Reiner Hähnle
Wil van der Aalst (Eds.)

ARCoSS
LNCS 11424

Fundamental Approaches to Software Engineering

22nd International Conference, FASE 2019 Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019 Prague, Czech Republic, April 6–11, 2019, Proceedings

Springer Open

Lecture Notes in Computer Science **11424**

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

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

More information about this series at http://www.springer.com/series/7407

Reiner Hähnle · Wil van der Aalst (Eds.)

Fundamental Approaches to Software Engineering

22nd International Conference, FASE 2019
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019
Prague, Czech Republic, April 6–11, 2019
Proceedings

Springer Open

Editors
Reiner Hähnle
Technische Universität Darmstadt
Darmstadt, Germany

Wil van der Aalst
RWTH Aachen University
Aachen, Germany

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-16721-9 ISBN 978-3-030-16722-6 (eBook)
https://doi.org/10.1007/978-3-030-16722-6

Library of Congress Control Number: 2019936008

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

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

ETAPS Foreword

Welcome to the 22nd ETAPS! This is the first time that ETAPS took place in the Czech Republic in its beautiful capital Prague.

ETAPS 2019 was the 22nd instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference established in 1998, and consists of five conferences: ESOP, FASE, FoSSaCS, TACAS, and POST. Each conference has its own Program Committee (PC) and its own Steering Committee (SC). The conferences cover various aspects of software systems, ranging from theoretical computer science to foundations to programming language developments, analysis tools, formal approaches to software engineering, and security.

Organizing these conferences in a coherent, highly synchronized conference program enables participation in an exciting event, offering the possibility to meet many researchers working in different directions in the field and to easily attend talks of different conferences. ETAPS 2019 featured a new program item: the Mentoring Workshop. This workshop is intended to help students early in the program with advice on research, career, and life in the fields of computing that are covered by the ETAPS conference. On the weekend before the main conference, numerous satellite workshops took place and attracted many researchers from all over the globe.

ETAPS 2019 received 436 submissions in total, 137 of which were accepted, yielding an overall acceptance rate of 31.4%. I thank all the authors for their interest in ETAPS, all the reviewers for their reviewing efforts, the PC members for their contributions, and in particular the PC (co-)chairs for their hard work in running this entire intensive process. Last but not least, my congratulations to all authors of the accepted papers!

ETAPS 2019 featured the unifying invited speakers Marsha Chechik (University of Toronto) and Kathleen Fisher (Tufts University) and the conference-specific invited speakers (FoSSaCS) Thomas Colcombet (IRIF, France) and (TACAS) Cormac Flanagan (University of California at Santa Cruz). Invited tutorials were provided by Dirk Beyer (Ludwig Maximilian University) on software verification and Cesare Tinelli (University of Iowa) on SMT and its applications. On behalf of the ETAPS 2019 attendants, I thank all the speakers for their inspiring and interesting talks!

ETAPS 2019 took place in Prague, Czech Republic, and was organized by Charles University. Charles University was founded in 1348 and was the first university in Central Europe. It currently hosts more than 50,000 students. ETAPS 2019 was further supported by the following associations and societies: ETAPS e.V., EATCS (European Association for Theoretical Computer Science), EAPLS (European Association for Programming Languages and Systems), and EASST (European Association of Software Science and Technology). The local organization team consisted of Jan Vitek and Jan Kofron (general chairs), Barbora Buhnova, Milan Ceska, Ryan Culpepper, Vojtech Horky, Paley Li, Petr Maj, Artem Pelenitsyn, and David Safranek.

The ETAPS SC consists of an Executive Board, and representatives of the individual ETAPS conferences, as well as representatives of EATCS, EAPLS, and EASST. The Executive Board consists of Gilles Barthe (Madrid), Holger Hermanns (Saarbrücken), Joost-Pieter Katoen (chair, Aachen and Twente), Gerald Lüttgen (Bamberg), Vladimiro Sassone (Southampton), Tarmo Uustalu (Reykjavik and Tallinn), and Lenore Zuck (Chicago). Other members of the SC are: Wil van der Aalst (Aachen), Dirk Beyer (Munich), Mikolaj Bojanczyk (Warsaw), Armin Biere (Linz), Luis Caires (Lisbon), Jordi Cabot (Barcelona), Jean Goubault-Larrecq (Cachan), Jurriaan Hage (Utrecht), Rainer Hähnle (Darmstadt), Reiko Heckel (Leicester), Panagiotis Katsaros (Thessaloniki), Barbara König (Duisburg), Kim G. Larsen (Aalborg), Matteo Maffei (Vienna), Tiziana Margaria (Limerick), Peter Müller (Zurich), Flemming Nielson (Copenhagen), Catuscia Palamidessi (Palaiseau), Dave Parker (Birmingham), Andrew M. Pitts (Cambridge), Dave Sands (Gothenburg), Don Sannella (Edinburgh), Alex Simpson (Ljubljana), Gabriele Taentzer (Marburg), Peter Thiemann (Freiburg), Jan Vitek (Prague), Tomas Vojnar (Brno), Heike Wehrheim (Paderborn), Anton Wijs (Eindhoven), and Lijun Zhang (Beijing).

I would like to take this opportunity to thank all speakers, attendants, organizers of the satellite workshops, and Springer for their support. I hope you all enjoy the proceedings of ETAPS 2019. Finally, a big thanks to Jan and Jan and their local organization team for all their enormous efforts enabling a fantastic ETAPS in Prague!

February 2019

Joost-Pieter Katoen
ETAPS SC Chair
ETAPS e.V. President

Preface

This volume contains the papers presented at the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019) held during April 9–11, 2019, in Prague. FASE 2019 was organized as part of the annual European Joint Conferences on Theory and Practice of Software (ETAPS 2019). ETAPS is the most important and visible annual European event related to software sciences.

As usual, the papers submitted to FASE focus on the foundations on which software engineering is built. The papers submitted covered topics such as software engineering, requirements engineering, software architectures, specification, software quality, validation, verification of functional and non-functional properties, model-driven development and model transformation, model transformations, software processes, and software evolution.

We received 94 abstract submissions of which 74 were turned into full submissions (63 research papers, five tool papers, and six demo papers). We had submissions from the following countries (sorted based on the number of submissions): Germany, France, Canada, Estonia, USA, Argentina, UK, Norway, Spain, Brazil, China, South Korea, Australia, Czechia, Austria, Denmark, Italy, Japan, the Netherlands, Pakistan, South Africa, Tunisia, India, Poland, Portugal, Romania, Turkey, Belgium, Colombia, Macedonia, Malta, Sweden, and Ukraine.

Of the 74 submitted papers, 24 papers were accepted after reviewing and discussions among the Program Committee (PC) members (20 research papers, two tool papers, and two demo papers). This corresponds to a 32% acceptance rate. Beside the 30 PC members, there were 100 external reviewers. For the fourth time, FASE used a double-blind reviewing process. Overall the reviewing process was smooth and it was possible to have consensus on all decisions. We thank the PC members and reviewers for doing a great job!

Apart from thanking the authors, we also thank Marsha Chechik (University of Toronto) for contributing a paper based on her plenary ETAPS 2019 invited talk, which is also included in these proceedings. The title of Marsha's talk was "Software Assurance in an Uncertain World." She discussed the problem that software systems are deeply rooted in uncertainty since most complex open-world functionality is either not completely specifiable or it is not cost-effective to do so. Moreover, these systems are placed in an uncertain ever-evolving environment.

This volume shows that, despite the rapid progress in software engineering, there are still many open problems. These problems are important for the way we do business, the way we govern, and the way we socialize. We depend on complex software artifacts, yet we still need to fully understand how to best develop and maintain them. The papers in this volume help to progress the state of the art and hopefully inspire and influence future work.

We thank the ETAPS 2019 organizers, in particular, Jan Kofron and Jan Vitek (general chairs), Barbora Buhnova (publicity chair), Vojtech Horkey and Arten

Pelnisyn (web chairs), and David Safranek (publications chair). We also thank Joost-Pieter Katoen, the ETAPS SC chair, for managing the whole process, and Gabriele Taentzer, the FASE SC chair, for swift feedback on several questions.

We hope that you will enjoy reading the volume.

