesise them into workflows. Depending on the selected components the output will contain different information useful for the developers; (ii) we will investigate methods for using word embeddings for *representing text* in the domain of discussions about OSS; (iii) *new sources*, such as social media and Stack Overflow, and the analysis of *code snippets*.

3.3 Development of System Configuration Analysis Tools

State of the art: The practices, principles, and tools associated with Infrastructure as Code (IaC) and the analysis of software configuration management systems are in nascent phase. Studies to explore the characteristics of configuration code written in languages such as Puppet and Chef are scarce. Similarly, tools to carry out analyses of system configuration code have just started to emerge. Jiang and Adams [12] study the co-evolution of Puppet and Chef configuration files with source, test, and build code. They analyse the software repositories of 256 OpenStack projects and distinguish files as infrastructure, which contain configuration code in Puppet or Chef language, production, build, and test. They find that configuration code comes in large files, changes more frequently, and presents tight coupling with test files. Sharma et al. [19] carry out an empirical study of 4,621 Puppet repositories to understand the characteristics of configuration code written in Puppet. Puppet Forge² is the repository of Puppet modules and provides an evaluation of configuration code quality through a quality score based on three aspects: code quality score provided by Puppet-Lint³, compatibility with Puppet, and metadata quality. On the other hand, although empirical studies have examined the build aspect of software configuration management [1, 3, 9, 16, 17], the corresponding results have not yet been adopted by software developers.

¹ <http://visualstudiogallery.msdn.microsoft.com/a1166718-a2d9-4a48-a5fd-504ff4ad1b65>.

² <http://forge.puppetlabs.com>.

³ <http://puppet-lint.com>.

Innovation: CROSSMINER aims to significantly improve the state of the art in the configuration management domain by introducing advanced analysis techniques to process configuration code and other relevant artefacts. In particular, *collecting meta-data and computing various metrics* is the first step towards a comprehensive analysis. CROSSMINER aims to analyse configuration code written in various system configuration management languages including Puppet, Chef, and CFEngine as well as software configuration management metadata. Such source-code analysis will provide a uniform and comprehensive set of metrics that could be used to reveal the characteristics of configuration management systems. *Combining metrics and metadata* collected from configuration code with the results of source-code and natural language analysis using advanced static analysis techniques will fetch interesting insights and actionable results. *Interactive visualisation techniques* will be employed to engage the users in an effective and productive manner. Thus, suitable dashboard with DevOps-level information will be developed to show relevant metrics and insights about the analysed systems.

3.4 Development of Workflow-Based Knowledge Extractors

State of the art: OSS forges such as GitHub, GitLab and SourceForge and bug tracking tools such as Bugzilla and JIRA provide REST APIs with which users can perform queries (as well as some updates) on remote data (e.g. repository metadata, bug reports). To protect the underlying systems from uncontrolled data harvesting, many of these REST APIs impose key-based rate limits that clients cannot exceed and attempts to work around them (e.g. using multiple accounts/keys) can result to network-level blocking of the offending network endpoints. In addition to making use of remote APIs to extract knowledge from open-source projects, the wide adoption of distributed version control systems (predominately Git) where the entire history of repositories can be easily cloned, has triggered the appearance of a number of tools (e.g. Gitana [6], Gitstats⁴) that can analyse locally-cloned repositories and extract and present general-purpose metrics such as development activity over time/contributions per developer etc. While such metrics are useful, more advanced knowledge extraction (e.g. such as the one conducted in [14] which measures the adoption of different model-based technologies in Github-based open-source projects) typically requires bespoke analysis which includes the use of remote APIs, cloning and local analysis of repositories, natural language processing, HTML scraping, regular expressions etc.

Innovation: In CROSSMINER we envision the development of a framework that can support the development of declarative and efficient OSS project analysis workflows. Using the envisioned framework, engineers will be able to plug together OSS data harvesting, analysis and transformation components and define their dependencies and interactions at a high level of abstraction. The

⁴ <http://gitstats.sourceforge.net>.

framework will provide built-in support for recurring concerns such as network/API error recovery and data caching so that engineers can focus on the core analysis of the workflows, thus enhancing both productivity and maintainability. The framework will ship with robust built-in components for extracting information from widely-used systems such as Git(Hub), GHTorrent [10], Bugzilla, JIRA, NNTP and StackOverflow and will also provide extensibility mechanisms through which engineers can integrate additional components. We will also develop a set of hybrid textual/graphical editors and viewers through which engineers will be able to define knowledge extraction workflows, and also debug and monitor their execution at a high level of abstraction.

3.5 Development of Cross-Project Relationship Analysis Tools

State of the art: Over the last decade several platforms have been introduced to support automated analysis of open source software. All of them provide techniques and tools to analyse projects individually and do not mine projects relationships that instead can give more insight about existing OSS components. Some representative analysis platforms are OSSMETER, SQO-OSS (Alitheia Core)⁵, Openhub⁶, Qualipso⁷, Flossmetrics⁸, and RISCOSS⁹. Also, many OSS forges (e.g., SourceForge and GitHub) provide built-in measurement facilities for the OSS projects they host.

Innovation: In CROSSMINER we envision the development of advanced techniques able to investigate relationships among different open source projects and properly organise them in a dedicated knowledge base. Beyond the typical project dependency and conflict relationships we aim at identifying and managing additional ones e.g., license compatibility, API compatibility, etc. A general way to represent project relationships will be devised in order to enable relevant features including the following: *(i)* support for automated classification of OSS projects and discovery of related projects based on source code, configuration code, licensing, communication channel and bug tracking system analysis; *(ii)* adoption of clustering mechanisms supporting multidimensional classification of OSS projects; *(iii)* support for issuing notifications when quality indicators of selected OSS projects fall below a user-defined level; *(iv)* support for suggesting OSS projects that can be alternatively used instead of OSS components, which have been previously selected and integrated in the software being developed.

3.6 Development of Advanced Integrated Development Environments

State of the art: Most of the current IDEs include a wide range of features to enhance developer productivity from various code completion and refactoring

⁵ <http://cordis.europa.eu/project/rcn/79362.en.html>.

⁶ <http://www.openhub.net>.

⁷ <http://cordis.europa.eu/project/rcn/80465.en.html>.

⁸ <http://dl.acm.org/citation.cfm?id=1545011.1545457>.

⁹ <http://www.riscoss.eu>.

actions to style and error corrections. For Eclipse, the most notable related plug-ins are Codetrails Connect Community Edition¹⁰ and Eclipse Code Recommenders¹¹. These plugins learn how to use a new API from the source code of other applications or from watching how experienced developers use it, and share this information among team members through functions like code completion or snippet search. There are more novel approaches to extend the abilities of modern IDEs, to enhance programmer productivity and coding quality. For instance, in [18] authors proposed a novel approach that, given a context in the IDE, automatically retrieves pertinent discussions from StackOverflow, and evaluates their relevance. Another example is the Adinda approach (developed by van Deursen et al. [7]) that re-thinks IDE features as web services to facilitate informal inter-project communication and collaboration. Hora and Valente [11] developed a tool that helps API comparison based on compatibility and popularity information of GitHub projects. The concept of the Change-Oriented Programming Environment (COPE) research project¹² is to monitor software changes in real-time and provide actionable feedback to the developer through the IDE.

Innovation: As the above examples show, there are many different ways to give real-time suggestions to developers within their accustomed IDE. CROSSMINER brings a whole new dimension to the advanced IDEs because it collects, processes and stores a huge amount of data about open source components in a complex and cross-project data model. This enables intelligent recommendations to be provided to the developer, by going far beyond the current “code completion-oriented” practice. Our Eclipse plug-in for CROSSMINER will be developed primarily with the objective in mind that it improves the productivity of developers *in real-time* and *transparently*. Furthermore, CROSSMINER will learn from past recommendations and feedback from the developer so that even more relevant help will be given after being in use for a certain time.

4 Evaluation and Conclusions

In this paper we provided an outline of CROSSMINER’s envisioned technical contributions. The techniques and tools will be assessed by considering the needs of six end-user partners (in the domains of IoT, multi-sector IT services, API co-evolution, software analytics, software quality assurance, and OSS forges). The full chain of retrieval, analysis and presentation of results will be implemented on large-scale open-source forges like Eclipse and OW2 to assist users and demonstrate the benefits of the solution. The technical outcomes of the project as well as the evaluation results will be the subject of follow-up publications.

¹⁰ <http://marketplace.eclipse.org/content/codetrails-connect-community-edition>.

¹¹ <http://marketplace.eclipse.org/content/eclipse-code-recommenders>.

¹² <http://cope.eecs.oregonstate.edu/>.

References

1. Adams, B., De Schutter, K., Tromp, H., De Meuter, W.: The evolution of the Linux build system. *Electron. Commun. EASST* **8** (2008)
2. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Boston (1988)
3. Al-Kofahi, J.M., Nguyen, H.V., Nguyen, A.T., Nguyen, T.T., Nguyen, T.N.: Detecting semantic changes in Makefile build code. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 150–159. IEEE (2012)
4. Almeida, B., Ananiadou, S., Bagnato, A., Barbero, A.B., Rocco, J.D., Ruscio, D.D., Kolovos, D.S., Korkontzelos, I., Hansen, S., Maló, P., Drivalos, N., Paige, R.F., Vinju, J.J.: OSSMETER: automated measurement and analysis of open source software. In: *Proceeding of the Projects Showcase, Part of STAF 2015*, pp. 36–43 (2015)
5. Androutsellis-Theotokis, S., Spinellis, D., Kechagia, M., Gousios, G.: Open source software: a survey from 10,000 feet. *Found. Trends Technol. Inf. Oper. Manag.* **4**(3–4), 187–347 (2011)
6. Cosentino, V., Izquierdo, J.L.C., Cabot, J.: Gitana: a SQL-based git repository inspector. In: Johannesson, P., Lee, M.L., Liddle, S.W., Opdahl, A.L., López, Ó.P. (eds.) *ER 2015*. LNCS, vol. 9381, pp. 329–343. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25264-3_24
7. van Deursen, A., Mesbah, A., Cornelissen, B., Zaidman, A., Pinzger, M., Guzzi, A.: Adinda: a knowledgeable, browser-based IDE. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010*, vol. 2, pp. 203–206. ACM (2010)
8. 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: *36th International Conference on Software Engineering, ICSE 2014*, pp. 779–790, June 2014
9. Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: DevOps. *IEEE Softw.* **33**(3), 94–100 (2016)
10. Gousios, G.: The GHTorrent dataset and tool suite. In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR 2013*, pp. 233–236. IEEE Press (2013)
11. Hora, A., Valente, M.T.: Apiwave: keeping track of API popularity and migration. In: *Proceeding International Conference on Software Maintenance*, pp. 321–323. IEEE (2015)
12. Jiang, Y., Adams, B.: Co-evolution of infrastructure and source code: an empirical study. In: *Proceedings of MSR 2015*, pp. 45–55. IEEE Press (2015)
13. Karlsson, E.A. (ed.): *Software Reuse: A Holistic Approach*. Wiley, New York (1995)
14. Kolovos, D., Matragkas, N., Korkontzelos, I., Ananiadou, S., Paige, R.: Assessing the use of eclipse MDE technologies in open-source software projects. In: *Proceedings of 2nd OSS4MDE at MODELS 2015* (2015)
15. Landman, D., Serebrenik, A., Bouwers, E., Vinju, J.J.: Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *J. Softw.: Evol. Process* **28**(7), 589–618 (2016)
16. McIntosh, S., Adams, B., Hassan, A.E.: The evolution of Java build systems. *Empir. Softw. Eng.* **17**(4–5), 578–608 (2012)
17. McIntosh, S., Nagappan, M., Adams, B., Mockus, A., Hassan, A.E.: A large-scale empirical study of the relationship between build technology and build maintenance. *Empir. Softw. Eng.* **20**(6), 1587–1633 (2015)

18. Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., Lanza, M.: Mining StackOverflow to turn the IDE into a self-confident programming prompter. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp. 102–111. ACM (2014)
19. Sharma, T., Fragkoulis, M., Spinellis, D.: Does your configuration code smell? In: Proceedings of the Thirteenth International Workshop on Mining Software Repositories, MSR 2016. ACM (to appear)
20. Tonella, P., Potrich, A.: Reverse Engineering of Object Oriented Code. Monographs in Computer Science. Springer, New York (2004). <https://doi.org/10.1007/b102522>

Technical Obsolescence Management Strategies for Safety-Related Software for Airborne Systems

Simos Gerasimou¹(✉), Dimitris Kolovos¹,
Richard Paige¹, and Michael Standish²

¹ Department of Computer Science, University of York, York, UK

{simos.gerasimou,dimitris.kolovos,richard.paige}@york.ac.uk

² Defence Science and Technology Laboratory, Ministry of Defence, London, UK

Abstract. Complex software systems deployed in safety-critical and business-critical application domains (e.g., avionics, defence, healthcare) are meant to provide service for decades. Although many of these systems withstand technological evolution and infrequently undergo substantial changes, they will likely face software obsolescence problems during their lifetime. Resolving these obsolescence problems is an expensive, time-consuming and labour intensive process. This project investigates technical obsolescence problems in safety-related software for airborne systems, examines the applicability of obsolescence management strategies and explores methods to automate the resolution of software obsolescence.

1 Project Identity

Name	Technical Obsolescence Management Strategies for Safety-Related Software for Airborne Systems
Funding Body	Defence Science and Technology Laboratory, Ministry of Defence, UK
Beneficiary	University of York, UK
Start Date	1 October 2016
Project Duration	11 months

2 Introduction

The fast changing market trends and the constantly growing business and customer needs increase the pressure for technology innovations in hardware and software-based industries. In their effort to stay ahead of the competition and not fall behind the technology wave, businesses from these domains invest a considerable amount of resources, time and effort to adapt their software and

hardware *products*¹ to the latest processes, technologies and materials. As a consequence, software and hardware products have ever shorter life cycles. Existing products are upgraded more frequently, new products with extra functionality and features are continually developed, while the research for advancing the state-of-the-art is never ending [5]. Illustrative examples of “high-tech” products from the software and hardware domains include operating systems (e.g., Microsoft Windows and Mac OS) and consumer-oriented electronic devices (e.g., mobile phones and tablets), respectively.

Despite the wide adoption of technology innovations in consumer-oriented products, complex systems deployed in safety-critical and business-critical application domains (e.g., defence, healthcare and avionics) are infrequently subject to heavy changes [5]. Systems within these domains may be operational over many decades with limited, or even no, changes during their lifetime. Even minor design or implementation changes require to rerun all necessary verification and assurance processes to establish that the updated system still meets its functional and non-functional requirements, and complies with national and international standards. This is a long and costly process that causes these systems to fall into “technology stagnation” [19].

Due to the challenges associated with technology stagnation, complex systems within critical application domains are likely to face the problem of *obsolescence* [18]. A product becomes obsolete when it is no longer available or produced by its original manufacturers or suppliers [19]. Obsolescence affects both software and hardware, and depends on the product type (software or hardware). Bartels et al. [5] indicate that the reasons behind obsolescence can be: (1) *logistical*, i.e., loss of ability to obtain the products necessary to develop or maintain a system; (2) *functional*, i.e., updating a product due to changes in functional and/or non-functional requirements can result in its incompatibility with other system components; and (3) *technological*, i.e., end-of-support of a product due to more technologically advanced products becoming available [5].

The consequences of obsolescence can have detrimental effects on the sustainability of complex systems and should not be taken lightly. Given that safety-critical and business-critical systems require many years of research and development, these systems could face obsolescence problems before they become operational and always encounter obsolescence during their lifetime [18]. To illustrate the magnitude of the problem, over 70% of the components of a surface ship sonar system were obsolete before the system is deployed [20], while more than \$500m were spent to redesign an obsolete radar system under a US Air Force program [5]. Also, a recent report of the UK Ministry of Defence about the Eurofighter Typhoon project highlights that “the risk of obsolescence was exacerbated in the case of Typhoon, which was not operational until two decades after the project started” [3].

National and international standards such as JSP886 Vol. 7 Part 8.13 [2] and IEC 62402 [1] specify obsolescence management plans for reducing the risk

¹ We use the term *product* for any hardware or software component/system developed by a company/organisation and which is part of a larger system.

and impact of obsolescence. For instance, *proactive obsolescence management* involves taking preemptive actions before obsolescence actually occurs, e.g., by monitoring the life cycle of selected high-risk products or by finding functionally equivalent products from different suppliers. *Reactive obsolescence management* refers to deciding an appropriate plan for minimising the cost of resolving an obsolescence problem after it has happened, e.g., by negotiating with the supplier to provide aftermarket support or by investigating how to redesign or modernise the system. *Strategic obsolescence management* includes techniques that combine data about the estimated lifetime of system components with business trends and expected customer needs to calculate the sustainment costs of the system and to derive plans for design refresh [5]. Furthermore, researchers in the area introduced approaches for managing obsolescence and reducing the threats to the sustainability of complex systems [18, 19]. We review recent research in managing software obsolescence in Sect. 3.

In this project, we explore reactive strategies for managing software obsolescence in safety-related software for airborne systems [21]. More specifically, we investigate the extent to which *software modernisation* – i.e., approaches that involve changes to the system structure, adaptation to more advanced technologies, and functionality enhancement – can mitigate the problem of software obsolescence and extend the life, performance and reliability of existing systems. We adopt an experimental-based approach and use a set of demonstrators to explore different facets of software modernisation, including reverse engineering, program understanding, demonstration of functional equivalence of migrated code, change in hardware platform, maintenance of performance, and preservation of Design Assurance Levels (DAL).

The remainder of the paper is structured as follows. Section 3 introduces related work on software obsolescence including mitigation strategies for managing both technical and socio-technical obsolescence. Section 4 provides an overview of our project for supporting software modernisation and mitigating software obsolescence, including a sketch of the demonstrators that will be used to evaluate the research and artifacts developed during the project.

3 Software Obsolescence and Mitigation Strategies

Software obsolescence is an increasingly important problem that reduces the ability to maintain and support complex systems, especially those deployed in safety-critical and business-critical application domains (e.g., defence, health-care, finance) [5]. Nevertheless, this problem is usually overlooked due to hardware obsolescence. Given that in most complex systems the maintenance cost for software and hardware is comparable, equal effort should be made to manage software obsolescence [19].

Software becomes obsolete because (1) a vendor no longer sells (end-of-sale) or maintains (end-of-support) a product (technological obsolescence); (ii) hardware, embedding system, requirement or other software changes to the system lead to hardware and software incompatibility (functional obsolescence); and (iii)

the equipment used to build and test the software (e.g., hard drive, processors) is no longer accessible or does not function properly (logistical obsolescence). Software obsolescence might also occur due to socio-technical reasons, including the unavailability of suitably trained personnel (e.g., lack of Ada95, Cobol or Assembly programmers) and gradual loss of tacit knowledge about the platform or operating environment of the software [5].

Dealing with software obsolescence using proactive or strategic management plans includes methods that aim at reducing the risk of obsolescence. Typical examples include using modular software architectures with low coupling, making code more portable, and using open-source software with large user and developer communities. Although these methods help to mitigate the footprint of obsolete software, they are inadequate in forecasting or resolving completely the problem. Thus, most organisations opt to handle software obsolescence in a reactive mode.

Recent research and existing standards [1,2] propose reactive obsolescence mitigation strategies to manage both the technical and socio-technical aspects of software obsolescence. From a socio-technical perspective, supplier and supply chain management serves to place more risk on the supplier of COTS or software systems (e.g., delegating responsibility to platform providers such as Microsoft or PTC). Supplier management must go hand-in-hand with capital upgrade plans and software strategies for the organisation, and technical strategies for managing obsolescence must be “planned for” in the context of the supplier management plan [19].

From a technical perspective, many mitigation strategies are available for managing software obsolescence and modernising a software system [5]. *Software rehosting* (i.e., wholesale migration) and *systematic migration*, for instance, address the problem by migrating a software system to a new platform or development environment all at once and in a planned way, respectively. Systematic migration considers the software architecture and can be employed when appropriate technology (e.g., services, precise software interfaces, well-defined protocols) is exploited within the architecture. When rehosting or migration is unfeasible or overly expensive, *virtualisation and emulation* can be used to build virtual machines or emulators that will support the obsolete platforms (e.g., hardware or operating systems, libraries). Finally, *redevelopment* might be used to modify the affected system components so that they function correctly with the new hardware or software.