February 2019

Wil van der Aalst
Reiner Hähnle

Organization

Program Committee

Christel Baier	TU Dresden, Germany
Stefano Berardi	University of Turin, Italy
Mario Bravetti	University of Bologna, Italy
Jordi Cabot	Open University of Catalonia, Spain
Ana Cavalcanti	University of York, UK
Marsha Chechik	University of Toronto, Canada
Ferruccio Damiani	University of Turin, Italy
Ewen Denney	NASA Ames Research Center, USA
Dilian Gurov	KTH Royal Institute of Technology, Sweden
Ludovic Henrio	CNRS, France
Reiner Hähnle	TU Darmstadt, Germany
Gerti Kappel	Vienna University of Technology, Austria
Ekkart Kindler	Technical University of Denmark, Denmark
Martin Leucker	University of Lübeck, Germany
Jun Pang	University of Luxembourg, Luxembourg
André Platzer	Carnegie Mellon University, USA
Bernhard Rumpe	RWTH Aachen University, Germany
Alessandra Russo	Imperial College London, UK
Rick Salay	University of Toronto, Canada
Ina Schaefer	Technische Universität Braunschweig, Germany
Andy Schürr	TU Darmstadt, Germany
Perdita Stevens	The University of Edinburgh, UK
Mariëlle Stoelinga	University of Twente, The Netherlands
Jun Sun	Singapore University of Technology and Design, Singapore
Gabriele Taentzer	Philipps-Universität Marburg, The Netherlands
Silvia Lizeth Tapia Tarifa	University of Oslo, Norway
Maurice H. Ter Beek	ISTI-CNR, Pisa, Italy
Wil M. P. van der Aalst	RWTH Aachen University, Germany
Heike Wehrheim	Paderborn University, Germany
Yingfei Xiong	Peking University, China

Additional Reviewers

Aspinall, David
Bafrani, Mahsa
Baxter, James
Berti, Alessandro
Bettini, Lorenzo
Bill, Robert
Bozzano, Marco
Bubel, Richard
Canovas Izquierdo, Javier Luis
Cerone, Andrea
Chen, Yifan
Ciancia, Vincenzo
Cordwell, Katherine
Dalibor, Manuela
Dashevskyi, Stanislav
Din, Crystal Chang
Drave, Imke Helene
Ed-Douibi, Hamza
Escobar, Santiago
Ferrari, Alessio
Fritsche, Lars
Fulton, Nathan
Gadyatskaya, Olga
Gario, Marco
Gerhold, Marcus
Gerking, Christopher
Giannini, Paola
Girault, Alain
Guanciale, Roberto
Gómez, Abel
Habermehl, Peter
Haglund, Jonas
Henderson, Robbie
Herda, Mihai
Hillemacher, Steffen
Johnsen, Einar Broch
Kamburjan, Eduard
Kharraz, Karam
Knüppel, Alexander
Kosiol, Jens
König, Jürgen
Lange, Felix Dino
Laurent, Jonathan
Leroy, Dorian
Lidström, Christian
Lienhardt, Michael
Lindner, Andreas
Lischke, Sabrina
Lochau, Malte
Lu, Sirui
Luthmann, Lars
Martínez, Salvador
Mauro, Jacopo
Mazzanti, Franco
Meijer, Jeroen
Mereuta, Radu
Michael, Judith
Mitsch, Stefan
Miyazawa, Alvaro
Mover, Sergio
Najafzadeh, Mahsa
Nassar, Nebras
Netz, Lukas
Oortwijn, Wytse
Palmskog, Karl
Paolini, Luca
Papadakis, Michail
Papadakis, Mike
Pedro, Andre
Petrocchi, Marinella
Pozzato, Gian Luca
Raco, Deni
Ren, Luyao
Ribeiro, Pedro
Ruijters, Enno
Ruland, Sebastian
Runge, Tobias
Schivo, Stefano
Schlatte, Rudolf
Schlie, Alexander
Schmalzing, David
Schmitz, Malte
Sharma, Arnab
Shumeiko, Igor
Sogokon, Andrew
Spagnolo, Giorgio Oronzo
Sproston, Jeremy
Steffen, Martin
Thoma, Daniel
Thüm, Thomas
Toews, Manuel
Tomaszek, Stefan
Tveito, Lars
Wally, Bernhard
Wang, Bo
Wang, Guancheng
Zacchiroli, Stefano
Zawadzki, Erik
Zhang, Yuhao
Zhu, Qihao

Contents

FASE Invited Talk

Software Assurance in an Uncertain World

Marsha Chechik(✉), Rick Salay, Torin Viger, Sahar Kokaly, and Mona Rahimi

University of Toronto, Toronto, Canada
chechik@cs.toronto.edu

Abstract. From financial services platforms to social networks to vehicle control, software has come to mediate many activities of daily life. Governing bodies and standards organizations have responded to this trend by creating regulations and standards to address issues such as safety, security and privacy. In this environment, the compliance of software development to standards and regulations has emerged as a key requirement. Compliance claims and arguments are often captured in assurance cases, with linked evidence of compliance. Evidence can come from testcases, verification proofs, human judgment, or a combination of these. That is, experts try to build (safety-critical) systems carefully according to well justified methods and articulate these justifications in an assurance case that is ultimately judged by a human. Yet software is deeply rooted in uncertainty; most complex open-world functionality (e.g., perception of the state of the world by a self-driving vehicle), is either not completely specifiable or it is not cost-effective to do so; software systems are often to be placed into uncertain environments, and there can be uncertainties that need to be We argue that the role of assurance cases is to be the grand unifier for software development, focusing on capturing and managing uncertainty. We discuss three approaches for arguing about safety and security of software under uncertainty, in the absence of fully sound and complete methods: assurance argument rigor, semantic evidence composition and applicability to new kinds of systems, specifically those relying on ML.

1 Introduction

From financial services platforms to social networks to vehicle control, software has come to mediate many activities of daily life. Governing bodies and standards organizations have responded to this trend by creating regulations and standards to address issues such as safety, security and privacy. In this environment, the compliance of software development to standards and regulations has emerged as a key requirement.

Development of safety-critical systems begins with *hazard analysis*, aimed to identify possible causes of harm. It uses severity, probability and controllability of a hazard's occurrence to assign the Safety Integrity Levels (in the automotive industry, these are referred to as ASILs [35]) – the higher the ASIL level,

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 3–21, 2019.
https://doi.org/10.1007/978-3-030-16722-6_1

the more rigor is expected to be put into identifying and mitigating the hazard. Mitigating hazards therefore becomes the main requirement of the system, with system safety requirements being directly linked to the hazards. These requirements are then refined along the LHS of the V until individual modules and their implementation can be built. The RHS includes appropriate testing and validation, used as supporting evidence in developing an argument that the system adequately handles its hazards, with the expectation that the higher the ASIL level, the stronger the required justification of safety is.

Assurance claims and arguments are often captured by *assurance cases*, with linked evidence supporting it. Evidence can come from testcases, verification proofs, human judgment, or a combination of these. Assurance cases organize information allowing argument unfolding in a comprehensive way and ultimately allowing safety engineers to determine whether they trust that the system was adequately designed to avoid systematic faults (before delivery) and adequately detect and react to failures at runtime [35].

Yet software is deeply rooted in uncertainty; most complex open-world functionality (e.g., perception of the state of the world by a self-driving vehicle), is either not completely specifiableor it is not cost-effective to do so [12]. Software systems are often to be placed into uncertain environments [48], and there can be uncertainties that need to be considered at the design phase [20]. Thus, we believe that the role of assurance cases is to *explicitly capture and manage uncertainty coming from different sources, assess it and ultimately reduce it to an acceptable level, either with respect to a standard, company processes, or assessor judgment*. The various software development steps are currently not well integrated, and uncertainty is not expressed or managed explicitly in a uniform manner. Our claim in this paper is that *an assurance case is the unifier among the different software development steps, and can be used to make uncertainties explicit, which also makes them manageable. This provides a well-founded basis for modeling confidence about satisfaction of a critical system quality (security, safety, etc.) in an assurance case, making assurance cases play a crucial role in software development*. Specifically, we enumerate sources of uncertainty in software development. We also argue that organizing software development and analysis activities around the assurance case as a *living document* allows all parts of the software development to explicitly articulate uncertainty, steps taken to manage it, and the degree of confidence that artifacts acting as evidence have been performed correctly. This information can then help potential assessors in checking that the development outcome adequately satisfies the software desired quality (e.g., safety).

The area of system dependability has produced a significant body of work describing how to model assurance cases (e.g., [4,5,14,38]), and how to assess reviewer's confidence in the argument being made (e.g., [16,31,45,59,60]). There is also early work on assessing the impact of change on the assurance argument when the system undergoes change [39]. A recent survey [43] provides a comprehensive list of assurance case tools developed over the past 20 years and an analysis of their functionalities including support for assurance case creation,

assessment and maintenance. We believe that the road to truly making assurance cases the grand unifier for software development for complex high-assurance systems has many challenges. One is to be able to successfully argue about safety and security of software under uncertainty, without fully sound and complete methods. For that, we believe that *assurance arguments must be rigorous* and that we need to properly understand how to perform *evidence composition* for traditional systems, but also for *new kinds of systems*, specifically those relying on ML. We discuss these issues below.

Rigor. To be validated or reused, assurance case structures must be as rigorous as possible [51]. Of course, assurance arguments ultimately depend on human judgment (with some facts treated as "obvious" and "generally acceptable"), but the structure of the argument should be fully formal so as to allow to assess its completeness. Bandur and McDermid called this approach "formal modulo engineering expertise" [1].

Evidence Composition. We need to effectively combine the top-down process of uncertainty reduction with the bottom-up process of composing evidence, specifically, evidence obtained from applying testing and verification techniques.

Applicability to "new" kinds of systems. We believe that our view – rigorous, uncertainty-reduction focused and evidence composing – is directly applicable to systems developed using machine learning, e.g., self-driving cars.

This paper is organized as follows: In Sect. 2, we briefly describe syntax of assurance cases. In Sect. 3, we outline possible sources of uncertainty encountered as part of system development. In Sect. 4, we describe the benefits of a rigorous language for assurance cases by way of example. In Sect. 5, we describe, again by way of example, a possible method of composing evidence. In Sect. 6, we develop a high-level assurance case for a pedestrian detection subsystem. We conclude in Sect. 7 with a discussion of possible challenges and opportunities.

2 Background on Assurance Case Modeling Notation

The most commonly used representation for safety cases is the graphical Goal Structuring Notation (GSN) [30], which is intended to support the assurance of critical properties of systems (including safety). GSN is comprised of six core elements – see Fig. 1. Arguments in GSN are typically organized into a tree of the core elements shown in Fig. 1[1]. The root is the overall goal to be satisfied by the system, and it is gradually decomposed (possibly via strategies) into sub-goals and finally into solutions, which are the leaves of the safety case. Connections between goals, strategies and solutions represent *supported-by* relations, which indicate inferential or evidential relationships between elements. Goals and strategies may be optionally associated with some contexts, assumptions and/or justifications by means of *in-context-of* relations, which declare a contextual relationship between the connected elements.

[1] In this paper, we use both diamond and triangle shapes interchangeably to depict an "undeveloped" element.

Fig. 1. Core GSN elements from [30].

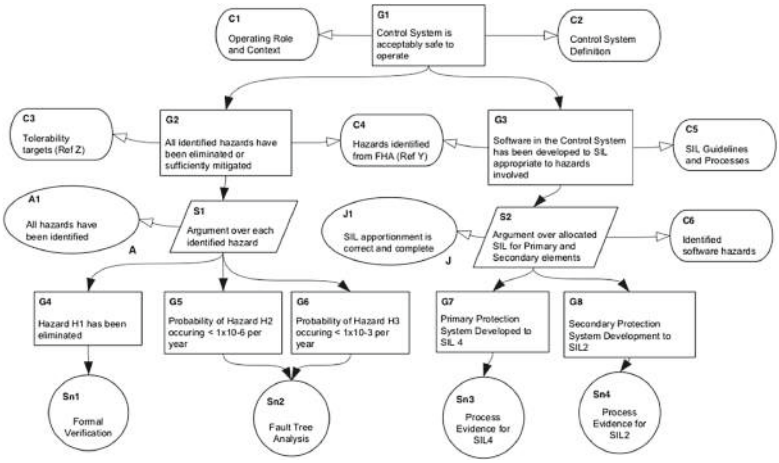


Fig. 2. Example safety case in GSN (from [30]).

For example, consider the safety case in Fig. 2. The overall goal **G1** is that the "Control System is acceptably safe to operate" given its role, context and definition, and it is decomposed into two sub-goals: **G2**, for eliminating and mitigating all identified hazards, and **G3**, for ensuring that the system software is developed to an appropriate ASIL. Assuming that all hazards have been identified, **G2** can in turn be decomposed into three sub-goals by considering each hazard separately (**S1**), and each separate hazard is shown to be satisfied using evidence from formal verification (**Sn1**) or fault tree analysis (**Sn2**). Similarly, under some specific context and justification, **G3** can be decomposed into two sub-goals, each of which is shown to be satisfied by the associated evidence.

3 Sources of Uncertainty in Software Development

In this section, we briefly survey uncertainty in software development, broadly split into the categories of uncertainties about the specifications, about the environment, about the system itself, and about the argument of its safety. For each

part, we aim to address how building an assurance case is related to understanding and mitigating such uncertainties.

Uncertainty in Specifications. Software specifications tend to suffer from incompleteness, inconsistency and ambiguity [42,46]. Specification uncertainty stems from a misunderstanding or an incomplete understanding of how the system is supposed to function in early phases of development; e.g., miscommunication and inability of stakeholders to transfer knowledge due to differing concepts and vocabularies [2,13]; unknown values for sets of known events (a.k.a. the *known unknowns*); and the unknown and unidentifiable events (a.k.a. the *unknown unknowns*) [57].

Recently, machine-learning approaches for interactively learning the software specifications have become popular; we discuss one such example, of pedestrian detection, in Sect. 6. Other mitigations of specification uncertainties, suggested by various standards and research, are identification of edge cases [36], hazard and obstacle analysis [55] to help identify unknown unknowns [35], step-wise refinement to handle partiality in specifications, ontology- [9] and information retrieval-driven requirements engineering approaches [21], as well as generally building arguments about addressing specification uncertainties.

Environmental Uncertainty. The system's environment can refer to adjacent agents interacting with the system, a human operator using the system, or physical conditions of the environment. Sources of environmental uncertainties have been thoroughly investigated [19,48]. One source originates from unpredictable and changing properties of the environment, e.g., assumptions about actions of other vehicles in the autonomous vehicle domain or assuming that a plane is on the runway if its wheels are turning. Another uncertainty source is input errors from broken sensors, missing, noisy and inaccurate input data, imprecise measurements, or disruptive control signals from adjacent systems. Yet another source might be when changes in the environment affect the specification. For example, consider a robotic arm that moves with the expected precision but the target has moved from its estimated position.

A number of techniques have been developed to mitigate environmental uncertainties, e.g., runtime monitoring systems such as RESIST [10], or machine-learning approaches such as FUSION [18] which self-tune the adaptive behavior of systems to unanticipated changes in the environment. More broadly, environmental uncertainties are mitigated by a careful requirements engineering process, by principled system design and, in assurance cases, by an argument that they had been adequately identified and adequately handled.

System Uncertainties. One important source of uncertainty is faced by developers who do not have sufficient information to make decisions about their system during development. For example, a developer may have insufficient information to choose a particular implementation platform. In [19,48], this source of uncertainty is referred to as *design-time uncertainty*, and some approaches to handling it are offered in [20]. Decisions made while resolving such uncertainties are crucial to put into an assurance argument, to capture the context, i.e.,

a particular platform is selected because of its performance, at the expense of memory requirements.

Another uncertainty refers to correctness of the implementation [7]. This uncertainty lays in the V&V procedure and is caused by whether the implementation of the tool can be trusted, whether the tool is used appropriately (that is, its assumptions are satisfied), and in general, whether a particular verification technique is the right one for verifying the fulfillment of the system requirements [15]. We address some of these uncertainties in Sect. 5.

Argument Uncertainty. The use of safety arguments to demonstrate safety of software-intensive systems raises questions such as the extent to which these arguments can be trusted. That is, how confident are we that a verified, validated software is actually safe? How much evidence and how thorough of an argument do we require for that?

To assess uncertainties which may affect the system's safety, researchers have proposed techniques to estimate confidence in structured assurance cases, either through qualitative or quantitative approaches [27,44]. The majority of these are based on the Dempster-Shafer Theory [31,60], Josang's Opinion Triangle [17], Bayesian Belief Networks (BNNs) [16,61], Evidential Reasoning (ER) [45] and weighted averages [59]. The approaches which use BBNs treat safety goals as nodes in the network and try to compute their conditional probability based on given probabilities for the leaf nodes of the network. Dempster-Shafer Theory is similar to BBNs but is based on the *belief function* and its *plausibility* which is used to combine separate pieces of information to calculate the probability. The ER approach [45] allows the assessors to provide individual judgments concerning the trustworthiness and appropriateness of the evidence, building a separate argument from the assurance case.

These approaches focus on assigning and propagating confidence measures but do not specifically address uncertainty in the argument. They also focus on aggregating evidence coming from multiple sources but treat it as a "black box", instead of how a piece of evidence from one source might compose with another. We look at these questions in Sects. 4 and 5, respectively.

4 Formality in Assurance Cases

As discussed in Sect. 1, we believe that the ultimate goal of an assurance case is to explicitly capture and manage uncertainty, and ultimately reduce it to an acceptable level. Even informal arguments improve safety, e.g., by making people decompose the top level goal case-wise, and examine the decomposed parts critically. But the decomposed cases tend to have an ad hoc structure dictated by experience and preference, with under-explored completeness claims, giving both developers and regulators a false sense of confidence, no matter how confidence is measured, since they feel that their reasoning is rigorous even though it is not [58]. Moreover, as assurance cases are produced and judged by humans, they are typically based on *inductive arguments*. Such arguments are susceptible to fallacies (e.g., arguing through circular reasoning, using justification based

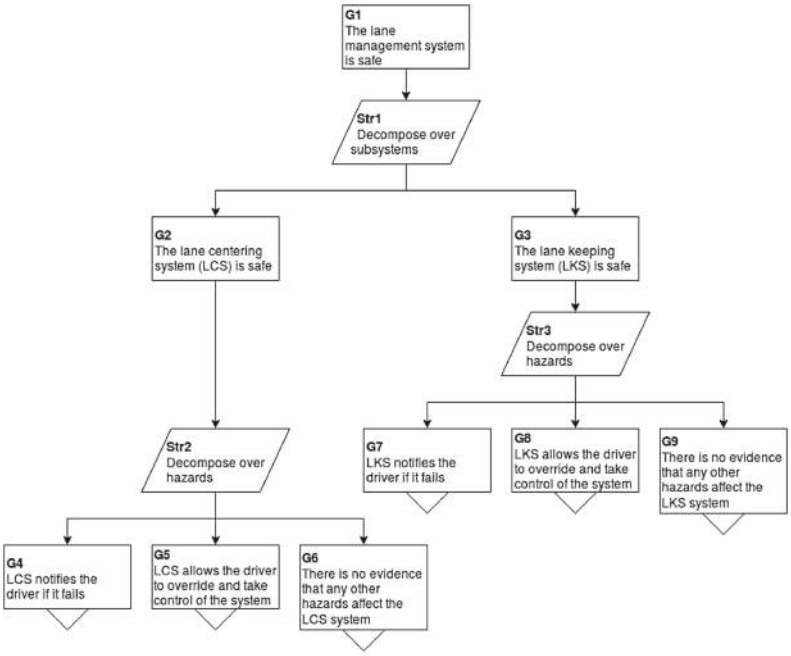


Fig. 3. A fragment of the Lane Management (LMS) Safety case.

on false dichotomies), and evaluations by different reviewers may lead to the discovery of different fallacies [28].