3.1 Obsolescence in Software Libraries and Frameworks

An increasing number of software systems is developed using publicly available/third-party products (i.e., frameworks and libraries) [13]. Many of these products provide robust and efficient functionalities that are available through Application Programming Interfaces (APIs). Although the use of APIs enhances modularity, reduces development time and increases the quality of software systems, it also reinforces the dependency of these systems on third-party software [9].

Since both software systems and products evolve independently, dependent software systems will likely encounter obsolescence issues during their lifetime [10]. New requirements or improved software designs lead to product updates, and thus to API modifications, that often break compliance with dependent systems. Other reasons include end-of-support of a software product and the introduction of a competing product with improved functionality and better features. Developers must select between the costly process of adapting a software system to resolve these obsolescence issues versus dealing with the imminent problems that arise from using obsolete software (e.g., bugs, security risks).

Resolving these obsolescence issues automatically is of great interest to this project. In the following paragraphs, we provide a brief overview of related work.

API Update. This issue occurs when a system uses an outdated API (source) and must be modified to start exercising a newer version of the same API (target). When changes between subsequent API versions are substantial (e.g., redesign of the API's architecture due to language or compiler updates), backward compatibility is not guaranteed [10]. Some approaches for resolving this issue use lexical comparison of method signatures, similarity detection between API versions, program differencing and origin analysis [7,15]. Other approaches record API changes in the form of refactorings and then recommend these refactorings to dependent systems to modify their code to the new API [8,11,22].

API Migration. This obsolescence issue arises when an API (source) must be replaced by a functionally equivalent API (target). Since the source and target APIs refer to completely different products (presumably with different architectures and API designs), this is a significantly more difficult issue. The challenge, in this case, is to transform the source code of dependent systems so that they are compliant with the target API [4]. The prevailing approaches to resolving this issue are *shallow* and *deep* transformation [6,16]. Shallow transformation entails modifying directly the source code that uses the source API to use the target API instead [4]. Deep transformation involves the generation of abstraction layers through which source code instructions that previously invoked the source API can now delegate the work to the target API [6]. Selecting between these approaches requires to consider development effort and maintainability as well as functional and non-functional aspects; shallow transformation is easier to implement but difficult to maintain, whereas deep transformation is more expensive but also more maintainable [16].

API Mapping Inference. Identifying mapping rules between source and target APIs remains a significant research challenge both for API update and API migration [17]. Although mapping inference between alternative APIs is traditionally a time-consuming developer-driven process [4,16], recent research proposes techniques for its automation. Simple differences between source and target APIs (e.g., rename, move, delete) can be detected using API comparison methods [8,11,22]. Another interesting technique involves mapping inference by analysing API usage patterns from software systems that have already undergone the adaptation [15]. Learning techniques have also been used to detect mappings

by analysing large open source software repositories [14]. MAM [23] uses API transformation graphs to infer API mappings between two API versions written in different languages. For a detailed review of techniques dealing with mining API mappings, see [17].

4 Project Overview

This project investigates how reactive strategies can mitigate software obsolescence problems in safety-related software for airborne systems [21]. Although many complex systems deployed in critical application domains have management plans in place to mitigate the impact of hardware obsolescence, this is hardly the case for software. For our project partner (DSTL), technological obsolescence problems include end-of-support of a crucial software component (which is part of a software system) that enforces adaptation to more advanced technologies, and the need to transform parts of a software system developed in a “legacy” programming language to a modern programming language (e.g., from Ada to C/C++). A common, but challenging, functional obsolescence problem of interest involves the migration of an entire software system from a legacy hardware platform to a modern, more powerful platform.

Resolving these software obsolescence problems is typically a manual and laborious process. In this project, we explore techniques to enable the partial automation of reactive mitigation strategies, targeting mainly C/C++ software systems. In particular, we propose a combination of code analysis, code-based transformation and verification/validation techniques for the *modernisation* of software systems. Figure 1 depicts an overview of the proposed approach to software modernisation. Through analysing the source code of the software system under consideration, we would gain insight into the architecture of the system, including interconnections between software modules and dependencies with external libraries and components. Furthermore, this code analysis will provide useful information on how to best address the obsolescence issues and re-architect the software, if needed.

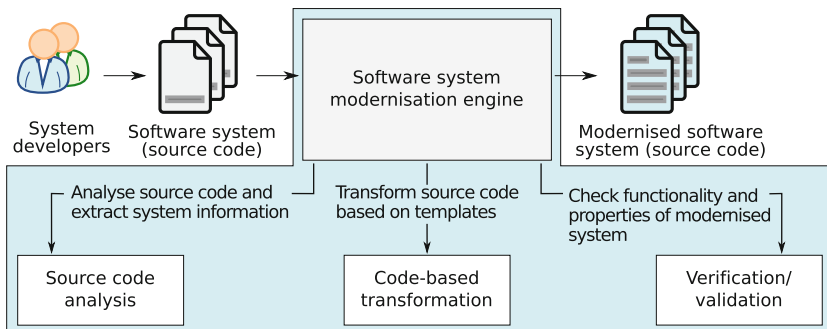


Fig. 1. High-level software modernisation approach.

Code-based transformation is based on the use of template-based programming. A template describes the generic code that will be produced by a transformation. Some of this generic code is static, i.e., it will be copied directly to the output of the transformation; the rest is dynamic, in the sense that it can be instantiated with data (in this case, from the legacy software) and then executed to produce output code. Template-based programming thus supports modular design and ideally easier reuse and maintenance. In this project, the overall code-based transformation will be implemented as a chain of templates, in order to better support future modification and maintenance and to ease verification and validation. Sets of templates will collaborate to implement specific concerns, e.g., message passing or space partitioning. Organising templates into sets (e.g., a “message passing” set) will allow for easier maintenance and verification.

Two types of transformations will be needed to support rehosting: extraction transformations (to “pull” abstract specifications from legacy code) and text generators. The transformations will be implemented using Epsilon [12], the rich C/C++ parsing and analysis infrastructure provided by the Eclipse CDT project, and ANTLR/Xtext to support additional languages such as Ada – the former for implementing the templates, the latter to extract abstract specifications from legacy code, in a form suitable for re-engineering and amenable to application of the templates. The transformation chain will automatically generate traceability information in a standard format (e.g., XML-based). This will thereafter be used to help to provide evidence that would be supportive of Software Level C. In particular, it will be used to demonstrate a partial validity argument, i.e., that all statements and expressions in the legacy program are transformed to statements and expressions in the retargeted program.

In addition, we aim to investigate (1) the feasibility of automatic generation of assurance cases from both the traceability information and the reengineered code; and (2) an outline qualification strategy for the transformation chain. The transformation chain as a whole will be validated by executing tests on inputs reused from the legacy system. Time permitting, the approach as a whole will be supplemented with an overview of an analysis of configurability, i.e., what parts of the transformation chain need to be configured in order to support additional changes to the legacy code, or to support a different target.

Demonstrators. The research focuses on the development of prototype/proof-of-concept demonstrators (case studies) of a software modernisation approach using template-based programming. The intention is to build working demonstrators iteratively and incrementally, to build up desired capability and ensure that there is always something demonstrable to our project partner. Each demonstrator will – at each iteration – demonstrate the application of the approach to a subset of software features, showing that functionality can be maintained. In this way, new features (e.g., message passing mechanisms) can be layered on top of existing demonstrators in new iterations.

These demonstrators intend to cover most of the software obsolescence problems that are of interest to our project partner. Demonstrators that illustrate technological obsolescence include software systems that use obsolete third-party

libraries (e.g., old XML library versions) and must switch to a newer version of the same library (API update) or to a completely different library (API upgrade). Interesting demonstrators to simulate functional obsolescence involve replacing hardware components (e.g., various types of sensors) with functionally equivalent components within the same microcontroller (e.g., Arduino, Raspberry Pi) or porting parts of this hardware infrastructure to a different microcontroller.

References

1. IEC 62402:2007: Obsolescence management. Application guide (2007)
2. JSPF 886, Volume 7, Part 8.13: Obsolescence management (2007)
3. Management of the Typhoon Project: House of Commons Committee of Public Accounts. Thirtieth Report of Session 2010–12 (2011)
4. Balaban, I., Tip, F., Fuhrer, R.: Refactoring support for class library migration. In: OOPSLA 2005, pp. 265–279 (2005)
5. Bartels, B., Ermel, U., Pecht, M., Sandborn, P.: Strategies to the Prediction, Mitigation and Management of Product Obsolescence. Wiley, Hoboken (2012)
6. Bartolomei, T.T., Czarnecki, K., Lämmel, R.: Swing to SWT and back: patterns for API migration by wrapping. In: ICSM 2010, pp. 1–10 (2010)
7. Cossette, B.E., Walker, R.J.: Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In: FSE 2012, pp. 1–11 (2012)
8. Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. *ACM Trans. Softw. Eng. Methodol.* **20**(4), 19:1–19:35 (2011)
9. Dig, D., Johnson, R.: The role of refactorings in API evolution. In: ICSM 2005, pp. 389–398 (2005)
10. Dig, D., Johnson, R.: How do APIs evolve? A story of refactoring: research articles. *J. Softw. Maint. Evol.* **18**(2), 83–107 (2006)
11. Henkel, J., Diwan, A.: Catchup!: Capturing and replaying refactorings to support API evolution. In: ICSE 2005, pp. 274–283 (2005)
12. Kolovos, D., Rose, L., Paige, R., Garcia-Dominguez, A.: The epsilon book. *Structure* **178**, 1–10 (2010)
13. Lämmel, R., Pek, E., Starek, J.: Large-scale, AST-based API-usage analysis of open-source Java projects. In: SAC 2011, pp. 1317–1324 (2011)
14. Nguyen, A.T., Nguyen, H.A., Nguyen, T.T., Nguyen, T.N.: Statistical learning approach for mining API usage mappings for code migration. In: ASE 2014, pp. 457–468 (2014)
15. Nguyen, H.A., Nguyen, T.T., Wilson Jr., G., Nguyen, A.T., Kim, M., Nguyen, T.N.: A graph-based approach to API usage adaptation. In: OOPSLA 2010, pp. 302–321 (2010)
16. Nita, M., Notkin, D.: Using twinning to adapt programs to alternative APIs. In: ICSE 2010, pp. 205–214 (2010)
17. Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T.: Automated API property inference techniques. *IEEE TSE* **39**(5), 613–637 (2013)
18. Romero Rojo, F.J., Roy, R., Shehab, E.: Obsolescence management for long-life contracts state of the art and future trends. *Int. J. Adv. Manuf. Technol.* **49**(9), 1235–1250 (2010)
19. Sandborn, P., Myers, J.: Designing engineering systems for sustainability. In: Misra, K.B. (ed.) *Handbook of Performability Engineering*, pp. 81–103. Springer, London (2008). <https://doi.org/10.1007/978-1-84800-131-2-7>

20. Singh, P., Sandborn, P.: Obsolescence driven design refresh planning for sustainment-dominated systems. *Eng. Econ.* **51**(2), 115–139 (2006)
21. University of York: Response to “Investigation into Technical Obsolescence Management Strategies for Safety-Related Software for Airborne Systems”, July 2016
22. Xing, Z., Stroulia, E.: API-evolution support with Diff-CatchUp. *IEEE TSE* **33**(12), 818–836 (2007)
23. Zhong, H., Thummalapenta, S., Xie, T., Zhang, L., Wang, Q.: Mining API mapping for language migration. In: *ICSE 2010*, pp. 195–204 (2010)

Mobile Health ID Card

Demonstrating the Realization of an mHealth Application in Austria

Malgorzata Zofia Goraczek¹(✉), Michael Sachs¹, Oliver Terbu², Lei Zhu³, Birgit Scholz³, Georg Egger-Sidlo³, Sebastian Zehetbauer², and Stefan Vogl²

¹ Department für E-Governance, Donau-Universität Krems,

Dr.-Karl-Dorrek-Straße 30, 3500 Krems, Austria

{malgorzata.goraczek,michael.sachs}@donau-uni.ac.at

² Österreichische Staatsdruckerei GmbH, Tenschertstraße 7, 1239 Wien, Austria

{terbu,zehetbauer,vogl}@staatsdruckerei.at

³ Research Industrial Systems Engineering (RISE), Forschungs-,

Entwicklungs- und Großprojektberatung GmbH, 2320 Schwechat, Austria

{lei.zhu,birgit.scholz,georg.egger}@rise-world.com

Abstract. The aim of this contribution is to present the project “Mobile e-Card” which demonstrates the concept for a realization of an eHealth service on a smartphone, based on an existing secure identity used in Austria. By building a demonstrator on the existing ID infrastructure of the Austrian Health ID Card it will be shown how a mobile application can be user-friendly and safe. It also indicates how existing functionalities from the Austrian Health ID Card could be adapted into potential functionalities for a mobile smartphone application, to be used by involved stakeholders of the health care system in Austria. Basic use cases of this application will be presented as “show cases”, as well as the technical concept of this mobile application.

Keywords: Mobile application · e-Services · eHealth · mHealth
Mobile devices

1 Introduction

This contribution presents an overview and insights into the ongoing research project “Mobile e-Card”. The focus lies on describing the project goals and involved project partners as well as the steps undertaken to develop the features of an mHealth application by the project team. The conceptualization and the potential for implementation in a real-world environment in Austria will be shown through exemplary use-cases.

According to Eysenbach (2001) the basic concepts of mHealth aim to describe the use of digital technologies and how they can enable health services. These concepts show not only the broad application of the usage of tools and data in the health domain, but also further aspects as the exchange of communication and information [1]. The concept of mHealth allows focus on smartphone-based patient applications which aim

to deliver services to patients, as well as to other stakeholders [2]. Positioning the project “Mobile e-Card” in the research domain of mHealth means to relate it to diverse core functionalities of mobile phones [3]. Functionalities implemented in this project range from information exchange between involved stakeholders of the Austrian health care system, e.g. how doctors and patients could use a dedicated smartphone app using Near Field Communication (NFC) or Internet for data transmission. Use cases describing these functionalities will be given in detail in Sect. 4. While building the demonstrator, the Austrian privacy law and the new GDPR (General Data Protection Regulation) were taken into account. The system architecture follows privacy- and security-by-design principles and will be described briefly in Sect. 3.

2 Project Overview

2.1 Project Objectives, Methods and Expected Results

Table 1 below presents an overview of the main objectives, used methods and expected results of the “Mobile e-Card” project, which aims to develop features of an mHealth demonstrator.

Table 1. Project overview

Objectives	Methods	Expected results
Create features of a potential mobile mHealth application	Gathering of requirements for potential features and description of use cases	Demonstrator of a mobile mHealth application
Develop a mobile mHealth application according to existing legal frameworks	Research on requirements obligated by legal frameworks	Demonstrator of an mHealth application in line with legal requirements
Evaluation of the user experience	Usability tests and interviews with involved stakeholders	User-friendly mobile mHealth application for involved stakeholders
Develop a technical security concept ensuring protection of personal data in regard to privacy and security	Risk analysis, data classification and technical measures protecting the data exchange	System architecture following privacy- and security-by-design principles

2.2 Funding and Current Status of the Project

The project “Mobile e-Card” is funded by the Austrian security research programme *KIRAS* of the Federal Ministry for Transport, Innovation and Technology (bmvit). The “Mobile e-Card” project started on the 01.10.2015 and will continue until the end of 30.09.2017. Further information on the project is also available online under: <http://bit.ly/2pwO78E>.

2.3 Consortium

The consortium consists of following project partners:

Research Industrial Systems Engineering GmbH (RISE) is leading the project consortium. RISE is an internationally established and recognized IT service provider with more than 20 years of IT experience in planning, IT architecture, IT infrastructure, IT strategy on the one hand and software development, smart piloting, project and risk management on the other. RISE supports the project with its knowledge and experiences in the areas of government, electronic identification, banking, payment systems and healthcare solutions. RISE has established large eHealth and ID projects around the world, e.g., Austrian eCard infrastructure.

Danube University Krems is specialized in applied research. The Department for E-Governance at the Danube University Krems conducts trans-disciplinary research on the effects of technological advances with regard to strategies, structure and processes in the digital network era. The focus of the team lies on evaluating the acceptance and usability of the demonstrator and its compliance with existing legal requirements.

Österreichische Staatsdruckerei (OeSD) is a leading international provider of identity management solutions. OeSD produces high-security identity documents, such as the chip- and biometrics-based passport. The team will contribute to the project with its knowledge in developing products and solutions in the domain of secure digital identities targeting secure environments.

Österreichische Ärztekammer/Austrian Medical Chamber (ÖÄK) represents the common professional, social and economic interests of all physicians in Austria. The partner supports the project in gathering the requirements from the perspective of stakeholders of the Austrian health care system, as well as in evaluating the demonstrator.

2.4 Tasks and Work-Packages

The whole project consists of six work-packages:

Work-package 1: Project Management ensures the successful coordination of project partners and involved stakeholders; planning and controlling of project stages, resources and finance; as well as project outcomes and their documentation.

Work-package 2: Requirements Engineering focuses on the one hand on gathering requirements of stakeholders and health experts that were identified in workshops and interviews. On the other hand, research on international and national market and evaluations of state-of-the-art implementations were conducted regarding security, identification and authentication.

Work-package 3: System Design uses privacy-by-design principles and data security.

Work-package 4: Implementation of the mHealth application follows the agile procedure of SCRUM to emphasize collaboration and flexibility to adapt to emerging changes regarding the requirements of involved partners and stakeholders.

Work-package 5: Impact Analysis and Evaluation: an online poll, as well as interviews and workshops with different stakeholder groups, were conducted. The evaluation of user experiences and usability tests with involved stakeholders to develop user-friendly mHealth applications is still ongoing.

Work-package 6: Dissemination aims to disseminate the project findings to a wide audience of practitioners and scientists across Europe.

3 Privacy- and Security-by-Design Architecture

The goal was to have an equivalent of the physical health card on the patient's smartphone which is compatible with the current Austrian health card infrastructure and can be used in a similar fashion but with extended functionality. The following challenges had to be taken into account while upholding security and privacy:

- Registration/activation on the smartphone
- Accounting during a consultation
- Communication between the doctor and the patient.

Registration is either done by using the Austrian electronic signature (Handy-Signatur/MOA ID), or by requesting a one-time verification code that is sent by e-mail. The physical health card, and optionally the passport, is used to avoid manual input of the required personal data (e.g. picture, name). If no passport was used, the user has to take a photograph to complete the registration form. A picture is required as a trust anchor in order to identify the patient in case of a consultation. After the patient gives explicit consent and the supplied data is approved, the mobile Health ID is activated. If the electronic signature in combination with passport was used, no manual confirmation is needed. In all other cases, the authenticity of the registration data is verified by the health care provider in an additional step. After activation, no sensitive data (e.g. social security number) is permanently stored on the patient's smartphone for privacy reasons. Instead, whenever data is required, this data is retrieved from a trusted mobile Health ID backend system (RESTful API) via a secure channel using certificate-based authentication.

Currently, in the event of a consultation, the patient provides his physical chip-based health card containing the patient's secret which is used to uniquely identify and authenticate the patient. A dedicated chip-card reader that is connected to the private network of the health insurance provider establishes a secure connection using the patient's secret in order to process accounting. The idea was to have a less invasive upgrade at minimal costs of the current landscape to support the mobile Health ID Application as further described in Sect. 4. From the reader's perspective, the mobile Health ID Application acts as a traditional chip-card by leveraging HCE (Host-based Card Emulation) while exchanging the same APDU (Application Protocol Data Unit) messages. In this manner, the mobile Health ID Application requests all required data from the trusted mobile Health ID backend via a secure channel facilitating the protection of sensitive data. During the consultation process, giving explicit consent to share personal data by pressing a button in the mobile Health ID Application additionally enforces privacy.

In order to provide additional use cases, which involve direct communication between doctors and patients, analogously an Admin Health ID Application was designed. This app resembles the physical Admin ID Card that is currently used by doctors for identification to the health insurance provider. Basically, it allows the establishment of a secure communication channel with the health insurance provider for accounting and other services. The registration process is similar to the mobile Health ID Application. These use cases may require additional information stored in external systems (e.g. clinical evidence, a patient’s health record stored in the ELGA system). Communication between both apps and external systems is also bridged via the mobile Health ID backend in the same secure way. Figure 1 provides an overview of the system architecture.