There have been several attempts to improve credibility of an argument by making the argument structure more formal. [25] introduces the notion of confidence maps as an explicit way of reasoning about sources of doubt in an argument, and proposes justifying confidence in assurance arguments through *eliminative induction* (i.e., an argument by eliminating sources of doubt). [29] highlights the need to model both evidential and argumentation uncertainties when evaluating assurance arguments, and considers applications of the formally evaluatable extension of Toulmin's argument style proposed by [56]. [11] details VAA – a method for assessing assurance arguments based on Dempster-Shafer theory. [51] is a proponent of completely deductive reasoning, narrowing the scope of the argument so that it can be formalized and potentially formally checked, using automated theorem provers, arguing that this would give a modular framework for assessing (and, we presume, reusing) assurance cases. [1] relaxes Rushby's position a bit, aiming instead at formal assurance argumentation "modulo engineering expertise", and proof obligations about consistency of arguments remain valid even for not fully formal assurance arguments. To this end, they provided a specific formalization of goal validity given validity of subgoals and contexts/context assumptions, resulting in such rules as

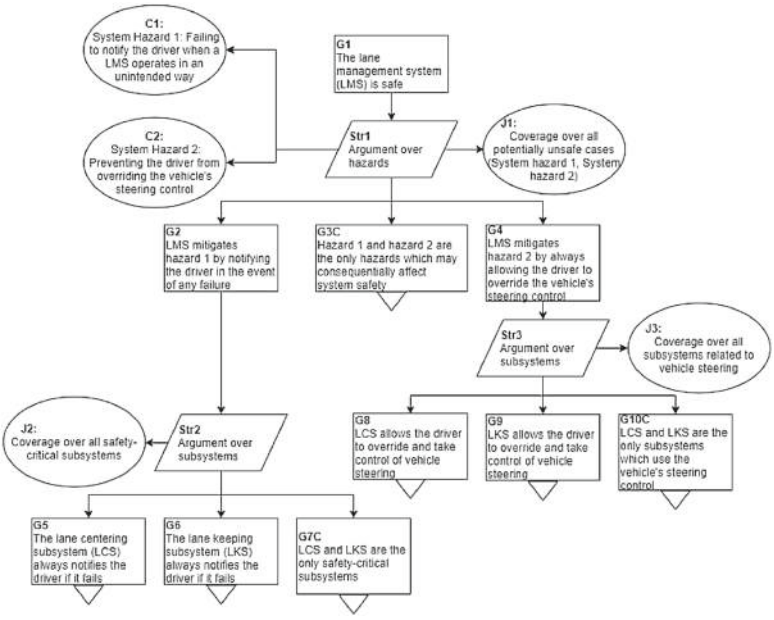


Fig. 4. An alternative representation of the same LMS fragment.

"assumptions on any given element must not be contradictory nor contradict the context assumed for that goal" [1].

Our Position. We believe that a degree of formality in assurance cases can go a long way not only towards establishing its validity, identifying and framing implicit uncertainties and avoiding fallacies, but also supporting assurance case modularity, refactoring and reuse. We illustrate this position on an example.

Example. Consider two partially developed assurance cases that argue that the lane management system (LMS) of a vehicle is safe (Figs. 3 and 4). The top-level safety goal **G1** in Fig. 3 is first decomposed by the strategy **Str1** into a set of subgoals which assert the safety of the LMS subsystems. An assessor can only trust that goals **G2** and **G3** imply **G1** by making an implicit assumption that the system safety is completely determined by the safety of its individual subsystems. Neither the need for this assumption nor the credibility of the assumption itself are made explicit in the assurance case, which weakens the argument and complicates the assessment process. The argument is further weakened by the absence of a completeness claim that all subsystems have been covered by this decomposition.

Strategies **Str2** and **Str3** in Fig. 3 decompose the safety claims about each subsystem into arguments over the relevant hazards. Yet the hazards themselves are never explicitly stated in the assurance case, making the direct relevance of each decomposed goal to its corresponding parent goal, and thus to the argument as a whole, unclear. While goals **G6** and **G9** attempt to provide completeness

claims for their respective decompositions, they do so by citing lack of negative evidence without describing efforts to uncover such evidence. This justification is fallacious and can be categorized as "an argument from ignorance" [28].

Now consider the assurance case in Fig. 4 which presents a variant of the argument in Fig. 3, refined with context nodes, justification nodes and completeness claims. The top-level goal **G1** is decomposed into a set of subgoals asserting that particular hazards have been mitigated, as well as a completeness claim **G3C** stating that hazards **H1** and **H2** are the only ones that may be prevalent enough to defeat claim **G1**. Context nodes **C1** and **C2** define the hazards themselves, which clarifies the relevance of each hazard-mitigating goal. The node **J1** provides a justification for the validity of **Str1** by framing the decomposition as a proof by (exhaustive) cases. That is, **Str1** is justified by the statement that if **H1** and **H2** are the only hazards that could potentially make the system unsafe, then the system is safe if **H1** and **H2** have been adequately mitigated. This rigorous argument can be represented by the logical expression $\mathbf{G3C} \implies ((\mathbf{G2} \wedge \mathbf{G4}) \implies \mathbf{G1})$, and if completeness holds then **G2** and **G4** are sufficient to show **G1**. We now have a rigorous argument step that our confidence in **G1** is a direct consequence of confidence in its decomposed goals **G2**, **G3C** and **G4**, even though there may still be uncertainty in the evidential evaluation of **G2**, **G3C** and **G4**. That is, uncertainty has been made explicit and can be reasoned about at the evidential level. By removing argumentation uncertainty and explicating implicit assumptions, we get a more comprehensive framework for assurance case evaluation, where the relation between all reasoning steps is formally clear. Note that if the justification provides an inference rule, then the argument becomes deductive. Otherwise, it is weaker (the justification node can be used to quantify just *how* weaker) but still rigorous.

While the completeness claim **G3C** in Fig. 4 may be directly supported by evidence, the goals **G2** and **G4** are further decomposed by the strategies **Str2** and **Str3**, respectively, which represent decompositions over subsystems. These strategies are structured similarly to **Str1**, and can be expressed by the logical expressions $\mathbf{G7C} \implies ((\mathbf{G5} \wedge \mathbf{G6}) \implies \mathbf{G2})$ and $\mathbf{G10C} \implies ((\mathbf{G8} \wedge \mathbf{G9}) \implies \mathbf{G4})$, respectively. In Fig. 3, a decomposition by subsystems was applied directly to the top-level safety goal which necessitated a completeness claim that the safety of all individual subsystems implied safety of the entire system. Instead, the argument in Fig. 4 only needs to show that the set of subsystems in each decomposition is complete w.r.t. a particular hazard, which may be a more feasible claim to argue. This ability to transform an argument into a more easily justifiable form is another benefit of arguing via rigorous reasoning steps.

5 Combining Evidence

Evidence for assurance cases can come from a variety of sources: results from different testing and verification techniques, human judgment, or their combination. Multiple testing and verification techniques may be used to make the evidence more complete. A verification technique *complements* another if it is able

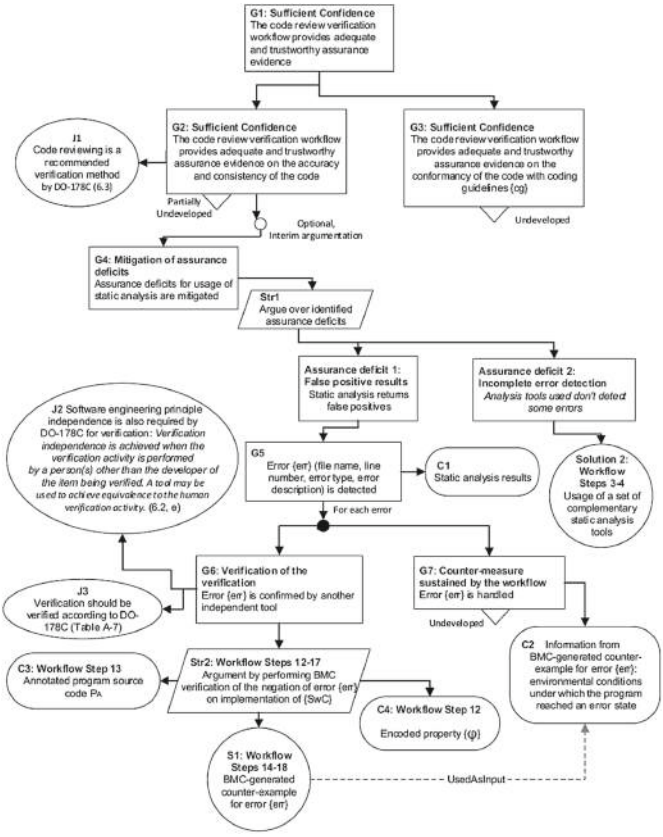


Fig. 5. Confidence argument for code review workflow (from [6]).

to verify types of requirements which cannot be verified by the other technique. For example, results of verification of properties via a bounded model checker (BMC) are complemented by additional test cases [8]. A verification technique *supports* another if it is used to detect faults in the other's verification results, thus providing backing evidence [33]. For example, a model checking technique may support a static analysis technique by verifying the faults detected [6]. Note that these approaches are principally different from just aggregating evidence treating it as a blackbox!

Habli and Kelly [32] and Denney and Pai [15] present safety case patterns for the use of formal method results for certification. Bennion et al. [3] present a safety case for arguing the compliance of a particular model checker, namely, the Simulink Design Verifier for DO-178C. Gallina and Andrews [23] argue about adequacy of a model-based testing process, and Carlan et al. [7] provide a safety pattern for choosing and composing verification techniques based on how they

contribute to the identification or mitigation of systematic faults known to affect system safety.