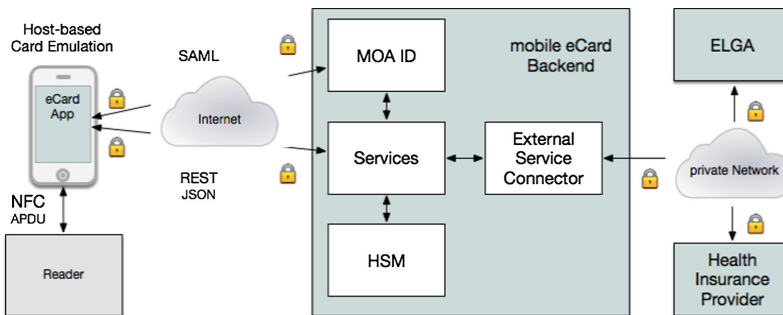


Fig. 1. Overview “Mobile e-Card” application system architecture

4 Technical Challenges

In addition to the overall design of the project “Mobile e-Card” described above, the following technical challenges were addressed.

- What is necessary to integrate the Mobile Health ID to the current system architecture? The actual Austrian Health ID Card infrastructure in medical centres has to be extended by a new interface, which provides the functionality to receive data from the patient’s smartphone and to communicate with the GIN-Network. To achieve this goal either the network based card reader has to be extended by an NFC-interface (Near Field Communication) or new NFC card readers have to be provided for medical centres, which communicate with the GIN-Network by LAN or WLAN.
- How can doctors get access to their patients’ data? After activation, the mobile Admin Health ID Application, described in Sect. 3, is able to exchange information with the patient’s mobile Health ID Application through the mobile Health ID backend using a secure mutual (client certificate, server certificate pinning) TLS (Transport Layer Security) connection over the Internet.

5 Project Outcomes: Selected Use Cases

Use cases were designed for physicians and for patients, and they encompass the essential functionalities of the current Austrian Health ID Card which is a chip card. However, the application could allow for the integration of further functionalities and services.

5.1 Registration

Use case for patients' Mobile Health ID Application: The application informs the user about the items necessary for registration with the Mobile Health ID Application: the Austrian Health ID Card, the Austrian digital signature, and ideally the passport. The data from the Austrian Health ID will be scanned and the respective field will be filled in automatically by taking a picture. Users can verify the correctness of the data and adapt it if necessary. A verified picture of the patient is taken from his or her passport using an NFC connection. To read the passport picture, the user has to insert some basic data from the passport; Fig. 2 shows this process. In cases where the person who registers does not have a passport, a selfie can be taken with the camera and be added to the profile. However, this profile picture would have to be verified by an authorized person and this process could be realized by a video call or by personal appearance in authorized premises.

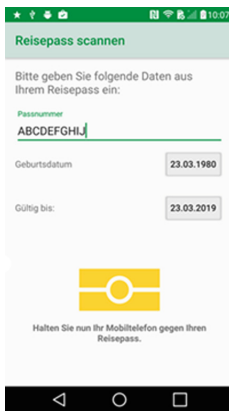


Fig. 2. Scan passport

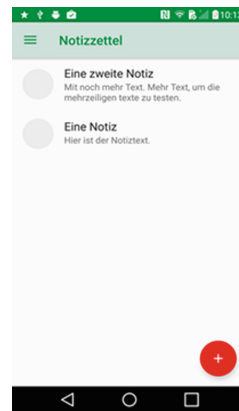


Fig. 3. List of notes in notepad

5.2 Visit to the Doctor

Use case for patients' Mobile Health ID Application: Currently, patients give their Austrian Health ID Card to the secretary of the doctor who inserts the card into the GIN

box to read it and start the medical consultation and transfer the data to the involved stakeholders such as the social insurance. With the Mobile Health ID Application, patients can use their mobile phone to connect to the doctor's Admin Health ID Application via NFC or wireless LAN. As doctors are asked to check the identity of patients on their first visit, usually using a photo identification document, the identity can then be confirmed through the verified photo that is saved in the data of the Austrian Health ID Card Application.

5.3 Personal Notepad

Use case for patients' Mobile Health ID Application: The notepad of the Mobile Health ID Application is saved in a secure environment. Patients can store information about substances they are allergic to; they can store all medication they take, including those medications that they buy in shops or pharmacies without the involvement of a doctor. Figure 3 (see Sect. 5.1) shows the notepad functionality of the Mobile Health Application.

5.4 Management of Data of Patients

Use case for doctors' Admin Health ID Application: Doctors can manage their patients, access their medical history, and delegate missions for other medical suppliers using their mobile application. Doctors can also access the patient's previous medical reports through the ELGA system (electronic medical records). In the ELGA system, files are saved decentralized at the institution of origin, and the Austrian Health ID Card is a key for the doctor to access them. Access to the ELGA system is granted to the doctor when a patient registers at his office, and this access is granted for a limited time only. It must be noted that if patients opted out of the ELGA system, access to a patient's history is not possible. Figure 4 shows a list view of different patients in the doctors' Admin Health ID Application. Patients whose ELGA access is active at the moment can be identified by the symbol on the right side of their list entry.

A doctor can delegate the ELGA access to other health care providers. For instance, the doctor can take blood samples from patients and send them to a laboratory for analysis. The results of the blood analysis can then be accessed by the doctor, again, through the ELGA system. The Admin Health ID Application can provide medical summary of key information of the patient as can be seen in Fig. 5. The list includes data such as blood type, vaccinations, allergies, implants, intolerances against substances, etc. Furthermore the application allows doctors to make documents by dictating texts through a speech-to-text functionality. Errors that might occur during the dictation can, of course, be edited manually. Figure 6 shows how doctors can create texts to be added to their patients' files. If documents need to be signed by the doctor, this can be done via the official digital signature of the Austrian state.



Fig. 4. List of patients



Fig. 5. General information about a patient



Fig. 6. Text editing

5.5 Information About Medical Services

Use case for patients' Mobile Health ID Application: As there is a lot of invalidated medical information available on the Internet, the Mobile Health ID Application provides information that was verified by a respected authority that is part of the Austrian health care system. As a result, the information from the application is trustworthy and can contain, for example, the following items that are currently available on relevant websites: information about the doctors (formation, training, opening hours), opening hours of pharmacies during the night, a search functionality for doctors and medications, information about vaccines and immunization therapies, etc.

5.6 Communication

Use case for doctors' Admin Health ID Application: Follow-up appointments for patients can be arranged through the use of the Admin Health ID Application. Patients will receive the notification of an appointment through their Mobile Health ID Application and can add the appointment to their personal calendars. Cancellation of appointments can be easily done via the application. The cancellation can be accompanied by information about replacements or information about alternative dates for appointments. Moreover, doctors can send documents or specific messages to patients that have registered with the doctor using the application. Such documents might include information about illnesses, diseases, vaccinations, letters of referral, or electronic prescriptions if implemented.

6 Innovation Potential and Expected Advances Beyond State of the Art

The innovation potential of the “Mobile e-Card” project lies in it being a solution for a Mobile Health ID Application based on an existing secure digital identity in Austria. The system architecture was designed to support security and privacy aspects according to Austrian legal requirements.

The “Mobile e-Card” offers a new and innovative application for potential mHealth services and a digital identity built on existing solutions, and further evaluates the implementation of functions with current technical and legal possibilities.

Acknowledgement. The project “Mobile e-Card” is funded by the Austrian security research programme *KIRAS* of the Federal Ministry for Transport, Innovation and Technology (bmvit).

References

1. Eysenbach, G.: What is e-health? J. Med. Internet Res. **3**(2), e20 (2001). <https://doi.org/10.2196/jmir.3.2.e20>
2. Mosa, A.S.M., Yoo, I., Sheets, L.: A systematic review of healthcare applications for smartphones. BMC Med. Inf. Decis. Mak. **12**, 67 (2012). <https://doi.org/10.1186/1472-6947-12-67>
3. World Health Organization: mHealth: new horizons for health through mobile technologies (2011). http://www.who.int/goe/publications/goe_mhealth_web.pdf

SECT-AIR: Software Engineering Costs and Timescales – Aerospace Initiative for Reduction

Richard F. Paige¹(✉), Athanasios Zolotas¹, Dimitrios S. Kolovos¹,
John A. McDermid¹, Mike Bennett²,
Stuart Hutchesson², and Andrew Hawthorn³

¹ University of York, York, UK

{richard.paige, thanos.zolotas, dimitris.kolovos, john.mcdermid}@york.ac.uk

² Rolls-Royce, Derby, UK

{mike.bennett, stuart.hutchesson}@rolls-royce.com

³ Altran Praxis, Bath, UK

andrew.hawthorn@altran.com

Abstract. Software is critical to the majority of functionality in avionics and aerospace systems. The amount of safety-related software in avionics is growing rapidly (doubling in size around every four years), and the costs of software programmes in industry are increasingly unaffordable – safety-related code can cost upwards of USD \$150 per line. At the same time, demands from avionics customers for increased scope and new functionality is increasing, and quality is non-negotiable: it is fixed by standards and safety requirements. The SECT-AIR project is addressing these cost and demand issues by focusing on automation in software engineering, with particular emphasis on model-based development. In this paper we provide an overview of the motivation behind the project, which started in 2016, and some of the key tasks it will carry out to help improve productivity, increase customer scope and maintain quality.

1 Project Identity

- **Project acronym:** SECT-AIR
- **Project title:** Software Engineering Costs and Timescales - Aerospace Initiative for Reduction
- **Project partners:** Rolls-Royce (Coordinator), BAE Systems, Leonardo, GE Aviation, MBDA, Rapita Systems, Altran Praxis, Cobham, D-RisQ, University of York, University of Oxford, University of Southampton
- **Website:** <http://www.ati.org.uk/atifundedprojects/113099/>
- **Funding:** Aerospace Technology Institute/Innovate UK, total budget GBP 10.1M.
- **Project start date/duration:** July 1, 2016 (36 months).

2 Introduction

The majority of functionality in modern aerospace and avionics systems critically depends on software. Unlike other software domains, where tradeoffs between

cost and quality can be made, quality requirements for avionics software systems are non-negotiable, fixed by standards such as DO-178C. As such, cost reductions for software have to be addressed by productivity improvements. One way to increase productivity is to better automate different engineering tasks, focusing on automating those tasks that are error-prone or repetitive, allowing engineers to focus on the challenging and creative aspects. Specific challenges that could be addressed to increase productivity and automation in avionics software engineering include:

- exploiting advanced architectures that support open and modular systems construction, e.g., service-oriented architectures, with the intent on reducing the burden of certification;
- optimising the development process via enhanced automated testing techniques and streamlining the handover between systems and software engineering;
- enhancing exploitation of model-based development, automated code generation, model-to-model transformation and automated formal analysis based on standards, so as to share development infrastructure costs and enable easier exchange of engineering artefacts;
- improving development processes for building high-integrity devices, e.g., FPGAs, system-on-chip, multicore.

At the same time, these challenges need to be addressed in a way that makes them ready to adopt by the avionics industry, taking into account the requirements for certification.

This paper discusses how the SECT-AIR project has contributed to tackling these challenges. Section 3 provides an overview of the overall SECT-AIR project execution. Section 4 outlines the different work packages with some focus on the model-based development tasks. Section 5 outlines the ongoing evaluation process and concludes the paper.

3 Execution

SECT-AIR is a project jointly funded by the industry partners and the Aerospace Technology Institute (ATI) via Innovate UK, the UK's innovation agency. As a result, the project's operation is overseen by both a project management team (led by Rolls-Royce) and an Innovate UK Program Manager, who monitors and assesses progress against key performance indicators, technical plans and financial plans. The project's overall execution is guided by the following key principles:

- Developing an industry-led aerospace sector wide strategy for software engineering;
- Exploiting technology used in other domains (e.g., high performance computing, multi- and many-core) in the aerospace domain;
- Validating research results using robust industrial aerospace case studies from project partners;

- Aligning industry development needs with academic research;
- Delivering mature technologies and processes at a high technical readiness level;
- Engaging with certification authorities, so as to be sure that the technologies and solutions produced will lead to systems that can realistically achieve certification. To support this, SECT-AIR will engage with the Civil Aviation Authority.

The overriding focus in all of the research activities (carried out in the work packages, described in the next section) of SECT-AIR is on automation as a means to improve productivity. Success measures in the case studies will be based on increases in automation in different processes.

4 Work Packages

Figure 1 shows the work package breakdown for SECT-AIR, including their inter-relationships. All work packages run for the full 36 months of the project, but having varying degrees of intensity, particularly in terms of the software and reports that are to be delivered.

Work Package (WP1) focuses on industry strategy and adoption of SECT-AIR deliverables: it includes baselining activities to measure current state of practice in aerospace software engineering, and will support experiments

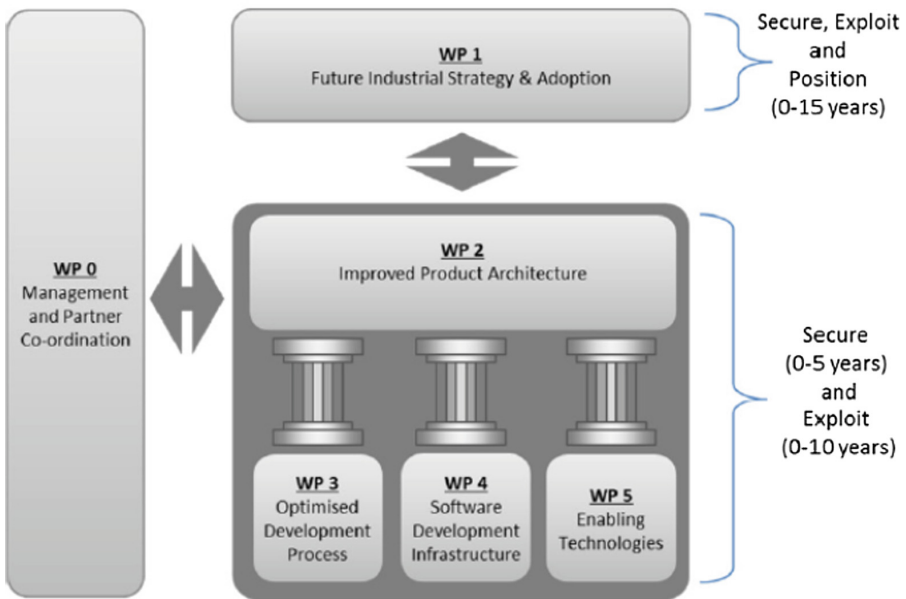


Fig. 1. Work package structure for SECT-AIR

designed to demonstrate potential improvements. It also aims to create a long-term industry strategy (for 10–15 years) that will ensure significant industry and academic alignment. It will be responsible for coordinating usable case study material for research in other work packages.

WP2 focuses on developing a common open and modular product architecture for aerospace software engineering, which will in turn enhance the product supply chain. It will also investigate the use of partitioning in aerospace products to reduce the amount of safety-critical functionality within embedded products. It will deliver demonstrators of open architectures at high technical readiness levels. The open architecture will be evaluated via demonstrator case studies that will be identified in WP1.

WP3 investigates software development processes. It is particularly targeting the application and evaluation of model-based techniques for requirements specification, to ensure composability and reuse of requirements: the view is that across a number of aerospace projects (requiring certification) there is a significant opportunity to reuse requirements – especially if they can be captured in a standard and tool-supported modelling language. As well, WP3 will investigate traceability and will build a toolchain to support end-to-end traceability from requirements models through to documentation and code. WP3 will in parallel trial the application of formal specification techniques used in synergy with model-based techniques to support richer analysis of requirements. The techniques that are developed will be applied to case studies that focus on novel interactive interfaces, e.g., for cockpit display systems. Such systems benefit from the development of rapid prototypes (even for critical systems) and thus make a challenging case study for model-based and formal methods, especially given that minimising the cost and time to change, for example, display formats, is an industry-wide issue in aerospace.

WP4 focuses on software development infrastructure for aerospace software engineering. It will endeavour to set a UK strategy for software development infrastructure improvements, and develop tool support to provide integration to various model-based languages used by industry, e.g., SysML, UML and various UML profiles such as MARTE. It will also aim to reduce the overhead for code-level verification and for generating qualification and assurance data. We discuss WP4 in more detail in the next subsection.

WP5, Enabling Technologies, aims to deliver a roadmap and demonstrator for next-generation high-performance obsolescence protected platforms. In particular it will aim to reduce the effort and costs for producing high-integrity firmware, such as system-on-chip and many-core processors. It will also work with certification authorities to ensure that the best practice that is developed will lead to systems that can achieve certification. A particular technique that will be investigated will be modular component-based development.

4.1 Model-Based Development

WP4 broadly focuses on software development infrastructure – i.e., producing a common platform (largely via reuse and agreement on standards) – for aerospace

software engineering. The vision is to exploit model-based development techniques as a means to increase productivity and to automate the error-prone, repetitive and tedious tasks of engineers. There are three components to achieving this vision:

- Exploiting model-based languages, including general-purpose languages (and tools) as well as techniques for building domain-specific languages for specific problems. There is broad agreement on the use of (profiles of) UML and SysML throughout SECT-AIR, and the project partners have predominantly agreed on the use of Eclipse Papyrus for many of their modelling needs within the project. A key requirement is to support efficient and effective development of UML profiles for individual partners, including those partners who have limited-to-no experience of building profiles. Thus, one objective of WP4 is to investigate new and efficient ways of *generating* profiles for UML, maximising reuse (e.g., when two or more profiles share features).
- Providing support for efficient and effective model transformations. Many partners in SECT-AIR need transformations to allow them to use new modelling technology such as Eclipse Papyrus in combination with legacy modelling technology, such as PTC Integrity Modeller. A particular objective of this work package is to ensure that any enabling model transformation technology is efficient, even when applied to large and complicated models. As such, WP4 is investigating *incremental* model transformations, where small changes to the source models mean that only small parts of the transformations need to be executed again. One of the techniques that will be considered for use is property traces, which have been used to support incremental code generation [1].
- Generating text and documentation. A particular use case for generation of text is producing template *assurance cases* from engineering artefacts (e.g., UML design models). Template assurance cases provide the structure and outline of an assurance argument that – after refinement and instantiation – will be delivered to a certifying authority along with evidence (e.g., test data, traceability data). Such arguments and evidence are critical in attempting to convince the certifying authority that a system is acceptably safe to deploy in its environment. Producing assurance cases is generally carried out manually and is an expensive process. Recertifying a system where requirements have changed is also a very expensive process. Hence, there is strong benefit from using model-to-text transformation to at least partly automate the process of producing assurance cases. SECT-AIR will investigate preliminary work on using weaving models [2] to underpin the process of generating template assurance cases from engineering artefacts supplied by industry partners.

Case studies from industry partners focusing on security and safety-critical systems development are being identified to help demonstrate the effectiveness of the techniques developed in this work package.

5 Evaluation and Conclusions

In this paper, we have given an overview of the SECT-AIR project, which aims to reduce software cost to the UK aerospace industry. The state of play is that the cost of software for aerospace is damaging the industry, and both industry and academia must collaborate in order to introduce controls, increase productivity and hence lower costs. SECT-AIR has been designed to bring together key UK capability in development of high-integrity aerospace software.

SECT-AIR has been running since June 2016 and already has delivered a number of results, including baselining surveys and experiments to establish cross-industry state of practice, an analysis of the use of model transformation across the industry, and the development of driver technology to allow SECT-AIR partners to use modern model management technology (i.e., Epsilon and its transformation language [3]) with legacy modelling tools (e.g., PTC Integrity Modeller). The focus over the next six months will be on leveraging these results to support the industry partners in more efficient development of their own profiles of UML and SysML, and on automating the generation of evidence that would be used as part of an assurance case for a high-integrity system.

Acknowledgements. This work was supported by the Aerospace Technology Institute and Innovate UK via the SECT-AIR grant, project number 113099.

References

1. 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
2. Hawkins, R., Habli, I., Kolovos, D.S., Paige, R.F., Kelly, T.: Weaving an assurance case from design: a model-based approach. In: 16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA, 8–10 January 2015, pp. 110–117 (2015)
3. 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

DEIS: Dependability Engineering Innovation for Cyber-Physical Systems

Ran Wei¹(✉), Tim P. Kelly¹, Richard Hawkins¹, and Eric Armengaud²

¹ University of York, York, UK

{ran.wei,tim.kelly,richard.hawkins}@york.ac.uk

² AVL List GmbH, Graz, Austria

eric.armengaud@avl.com

Abstract. The open and cooperative nature of Cyber-Physical Systems (CPS) poses a significant new challenge in assuring dependability. The DEIS project addresses this important and unsolved challenge by developing technologies that enable a science of dependable system integration. Such technologies facilitate the efficient synthesis of components and systems based on their dependability information, covering application domains such as automotive, railways, home automation and healthcare.

The DEIS project will bring significant impact to the CPS market by providing new engineering methods and tools reducing development time and cost of ownership, as well as supporting integration and interoperability of dependability information over the product life-cycle and over the supply chain.