Our Position. We, as a community, need to figure out the precise conditions under which particular testing and verification techniques "work" (e.g., modeling floating-point numbers as reals, making a small model hypothesis to justify sufficiency of a particular loop unrolling, etc.), and how they are intended to be composed in order to reduce uncertainty about whether software satisfies its specification. We illustrate a particular composition here.

Example. In this example, taken from [6], a model checker supports static analysis tools (that produce false negatives) by verifying the detected faults [6]. The assurance case is based on a workflow (not shown here) where an initial review report is constructed, by running static analysis tools and possibly peer code reviews. Then the program is annotated with the negation of each potential erroneous behavior as a desirable property for the program, and given to a model-checker. If the model-checker is able to verify the property, it is removed from the initial review report and not considered as an error. If the model-checker finds a violation, the alleged error is confirmed. In this case, a weakest-precondition generation mechanism is applied to find out the environmental conditions (external parameters that are not under the control of the program) under which the program shows the erroneous behavior. These conditions and the error trace are then added to the error description.

The paper [6] presents both the assurance case and the confidence argument for the code review workflow. We reproduce only the latter here (see Fig. 5), focusing on reducing uncertainty about the accuracy and consistency of the code property (goal **G2**). False positives generated by static analysis are mitigated using BMC – a method with a completely different verification rationale, thus implementing the safety engineering principle of independence (**J2**). Strategy (**Str2**) explains how errors can be confirmed or dismissed using BMC (goal **G6**). The additional information given by BMC can be used for the mitigation of the error (**C2**).

This approach takes good steps towards mitigating particular assurance deficits using a composition of verification techniques but leaves open several problems: how to ensure that BMC runs under the same environmental conditions as the static analysis tools? how deeply should the loops be unrolled? what to do with cases when the model-checker runs out of resources without giving a conclusive answer? and in general, what are the conditions under which it is safe to trust the "yes" answers of the model-checker.

6 Assurance Cases for ML Systems

Academia and industry are actively building systems using AI and machine learning, including a rapid push for ML in safety-critical domains such as medical devices and self-driving cars. For their successful adoption in society, we need to ensure that they are trustworthy, including obtaining confidence in their behavior and robustness.

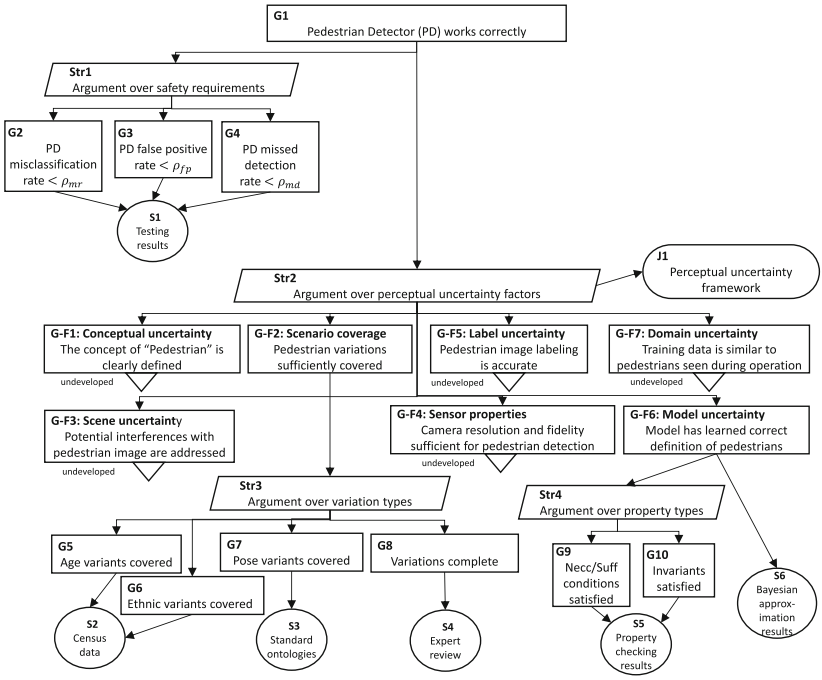


Fig. 6. A partially developed GSN safety case of pedestrian detector example.

Significant strides have already been made in this space, from extending mature testing and verification techniques to reasoning about neural networks [24,37,47,54] for properties such as safety, robustness and adequate handling of adversarial examples [26,34]. There is active work in designing systems that balance learning under uncertainty and acting safely, e.g., [52] as well as the broad notion of fairness and explainability in AI, e.g., [49].

Our Position. We believe that assurance cases remain a unifying view for ML-based systems just as much as for more conventional systems, allowing us to understand how the individual approaches fit into the overall goal of assuring safety and reliability and where there are gaps.

Example. We illustrate this idea with an example of a simple pedestrian detector (PD) component used as part of an autonomous driving system. The functions that PD supports consist of detection of objects in the environment ahead of the vehicle, classification of an object as a *pedestrian* or *other*, and localization of the position and extent of the pedestrian (indicated by bounding box). We assume that PD is implemented as a convolutional deep neural network with various stages to perform feature extraction, proposing regions containing objects and classification of the proposed objects. This is a typical approach for two-stage object detectors (e.g., see [50]).

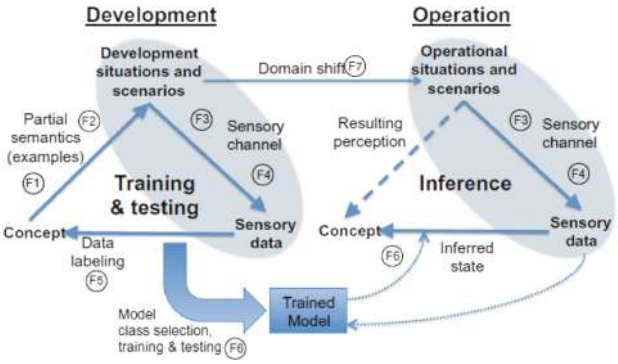


Fig. 7. A framework for factors affecting perceptual uncertainty (source: [12]).

As part of a safety critical system, PD contributes to the satisfaction of a top-level safety goal requiring that the vehicle always maintain a safe distance from all pedestrians. Specific safety requirements for PD can be derived from this goal, such as (RQ1) PD misclassification rate (i.e., classifying a pedestrian as "other") must be less than ρ_{mc}, (RQ2) PD false positive rate (i.e., classifying any non-pedestrian object or non-object as "pedestrian") must be less than ρ_{fp}, and (RQ3) PD missed detection rate (i.e., missing the presence of pedestrian) must be less than ρ_{md}. Here, the parameters ρ_{mc}, ρ_{fp} and ρ_{md} must be derived in conjunction with the control system that uses the output from PD to plan the vehicle trajectory.

The partially developed safety case for PD is shown in Fig. 6. The three safety requirements are addressed via the strategy **Str1** and, as expected, testing results are given as evidence of their satisfaction. However, since testing can only provide limited assurance about the behaviour of PD in operation, we use an additional strategy, **Str2**, to argue that a rigorous method was followed to develop PD. Specifically, we follow the framework of [12] for identifying the factors that lead to uncertainty in ML-based perceptual software such as PD.

The framework is defined at a high level in Fig. 7. The left "perception triangle" shows how the perceptual concept (in the case of PD, the concept "pedestrian") can occur in various scenarios in the world, how it is detected using sensors such as cameras, and how this can be used to collect and label examples in order to train an ML component to learn the concept. The perception triangle on the right is similar but shows how the trained ML component can be used during the system operation to make inferences (e.g., perform the pedestrian detection). The framework identifies seven factors that could contribute to uncertainty in the behaviour of the perceptual component. A safety case demonstrating a rigorous development process should provide evidence that each factor has been addressed.

In Fig. 6, strategy **Str2** uses the framework to argue that the seven factors are adequately addressed for PD. We illustrate development of two of these factors

here. Scenario coverage (Goal **G-F2**) deals with the fact that the training data must represent the concept in a sufficient variety of scenarios in which it could occur in order for the training to be effective. The argument here first decomposes this goal into different types of variation (**Str3**) and provides appropriate evidence for each. The adequacy of age and ethnicity variation in the data set is supported by census data (**S2**) about the range of these dimensions of variation in the population. The variation in the pedestrian pose (i.e., standing, leaning, crouching, etc.) is supplied by a standard ontology of human postures (**S3**). Finally, evidence that the types are adequate to provide sufficient coverage of variation (completeness) is provided by an expert review (**S4**).

Another contributing factor developed in Fig. 6 is model uncertainty (Goal **G-F6**). Since there is only finite training data, there can be many possible models that are equally consistent with the training data, and the training process could produce any one of them, i.e., there is residual uncertainty whether the produced model is in fact correct. The presence of model uncertainty means that while the trained model may perform well on inputs similar to the training data, there is no guarantee that it will produce the right output for other inputs. Some evidence of good behaviour here can be gathered if there are known properties that partially characterize the concept and can be checked. For example, a reasonable necessary condition for PD is that the object being classified as a pedestrian should be less than 9 ft tall. Another useful property type is an invariant, e.g., a rotated pedestrian image is still a pedestrian. Tools for property checking of neural networks (e.g., [37]) can provide this kind of evidence (**S5**). Another way to deal with model uncertainty is to estimate it directly. Bayesian deep learning approaches [22] can do this by measuring the degree of disagreement between multiple trained models that are equally consistent with the training data. The more the models are in agreement are about how to classify a new input, the less model uncertainty is present and the more confident one can be in the prediction. Using this approach on a test data set can provide evidence (**S6**) about the degree of model uncertainty in the model. This approach can also be used during the operation to generate a confidence score in each prediction and use a fault tolerance strategy that takes a conservative action when the confidence falls below a threshold.