1 Project Information

- **Project acronym:** DEIS
- **Project title:** Dependability Engineering Innovation for Cyber Physical Systems (CPS)
- **Project funding:** Total cost €4,889,290 (funded by H2020-EU.2.1.1)
- **Project partners:** AVL List GmbH (project coordinator), Siemens AG, General Motors Powertrain-Europe SRL, Ideas & Motion SRL, Portable Medical Technology Ltd, Fraunhofer Gesellschaft zur Förderung der angewandten forschung E.V, University of Hull, University of York, Politecnico of Milano, RSRC at Dundalk Institute of Technology.
- **Project start date/duration:** 01 January, 2017/36 months.
- **Project Website:** www.deis-project.eu

2 Background

It is expected that in the future, the physical and digital worlds will merge into a largely connected globe. This is backed by the emergence of notions such as Cyber-Physical Systems (CPS). CPS harbour the potential for vast economic and

societal impact in domains such as automotive, health care, home automation, etc. At the same time, if these systems fail, they may cause harm and lead to temporary collapse of important infrastructures, with catastrophic consequences for industry and society. Therefore, in order to realise the full potential for innovation of CPS, it is important to ensure the dependability of CPS.

CPS are typically loosely connected and come together as temporary configurations of smaller systems which dissolve and give place to other configurations. Therefore, the configurations CPS may assume over its lifetime are unknown and potentially infinite. Thus, currently available approaches are not possible to assure the dependability of CPS and it is a grand technology challenge to address the dependability of CPS.

The DEIS project identifies this challenge and takes a first step towards dependability assurance of CPS by focusing on system safety and security, because assuring safety of CPS is an indispensable prerequisite in order to realise their economic and social potential.

The key innovation in the approach of the DEIS project is the concept of Digital Dependability Identity (DDI), which was outlined by key partners of DEIS in [1]. A Digital Dependability Identity (DDI) contains all the information that uniquely describes the dependability characteristics of a CPS or a CPS component. This includes two key aspects: (a) attributes that describe the system's or component's dependability behaviour, such as faults and possible fault propagations through the CPS architecture, which can be described using concepts from the theory of safety contracts; and (b) requirements on how the component interacts with other entities in a dependable way, described in terms of the level of trust and assurance. DDI is therefore an evolution of current modular dependability assurance models for systems. It is produced during design, issued when the component is released, and then continuously maintained over the complete lifetime of a component or system. DDIs are used for the integration of components into systems during development as well as for the dynamic integration of systems into *systems of systems* in the field.

Based on the concept of DDI, the DEIS project seeks to provide a modelling and integration framework that lays the foundation for assuring the dependability of CPS.

3 Identified Challenges and Project Concept

A DDI is potentially a very useful digital artefact - It is a versatile dependability assurance case, the utility of which spans from component design to in-the-field operation of a CPS. However, the production and use of DDIs for heterogeneous systems poses a number of significant technological and engineering challenges that are pertinent and important in industry and motivate the objectives of the DEIS project.

DEIS identifies the following challenges for DDI:

1. Universal exchange of dependability information

- Currently there is a lack of common model representations for the exchange of dependability information. Thus, a precondition for DDI is the existence of an open dependability metamodel.
- DDIs should be sufficiently expressive to enable the component integrator to compile DDIs from the sub-components DDIs, for system synthesis.
- DDIs should optionally shield sensitive details through abstraction to protect the component provider's intellectual property.

2. Efficient dependability assurance across industries and value chains

- Component providers should be able to generate DDIs based on the dependability information of their components/systems that is already available in their existing tools.
- It must be possible to include the information contained in DDIs into the dependability assurance lifecycle and tool chain of the component integrator, to cater with the change in component/system requirements and integration context.
- Dependability should be considered from the early stages of design, so that a model-based approach can be adopted to enable automation, and eventually the automatic synthesis of systems/DDIs.

3. Dependable integration of systems in the field

- In CPS, dependability cannot be fully assured prior to deployment. This requires certain degree of automation in evaluation of DDIs. Thus, DDIs must become executable specifications.
- DDIs must be stored in a centralised repository so that they can be accessed in a uniform way, and changes in them are synchronised.
- Fully automated evaluation of DDIs required for highly dynamic environments.

To address these major technology and research challenges, the DEIS project sets out a four-stage innovation cycle, as illustrated in Fig. 1. The first stage aims at the fundamentals of DDIs, such as the definition of a universal format of DDIs based on an open information model for the exchange of dependability information. The second stage is to provide semi-automated and increasingly automated support for generating DDIs out of existing design and safety models, as well as for integrating the DDIs of sub-components into DDIs of larger systems by integrators. The third stage facilitates dependable integration of systems in the field through automated evaluation of DDIs that includes the concept of executable DDIs on-board. Finally, the fourth stage is to have continuous validation of the project results in realistic scenarios and case studies.

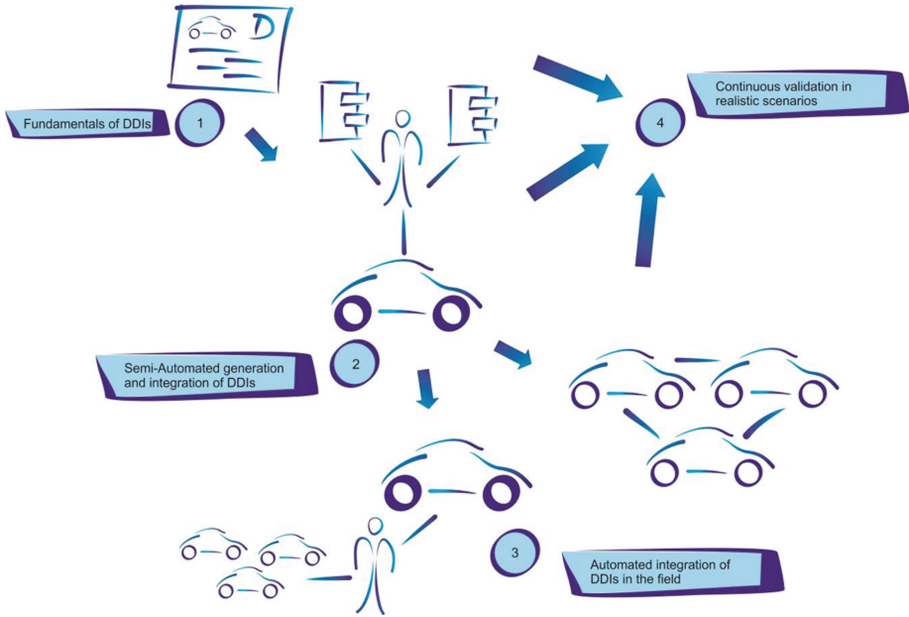


Fig. 1. DEIS project concept

4 Project Objectives

Based on the identified challenges and the project concept, the objectives of the DEIS project are set out as the following.

Objective 1. An open dependability exchange (ODE) metamodel and a universal format for specifying DDIs

Based on existing work, DEIS will produce an Open Dependability Exchange (ODE) metamodel. ODE provides the means to express, connect and communicate dependability information. With ODE, it would be possible to specify the level of trust of assured dependability properties with respect to the trust of the issuer and to the trust level of the promised services during field operation. DDIs should also be generated based on the information defined in ODE. DDIs will also be formalised in order to support their semi-automated evaluation.

Measurable sub-objectives of objective 1 are:

1. Definition of the Open Dependability Exchange (ODE) metamodel
2. Definition of general form of Digital Dependability Identity (DDI)
3. Tooling support for the manual modelling of DDIs
4. Tooling support to check the validity of DDIs.

Objective 2. A framework for the creation and modular synthesis of DDIs

Once an appropriate format for the ODE and DDIs is defined, DEIS will provide support for the creation and modular synthesis of DDIs from existing

dependability information. Such support is a prerequisite for the practical applicability of the approach. Thus a framework that serves such purpose will be developed, covering the following sub-objectives:

1. Tooling support for expressing existing dependability models in ODE-compliant format
2. Algorithms and tooling support for synthesis of DDIs
3. Algorithms and tooling support for integration of DDIs into the dependability assurance cases
4. Algorithms and tooling support for change-impact analysis on DDIs.

Objective 3. A framework for the in-the-field dependability assurance in CPS

A framework which enables the dependable integration of open CPS is required. Such framework consists a centralised DDI registry which is publicly available on-the-cloud. By using the centralised DDI registry, system manufacturers can check if their systems can be dependably integrated with already existing systems. Beside the centralised DDI registry, the framework should also enable on-board evaluation. With on-board evaluation, systems carry DDIs with them and evaluate if they can collaborate with each other in the field.

The framework covers the following sub-objectives:

1. Development of infrastructures for evaluation of integration of new systems in the field
2. Development of algorithms for the on-board evaluation of DDIs.

Objective 4. Development of autonomous and connected CPS use cases for different application domains, and validation of applicability and scalability of the DDIs

The scope of the project and the technology it develops is wide reaching and fundamental for CPS and the industries involved in the project (road transport, railway, healthcare). As such, the project results are expected to create significant impact. For this reason, it is a further objective of the project to validate the results in four realistic scenarios based on representative projects.

The studies of the four scenarios covers the following sub-objectives:

1. Evaluation of effectiveness of approach
2. Evaluation of applicability across industries
3. Evaluation of runtime mechanisms
4. Evaluation of systems produced in four case studies.

5 Related Work

In order to ensure successful project results, the project will not aim at developing an entirely new solution. In fact, the project will use, wherever appropriate, the results from other previous and current projects. In particular, projects that are related to DEIS are:

- VETESS: Verification and Testing to Support Functional Safety Standards
- SPES XT: Software Platform Embedded Systems
- SAFECER: Safety Certification of Software-Intensive Systems with Reusable Components
- CESAR/CRYSTAL: Cost Effective Small AiRcraft/CRITICAL sYSTEM engineering AccELeration
- SAFE: Safe Automotive soFTware architECTure
- EMC²: Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments
- SafeAdapt: Safe Adaptive Software for Fully Electric Vehicles
- OPENCROSS: Open Platform for EvolutioNary Certification Of Safety-critical Systems
- D-MILS: Distributed MILS for dependable information and communication infrastructures
- MAENAD: Model-based Analysis & Engineering of Novel Architectures for Dependable electric vehicles
- ATESTS2: Advancing Traffic Efficiency and Safety through Software Technology phase 2
- COMPASS: Comprehensive Modelling for Advanced Systems of Systems

The research in DEIS can also be based upon different existing approaches, like Component Fault Trees [2] and HiP- HOPS [3] for dependability analysis, GSN [4] for specifying safety cases, SACM [5] for specifying structured assurance cases, or ConSerts [6] as a starting point for runtime certification. All of these approaches were defined by partners involved in DEIS and have proven their value in many practical applications. The fundamental competence and previous work results provided by the partners involved in DEIS therefore build a sound basis, which gives confidence that the project objectives are achievable within the proposed time and budget of the project.

6 Expected Outcomes

CPS market accounted for almost €472 billion in the automotive, industrial, medical, aerospace and defence industries in 2012¹. By improving the development of dependable CPS and supporting the ad-hoc connection of dependable systems during field operation, DEIS holds the opportunity to bring a significant impact on the existing market and be an enabler for future solutions based on dependable CPS.

For the automotive market, providing means for ensuring dependability of collaboration at runtime will be the enabler to gain market shares in novel market segments. For the railway market, in particular, European rail transport, the harmonisation and supervision of safety certification are essential in a Single European Railway Area and for railway suppliers to deliver cost-efficient

¹ <https://ec.europa.eu/digital-single-market/en/news/european-industrial-strategic-roadmap-micro-and-nano-electronic-components-and-systems>.

and quality products. For the healthcare market, the need for improved science system integration for dependable, autonomous and connected CPS is also imperative.

The expected outcomes based on the objectives mentioned in Sect. 4 are:

1. Definition of the ODE metamodel and the specification of DDI.
2. A semi-automated framework for the generation and evaluation of DDIs.
3. A framework for the in-the-field dependability assurance in CPS.
4. Autonomous and connected CPS use cases, which is shown in Fig. 2.

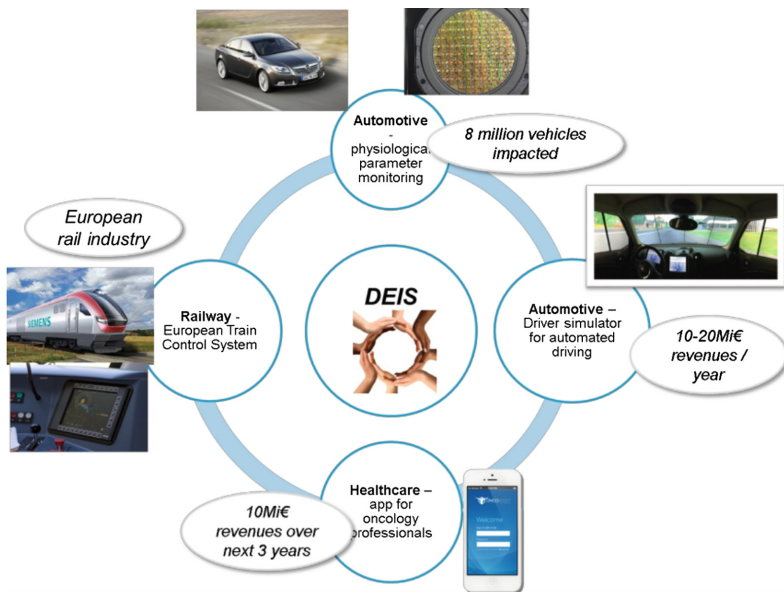


Fig. 2. Expected impact of the use cases on their respective markets.

7 Current Status

DEIS has been running for 3 months. Currently, requirements are being elicited among the partners of DEIS, which include:

- Requirements for the exchange of dependability-related information (ODE), and the specification of modular DDIs;
- Requirements for the tooling support needed to check the validity of the available dependability information and to model DDIs;
- Requirements for applying the DEIS approach within the automotive, healthcare and the railway domain and evaluating the project results compared to the state-of-the-art and the state-of-practice.

Acknowledgements. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 732242.

This paper summarises the project description, concepts, and plans from the original DEIS proposal. We acknowledge the original authors of the original DEIS proposal:

- Eric Armengaud, Nadine Knopper, Stephen Jones, Mario Oswald and Gerhard Griessnig (**AVL List GmbH, Austria**)
- Martin Rothfelder, Kai Höfig and Marc Zeller (**Siemens AG, Germany**)
- Alberto Pisoni, Federico Galliano and Massimiliano Melis (**General Motors Powertrain-Europe S.r.l, Italy**)
- Riccardo Groppo, Alberto Manzone, Paolo Santero and Marco Novaro (**Ideas & Motions S.r.l, Italy**)
- Eoin O'Carroll, Kevin Bambury and Richard Bambury (**Portable Medical Technology Ltd, Ireland**)
- Mario Trapp and Daniel Schneider (**Fraunhofer-Institute for Experimental Software Engineering, Germany**)
- Yiannis Papadopoulos (**University of Hull, United Kingdom**)
- Federica Villa, Franco Zappa, Alberto Tosi and Marco Marcon (**Politecnico di Milano, Italy**)
- Fergal McCaffery and Anita Finnegan (**RSRC ad Dundalk Institute of Technology, Ireland**).

References

1. Schneider, D., Trapp, M., Papadopoulos, Y., Armengaud, E., Zeller, M., Höfig, K.: WAP: digital dependability identities. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 324–329, November 2015
2. Kaiser, B., Liggesmeyer, P., Mäckel, O.: A new component concept for fault trees. In: Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software, SCS 2003, vol. 33, pp. 37–46. Australian Computer Society Inc., Darlinghurst (2003)
3. Papadopoulos, Y., McDermid, J.A.: Hierarchically performed hazard origin and propagation studies. In: Felici, M., Kanoun, K. (eds.) SAFECOMP 1999. LNCS, vol. 1698, pp. 139–152. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48249-0_13
4. Kelly, T., Weaver, R.: The goal structuring notation-a safety argument notation. In: Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases. Citeseer (2004)
5. Object Management Group: Structured Assurance Case Metamodel. <http://www.omg.org/spec/SACM/>. Accessed 27 Apr 2017
6. Schneider, D., Trapp, M.: Conditional safety certification of open adaptive systems. *ACM Trans. Auton. Adapt. Syst.* **8**(2), 8:1–8:20 (2013)

Author Index

- Aksit, Mehmet 30, 129
Albert, Elvira 367
Alfraihi, Hessa 317
Althaus, Ernst 43
Armengaud, Eric 409
Aßmann, Uwe 280
- Babur, Önder 129
Bagnato, Alessandra 375
Barpis, Konstantinos 375
Beaudoux, Olivier 302
Bencomo, Nelly 161
Bennett, Mike 403
Bessis, Nik 375
Béziers la Fosse, Thibault 3
Bill, Robert 136, 297
Bockisch, Christoph 30
Brambilla, Marco 154
Brucker, Achim D. 297
Brun, Matthias 302
Burdusel, Alexandru 60
Butting, Arvid 146
- Cabot, Jordi 154, 297
Cabrera-Diego, Luis Adrián 375
Cañizares, Pablo C. 367
Chhel, Fabien 302
Cimiano, Philipp 263
Clarísó, Robert 154
Clavreul, Mickaël 302
Cleophas, Loek 129
Combemale, Benoit 193
Cornelius, Gary 219
- de Lara, Juan 367
Di Rocco, Juri 375
Di Ruscio, Davide 375
Diskin, Zinovy 200
- Egger-Sidlo, Georg 394
Elyasaf, Achiya 225
- Fekete, Tamás 14
- Garcia-Dominguez, Antonio 161
Gérard, Sébastien 154
Gerasimou, Simos 385
Gergely, Tamás 375
Gogolla, Martin 172, 232, 297
Goraczek, Malgorzata Zofia 394
Götz, Sebastian 280
Greenyer, Joel 247
Greifenberg, Timo 146
Gritzner, Daniel 247
Guerra, Esther 367
- Hansen, Scott 375
Harel, David 225
Hawkins, Richard 409
Hawthorn, Andrew 403
Hildebrandt, Andreas 43
Hilken, Frank 172
Hochgeschwender, Nico 219
Hoffmann, Berthold 76
Hristakiev, Ivaylo 92
Hutchesson, Stuart 403
- Jouault, Frédéric 302
- Kästner, Andreas 172
Kelly, Steven 20, 178
Kelly, Tim P. 409
Kolahdouz-Rahimi, Shekoufeh 317
Kolovos, Dimitrios S. 403
Kolovos, Dimitris 375, 385
König, Harald 200
Korkontzelos, Ioannis 375
Köster, Norman 263
Krief, Philippe 375
Kulkarni, Vinay 184
- Lano, Kevin 317
Laurière, Stéphane 375
Lawford, Mark 200
Lopez de la Fuente, Jose Manrique 375
- Maibaum, Tom 200
Maló, Pedro 375

- Mansutti, Alessio 109
Marcos, Esperanza 367
Marron, Assaf 225
Mayerhofer, Tanja 193
Mazak, Alexandra 136
McDermid, John A. 403
Mezei, Gergely 14
Miculan, Marino 109
Minas, Mark 76
Mosca, Domenico 43
Mottu, Jean-Marie 3
- Núñez, Manuel 367
- Paige, Richard F. 375, 385, 403
Peressotti, Marco 109
Plump, Detlef 92
- Reddy, Sreedhar 184
Rensink, Arend 30
Román-Díez, Guillermo 367
Rumpe, Bernhard 146
- Sachs, Michael 394
Scholz, Birgit 394
Spinellis, Diomidis 375
Standish, Michael 385
- Tekinerdogan, Bedir 129
Terbu, Oliver 394
Thangaraj, Jagadeeswaran 331
- Thomas, Cedric 375
Tisi, Massimo 3
- Ulaganathan, SenthilKumaran 331
- Vallecillo, Antonio 232, 297
van den Brand, Mark 129
Vara, Juan Manuel 367
Vinju, Jurgen 375
Vogel-Heuser, Birgit 136
Vogl, Stefan 394
Voos, Holger 219
- Wei, Ran 409
Weiss, Gera 225
Werner, Christopher 280
Willink, Edward D. 297, 340
Wimmer, Manuel 136
Wortmann, Andreas 146
Wrede, Sebastian 263
Wu, Hao 356
- Yassipour-Tehrani, Sobhan 317
Yildiz, Bugra M. 30
- Zanardini, Damiano 367
Zehetbauer, Sebastian 394
Zhu, Lei 394
Zolotas, Athanasios 403
Zschaler, Steffen 60