7 Summary and Future Outlook

In this paper, we tried to argue that an assurance case view on establishing system correctness provides a way to unify different components of the software development process and to explicitly manage uncertainty. Furthermore, although our examples came from the world of safety-critical automotive systems, the assurance case view is broadly applicable to a variety of systems, not just those in the safety-critical domain and includes those constructed by non-traditional means such as ML. This view is especially relevant to much of the research activity being conducted by the ETAPS community since it allows, in principle, to understand how each method contributes to the overall problem of system assurance.

Most traditional assurance methods aim to build an informal argument, ultimately judged by a human. However, while these are useful for showing compliance to standards and are relatively easy to construct and read, such arguments may not be rigorous, missing essential properties such as completeness, independence, relevance, or a clear statement of assumptions [51]. As a result, fallacies in existing assurance cases are present in abundance [28]. To address this weakness, we argued that building assurance cases should adhere to systematic principles that ensure rigor. Of course, not all arguments can be fully deductive since relevance and admissibility of evidence is often based on human judgment. Yet, an explicit modeling and management of uncertainty in evidence, specifications and, assumptions as well as the clear justification of each step can go a long way toward making such arguments valid, reusable, and generally useful in helping produce high quality software systems.

Challenges and Opportunities. Achieving this vision has a number of challenges and opportunities. In our work on impact assessment of model change on assurance cases [39,40], we note that even small changes to the system may have significant impact on the assurance case. Because creation of an assurance case is costly, this brittleness must be addressed. One opportunity here is to recognize that assurance cases can be refactored to improve their qualities without affecting their semantics. For example, in Sect. 4, we showed that the LMS safety claim could either be decomposed first by hazards and then by subsystems or vice versa. Thus, we may want to choose the order of decomposition based on other goals, e.g., to minimize the impact of change on the assurance case by pushing the affected subgoals lower in the tree. Another issue is that complex systems yield correspondingly complex assurance cases. Since these must ultimately be judged by humans, we must manage the cognitive load the assurance case puts on the assessor. This creates opportunities for mechanized support, both in terms of querying, navigating and analyzing assurance cases as well as in terms of modularization and reuse of assurance cases.

Evidence composition discussed in Sect. 5 also presents significant challenges. While standards such as DO-178C and ISO26262 give recommendations on the use of testing and verification, it is not clear how to compose partial evidence or how to use results of one analysis to support another. Focusing on how each technique reduces potential faults in the program, clearly documenting their context of applicability (e.g., the small model hypothesis justifying partial unrolling of loops, properties not affected by approximations of complex program operations and datatypes often done by model-checkers, connections between the modeled and the actual environment, etc.) and ultimately connecting them to reducing uncertainties about whether the system satisfies the essential property are keys to making tangible progress in this area.

Finally, in Sect. 6, we showed how the assurance case view could apply to new development approaches such as ML. Although such new approaches provide benefits over traditional software development, they also create challenges for assurance. One challenge is that analysis techniques used for verification may be immature. For example, while neural networks have been studied since the

1950's, pragmatic approaches to their verification have been investigated only recently [53]. Another issue is that prerequisites for assurance may not be met by the development approach. For example, although they are expressive, neural networks suffer from uninterpretability [41] – that is, it is not feasible for a human to examine a trained network and understand what it is doing. This is a serious obstacle to assurance because formal and automated methods account for only part of the verification process, augmented by reviews. As a result, increasing the interpretability of ML models is an active area of current research.

While all these challenges are significant, the benefit of addressing them is worth the effort. As our world moves towards increasing automation, we must develop approaches for assuring the dependability of the complex systems we build. Without this, we either stall progress or run the risk of endangering ourselves – neither alternative seems desirable.

References

1. Bandur, V., McDermid, J.: Informing assurance case review through a formal interpretation of GSN core logic. In: Koornneef, F., van Gulijk, C. (eds.) SAFECOMP 2015. LNCS, vol. 9338, pp. 3–14. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24249-1_1
2. Bell, T.E., Thayer, T.A.: Software requirements: are they really a problem? In: Proceedings of the 2nd International Conference on Software Engineering, pp. 61–68. IEEE Computer Society Press (1976)
3. Bennion, M., Habli, I.: A candid industrial evaluation of formal software verification using model checking. In: Companion Proceedings of ICSE 2014, pp. 175–184 (2014)
4. Bloomfield, R., Bishop, P.: Safety and assurance cases: past, present and possible future - an Adelard perspective. In: Dale, C., Anderson, T. (eds.) Safety-Critical Systems: Problems, Process and Practice, pp. 51–67. Springer, London (2010). https://doi.org/10.1007/978-1-84996-086-1_4
5. Brunel, J., Cazin, J.: Formal verification of a safety argumentation and application to a complex UAV system. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7613, pp. 307–318. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33675-1_27
6. Carlan, C., Beyene, T.A., Ruess, H.: Integrated formal methods for constructing assurance cases. In: Proceedings of ISSRE 2016 Workshops (2016)
7. Cârlan, C., Gallina, B., Kacianka, S., Breu, R.: Arguing on software-level verification techniques appropriateness. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2017. LNCS, vol. 10488, pp. 39–54. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66266-4_3
8. Cârlan, C., Ratiu, D., Schätz, B.: On using results of code-level bounded model checking in assurance cases. In: Skavhaug, A., Guiochet, J., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2016. LNCS, vol. 9923, pp. 30–42. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45480-1_3
9. Castaameda, V., Ballejos, L., Caliusco, M.L., Galli, M.R.: The use of ontologies in requirements engineering. Glob. J. Res. Eng. **10**(6) (2010)
10. Cooray, D., Malek, S., Roshandel, R., Kilgore, D.: RESISTing reliability degradation through proactive reconfiguration. In: Proceedings of ASE 2010, pp. 83–92. ACM (2010)

11. Cyra, L., Gorski, J.: Support for argument structures review and assessment. J. Reliab. Eng. Syst. Saf. **96**, 26–37 (2011)
12. Czarnecki, K., Salay, R.: Towards a framework to manage perceptual uncertainty for safe automated driving. In: Gallina, B., Skavhaug, A., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2018. LNCS, vol. 11094, pp. 439–445. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99229-7_37
13. Davis, A., et al.: Identifying and measuring quality in a software requirements specification. In: 1993 Proceedings First International Software Metrics Symposium, pp. 141–152. IEEE (1993)
14. de la Vara, J.L.: Current and necessary insights into SACM: an analysis based on past publications. In: Proceedings of RELAW 2014, pp. 10–13. IEEE (2014)
15. Denney, E., Pai, G.: Evidence arguments for using formal methods in software vertification. In: Proceedings of ISSRE 2013 Workshops (2013)
16. Denney, E., Pai, G., Habli, I.: Towards measurement of confidence in safety cases. In: Proceedings of ESEM 2011 (2011)
17. Duan, L., Rayadurgam, S., Heimdahl, M.P.E., Sokolsky, O., Lee, I.: Representing confidence in assurance case evidence. In: Koornneef, F., van Gulijk, C. (eds.) SAFECOMP 2015. LNCS, vol. 9338, pp. 15–26. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24249-1_2
18. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: a framework for engineering self-tuning self-adaptive software systems. In: Proceedings of FSE 2010, pp. 7–16. ACM (2010)
19. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 214–238. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_9
20. Famelis, M., Chechik, M.: Managing design-time uncertainty. J. Softw. Syst. Model. (2017)
21. Fanmuy, G., Fraga, A., Llorens, J.: Requirements verification in the industry. In: Hammami, O., Krob, D., Voirin, J.L. (eds.) Complex Systems Design & Management, pp. 145–160. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-25203-7_10
22. Gal, Y., Ghahramani, Z.: Dropout as a Bayesian approximation: representing model uncertainty in deep learning. In: Proceedings of ICML 2016, pp. 1050–1059 (2016)
23. Gallina, B., Andrews, A.: Deriving verification-related means of compliance for a model-based testing process. In: Proceedings of DASC 2016 (2016)
24. Gehr, T., Milman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings of IEEE S&P 2018 (2018)
25. Goodenough, J., Weinstock, C., Klein, A.: Eliminative induction: a basis for arguing system confidence. In: Proceedings of ICSE 2013 (2013)
26. Gopinath, D., Wang, K., Zhang, M., Pasareanu, C., Khunshid, S.: Symbolic execution for deep neural networks. arXiv:1807.10439v1 (2018)
27. Graydon, P.J., Holloway, C.M.: An investigation of proposed techniques for quantifying confidence in assurance arguments. J. Saf. Sci. **92**, 53–65 (2017)
28. Greenwell, W.S., Knight, J.C., Holloway, C.M., Pease, J.J.: A taxonomy of fallacies in system safety arguments. In: Proceedings of ISSC 2006 (2006)
29. Grigorova, S., Maibaum, T.: Argument evaluation in the context of assurance case confidence modeling. In: Proceedings of ISSRE Workshops (2014)

30. GSN: Goal Structuring Notation Working Group, "GSN Community Standard Version 1", November 2011. http://www.goalstructuringnotation.info/
31. Guiochet, J., Hoang, Q.A.D., Kaaniche, M.: A model for safety case confidence assessment. In: Koornneef, F., van Gulijk, C. (eds.) SAFECOMP 2015. LNCS, vol. 9337, pp. 313–327. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24255-2_23
32. Habli, I., Kelly, T.: A generic goal-based certification argument for the justification of formal analysis. ENTCS **238**(4), 27–39 (2009)
33. Hawkins, R., Kelly, T.: A structured approach to selecting and justifying software safety evidence. In: Proceedings of SAFECOMP 2010 (2010)
34. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1
35. International Organization for Standardization: ISO 26262: Road Vehicles – Functional Safety, 1st version (2011)
36. International Organization for Standardization: ISO/AWI PAS 21448: Road Vehicles – Safety of the Intended Functionality (2019)
37. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
38. Kelly, T., Weaver, R.: The goal structuring notation – a safety argument notation. In: Proceedings of Dependable Systems and Networks Workshop on Assurance Cases (2004)
39. Kokaly, S., Salay, R., Cassano, V., Maibaum, T., Chechik, M.: A model management approach for assurance case reuse due to system evolution. In: Proceedings of MODELS 2016, pp. 196–206. ACM (2016)
40. Kokaly, S., Salay, R., Chechik, M., Lawford, M., Maibaum, T.: Safety case impact assessment in automotive software systems: an improved model-based approach. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2017. LNCS, vol. 10488, pp. 69–85. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66266-4_5
41. Lipton, Z.C.: The mythos of model interpretability. Commun. ACM **61**(10), 36–43 (2018)
42. Lutz, R.R.: Analyzing software requirements errors in safety-critical, embedded systems. In: Proceedings of IEEE International Symposium on Requirements Engineering, pp. 126–133. IEEE (1993)
43. Maksimov, M., Fung, N.L.S., Kokaly, S., Chechik, M.: Two decades of assurance case tools: a survey. In: Gallina, B., Skavhaug, A., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2018. LNCS, vol. 11094, pp. 49–59. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99229-7_6
44. Nair, S., de la Vara, J.L., Sabetzadeh, M., Falessic, D.: Evidence management for compliance of critical systems with safety standards: a survey on the state of practice. Inf. Softw. Technol. **60**, 1–15 (2015)
45. Nair, S., Walkinshaw, N., Kelly, T., de la Vara, J.L.: An evidential reasoning approach for assessing confidence in safety evidence. In: Proceedings of ISSRE 2015 (2015)
46. Nikora, A., Hayes, J., Holbrook, E.: Experiments in automated identification of ambiguous natural-language requirements. In: Proceedings 21st IEEE International Symposium on Software Reliability Engineering. IEEE Computer Society, San Jose (2010, to appear)

47. Pei, K., Cao, Y., Yang, J., Jana, S.: DeepXplore: automated whitebox testing of deep learning systems. In: Proceedings of SOSP 2017 (2017)
48. Ramirez, A.J., Jensen, A.C., Cheng, B.H.: A taxonomy of uncertainty for dynamically adaptive systems. In: Proceedings of SEAMS 2012 (2012)
49. Ras, G., van Gerven, M., Haselager, P.: Explanation methods in deep learning: users, values, concerns and challenges. In: Escalante, H.J., et al. (eds.) Explainable and Interpretable Models in Computer Vision and Machine Learning. TSSCML, pp. 19–36. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98131-4_2
50. Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: towards real-time object detection with region proposal networks. In: Advances in Neural Information Processing Systems, pp. 91–99 (2015)
51. Rushby, J., Xu, X., Rangarajan, M., Weaver, T.L.: Understanding and evaluating assurance cases. Technical report CR-2015-218802, NASA (2015)
52. Sadigh, D., Kapoor, A.: Safe control under uncertainty with probabilistic signal temporal logic. In: Proceedings of RSS 2016 (2016)
53. Seshia, S.A., Sadigh, D.: Towards verified artificial intelligence. CoRR, abs/1606.08514 (2016)
54. Tian, Y., Pei, K., Jana, S., Ray, B.: DeepTest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of ICSE 2018 (2018)
55. Van Lamsweerde, A.: Goal-oriented requirements engineering: a guided tour. In: Proceedings of RE 2001, pp. 249–262. IEEE (2001)
56. Verheij, B.: Evaluating arguments based on Toulmin's scheme. Argumentation **19**(3), 347–371 (2005)
57. Ward, S., Chapman, C.: Transforming project risk management into project uncertainty management. Int. J. Proj. Manag. **21**(2), 97–105 (2003)
58. Wassyng, A.: Private Communication (2019)
59. Yamamoto, S.: Assuring security through attribute GSN. In: Proceedings of ICITCS 2015 (2015)
60. Zeng, F., Lu, M., Zhong, D.: Using DS evidence theory to evaluation of confidence in safety case. J. Theoret. Appl. Inf. Technol. **47**(1) (2013)
61. Zhao, X., Zhang, D., Lu, M., Zeng, F.: A new approach to assessment of confidence in assurance cases. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7613, pp. 79–91. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33675-1_7

Software Verification I

Tool Support for Correctness-by-Construction

Tobias Runge[1(✉)], Ina Schaefer[1], Loek Cleophas[2,3], Thomas Thüm[1], Derrick Kourie[3,4], and Bruce W. Watson[3,4]

[1] Software Engineering, TU Braunschweig, Braunschweig, Germany
{tobias.runge,i.schaefer,t.thuem}@tu-bs.de
[2] Software Engineering Technology, TU Eindhoven, Eindhoven, The Netherlands
[3] Information Science, Stellenbosch University, Stellenbosch, South Africa
{loek,derrick,bruce}@fastar.org
[4] Centre for Artificial Intelligence Research, CSIR, Pretoria, South Africa

Abstract. Correctness-by-Construction (CbC) is an approach to incrementally create formally correct programs guided by pre- and postcondition specifications. A program is created using refinement rules that guarantee the resulting implementation is correct with respect to the specification. Although CbC is supposed to lead to code with a low defect rate, it is not prevalent, especially because appropriate tool support is missing. To promote CbC, we provide tool support for CbC-based program development. We present CorC, a graphical and textual IDE to create programs in a simple while-language following the CbC approach. Starting with a specification, our open source tool supports CbC developers in refining a program by a sequence of refinement steps and in verifying the correctness of these refinement steps using the theorem prover KeY. We evaluated the tool with a set of standard examples on CbC where we reveal errors in the provided specification. The evaluation shows that our tool reduces the verification time in comparison to post-hoc verification.

1 Introduction

Correctness-by-Construction (CbC) [12,13,19,23] is a methodology to construct formally correct programs guided by a specification. CbC can improve program development because every part of the program is designed to meet the corresponding specification. With the CbC approach, source code is incrementally constructed with a low defect rate [19] mainly based on three reasons. First, introducing defects is hard because of the structured reasoning discipline that is enforced by the refinement rules. Second, if defects occur, they can be tracked through the refinement structure of specifications. Third, the trust in the program is increased because the program is developed following a formal process [14].

Despite these benefits, CbC is still not prevalent and not applied for large-scale program development. We argue that one reason for this is missing tool

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 25–42, 2019.
https://doi.org/10.1007/978-3-030-16722-6_2

support for a CbC-style development process. Another issue is that the programmer mindset is often tailored to the prevalent post-hoc verification approach. CbC has been shown to be beneficial even in domains where post-hoc verification is required [29]. In post-hoc verification, a method is verified against pre- and postconditions. In the CbC approach, we refine the method stepwise, and we can check the method partially after each step since every statement is surrounded by a pair of pre- and postconditions. The verification of refinement steps and Hoare triples reduces the proof complexity since the proof task is split into smaller problems. The specifications and code developed using the CbC approach can be used to bootstrap the post-hoc verification process and allow for an easier post-hoc verification as the method constructed using CbC generally is of a structure that is more amenable to verification [29].

In this paper, we present CorC,[1] a tool designed to develop programs following the CbC approach. We deliberately built our tool on the well-known post-hoc verifier KeY [4] to profit from the KeY ecosystem and future extensions of the verifier. We also add CbC as another application area to KeY, which opens the possibility for KeY users to adopt the CbC approach. This could spread the constructive CbC approach to areas where post-hoc verification is prevalent.

Our tool CorC offers a hybrid textual-graphical editor to develop programs using CbC. The textual editor resembles a normal programming editor, but is enriched with support for pre- and postcondition specifications. The graphical editor visualizes the code, its specification, and the program refinements in a tree-like structure. The developers can switch back and forth between both views. In order to support the correct application of the refinement rules, the tool is integrated with KeY [4] such that proof obligations can be immediately discharged during program development. In a preliminary evaluation, we found benefits of CorC compared to paper-and-pencil-based application of CbC and compared to post-hoc verification.

2 Foundations of Correctness-by-Construction

Classically, CbC [19] starts with the specification of a program as a Hoare triple comprising a precondition, an abstract statement, and a postcondition. Such a triple, say T, should be read as a total correctness assertion: if T is in a state where the precondition holds and its abstract statement is executed, then the execution will terminate and the postcondition will hold. T will be true for a certain set of concrete program instantiations of the abstract program and false for other instantiations. A refinement of T is a triple, say T', which is true for a subset of concrete programs that render T to be true.

In our work, pre-/post-condition specifications for programs are written in *first-order logic* (FOL). A formula in FOL consists of atomic formulas which are logically connected. An atomic formula is a predicate which evaluates to true or

[1] https://github.com/TUBS-ISF/CorC, CorC is an acronym for Correctness-by-Construction.

$\{P\}\ S\ \{Q\}$	*can be refined to*
1. Skip :	$\{P\}$ *skip* $\{Q\}$ *iff* P *implies* Q
2. Assignment :	$\{P\}\ x := E\ \{Q\}$ *iff* P *implies* $Q[x := E]$
3. Composition :	$\{P\}\ S1\ ;\ S2\ \{Q\}$ *iff there is an intermediate condition* M *such that* $\{P\}\ S1\ \{M\}$ *and* $\{M\}\ S2\ \{Q\}$
4. Selection :	$\{P\}$ **if** $G_1 \rightarrow S_1$ **elseif** $\ldots\ G_n \rightarrow S_n$ **fi** $\{Q\}$ *iff* (P *implies* $G_1 \vee G_2 \vee \ldots \vee G_n$) *and* $\{P \wedge G_i\}\ S_i\ \{Q\}$ *holds for all* i.
5. Repetition :	$\{P\}$ **do** $[I, V]\ G \rightarrow S$ **od** $\{Q\}$ *iff* (P *implies* I) *and* ($I \wedge \neg G$ *implies* Q) *and* $\{I \wedge G\}\ S\ \{I\}$ *and* $\{I \wedge G \wedge V{=}V_0\}\ S\ \{I \wedge 0{\leq}V \wedge V{<}V_0\}$
6. Weaken pre :	$\{P'\}\ S\ \{Q\}$ *iff* P *implies* P'
7. Strengthen post :	$\{P\}\ S\ \{Q'\}$ *iff* Q' *implies* Q
8. Subroutine :	$\{P\}$ *Sub* $\{Q\}$ *with subroutine* $\{P'\}$ *Sub* $\{Q'\}$ *iff* P *is equal to* P' *and* Q' *is equal to* Q

Fig. 1. Refinement rules in CbC [19]

false. Programs in this work are written in the CorC language, which is inspired by the *Guarded Command Language* (GCL) [11] and presented below.

For the concrete instantiation of conditions and assignments, our tool uses a host language. We decided for Java, but other languages are also possible.

To create programs using CbC, we use refinement rules. A Hoare triple is refined by applying rules, which introduce CorC language statements, so that a concrete program is created. The concrete program obtained by refinement is guaranteed to be correct by construction, provided that the correctness-preserving refinement steps have been accurately applied. In Fig. 1, we present the statements and refinement rules used in CbC and our tool.

Skip. A skip or empty statement is a statement that does not alter the state of the program (i.e., it does nothing) [11,19]. This means a Hoare triple with a skip statement evaluates to true if the precondition implies the postcondition.

Assignment. An assignment statement assigns an expression of type T to a variable, also of type T. In the tool, we use a Java-like assignment ($x = y$). To refine a Hoare triple $\{P\}\ S\ \{Q\}$ with an assignment statement, the assignment rule is used. This rule replaces the abstract statement S by an assignment $\{P\}\ x = E\ \{Q\}$ iff P implies $Q[x := E]$.

Composition. A composition statement is a statement which splits one abstract statement into two. A Hoare triple $\{P\}\ S\ \{Q\}$ is split to $\{P\}\ S_1\ \{M\}$ and $\{M\}\ S_2\ \{Q\}$ in which S is refined to $S1$ and $S2$. M is an intermediate condition which evaluates to true after $S1$ and before $S2$ is executed [11].

Selection. Selection in our CorC language works as a switch statement. It refines a Hoare triple $\{P\}\ S\ \{Q\}$ to $\{P\}$ **if** $G_1 \rightarrow S_1$ **elseif** $\ldots\ G_n \rightarrow S_n$ **fi** $\{Q\}$. The guards G_i are evaluated, and the sub-statement S_i of the *first* satisfied guard is executed.

We use a switch-like statement so that every sub-statement has an associated guard for further reasoning. The selection refinement rule can only be used if the precondition P implies the disjunction of all guards so that at least one sub-statement could be executed.

Repetition. The repetition statement $\{\mathtt{P}\}$ **do** $[\mathtt{I}, \mathtt{V}]$ $\mathtt{G} \rightarrow$ S **od** $\{\mathtt{Q}\}$ works like a while loop in other languages. If the loop guard G evaluates to true, the associated loop statement S is executed. The repetition statement is specified with an invariant I and a variant V. To refine a Hoare triple $\{\mathtt{P}\}$ S $\{\mathtt{Q}\}$ with a repetition statement, (1) the precondition P has to imply the invariant I of the repetition statement, (2) the conjunction of invariant and the negation of the loop guard G have to imply the postcondition Q, and (3) the loop body has to preserve the invariant by showing that $\{\mathtt{I} \wedge \mathtt{G}\}$ S $\{\mathtt{I}\}$ holds. To verify termination, we have to show that the variant V monotonically decreases in each loop iteration and has 0 as a lower bound.

Weaken precondition. The precondition of a Hoare triple can be weakened if necessary. The weaken precondition rule replaces the precondition P with a new one $\mathtt{P}'$ only if P implies $\mathtt{P}'$ [12].

Strengthen postcondition. To strengthen a postcondition, the strengthen postcondition rule can be used. A postcondition Q is replaced by a new one $\mathtt{Q}'$ only if $\mathtt{Q}'$ implies Q [12].

Subroutine. A subroutine can be used to split a program into smaller parts. We use a simple subroutine call where we prohibit side effects and parameters. A triple $\{\mathtt{P}\}$ S $\{\mathtt{Q}\}$ can be refined to a subroutine $\{\mathtt{P}'\}$ *Sub* $\{\mathtt{Q}'\}$, if the precondition $\mathtt{P}'$ of the subroutine is equal to the precondition P of the refined statement and the postcondition $\mathtt{Q}'$ of the subroutine is equal to the postcondition Q of the refined statement. The subroutine can be constructed as a separate CbC program to verify that it satisfies the specification. The Hoare triple $\{\mathtt{P}'\}$ *Sub* $\{\mathtt{Q}'\}$ is the starting point to construct a program using CbC.

3 Correctness-by-Construction by Example

To introduce the programming style of CbC, we demonstrate the construction of a linear search algorithm using CbC [19]. The linear search problem is defined as follows: We have an integer array a of some length, and an integer variable x. We try to find an element in the array a which has the same value as the variable x, and we return the index i where the (last) element x was found, or -1 if the element is not in the array.

To construct the algorithm, we start with concretizing the pre- and postcondition of the algorithm. Before the algorithm is executed, we know that we have an integer array. Therefore, we specify $\mathtt{a} \neq \mathtt{null} \wedge \mathtt{a.length} \geq 0$ as precondition P. The postcondition forces that if the index i is greater than or equal to zero, the element is found on the returned index i ($\mathtt{Q} := (\mathtt{i} \geq 0 \implies \mathtt{a}[\mathtt{i}] = \mathtt{x})$).

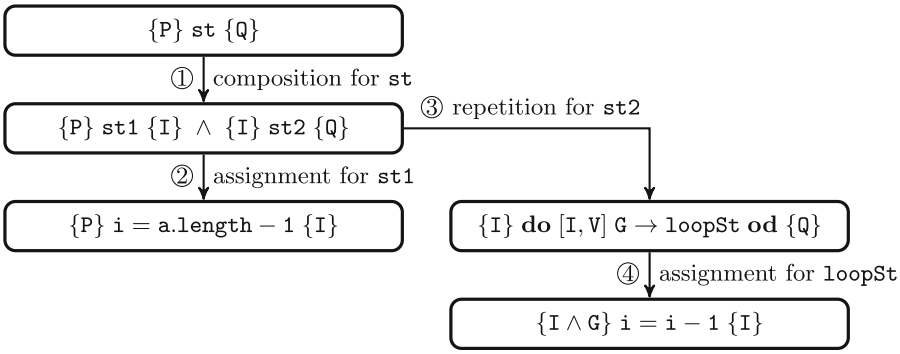


Fig. 2. Refinement steps for the linear search algorithm

Our algorithm traverses the array in reverse order and checks for each index whether the value is equal to `x`. In this case, the index is returned. To create this algorithm, we construct an invariant `I` for the loop:

$$\texttt{I} := \neg\texttt{appears}(\texttt{a}, \texttt{x}, \texttt{i}+1, \texttt{a.length}) \wedge \texttt{i} \geq -1 \wedge \texttt{i} < \texttt{a.length}$$

The invariant is used to split the array into two parts. A part from $\texttt{i}+1$ to `a.length` where `x` is not contained, and a part from zero to `i` which is not checked yet. In every iteration, the next index of the array is checked. The predicate $\texttt{appears}(\texttt{a}, \texttt{x}, \texttt{l}, \texttt{h})$ asserts that `x` occurs in array `a` inside the range from `l` (included) to `h` (excluded). The predicate can be translated to FOL as $\exists \texttt{i} : (\texttt{i} \geq \texttt{l} \wedge \texttt{i} < \texttt{h} \wedge \texttt{a}[\texttt{i}] = \texttt{x})$.

We can use the CbC refinement rules to implement linear search. The refinement steps for the example are shown in Fig. 2 and numbered from ① to ④. To create a loop in the program, we need to initialize a loop counter variable to establish the invariant. Therefore, we split the program by introducing a composition statement (① in Fig. 2). The invariant `I` is used as intermediate condition (i.e., $\texttt{M} := \texttt{I}$), because it has to be true after the initialization, and before the first loop step. The statement `st1` is refined to an assignment statement ②. We initialize `i` with $\texttt{a.length} - 1$ to start at the end of the array. This assignment satisfies the intermediate condition `I` where `i` is replaced by $\texttt{a.length} - 1$. The range of `appears` is empty, and therefore the predicate evaluates to true. To refine the second statement (`st2`), we use the repetition refinement rule ③. As long as `x` is not found, we iterate through the array. As guard of the repetition, we use $(\texttt{i} \geq 0 \wedge \texttt{a}[\texttt{i}] \neq \texttt{x})$. The invariant of the repetition is the invariant `I` introduced above. The variant `V` is $\texttt{i}+1$. To verify that this refinement is valid, we have to verify that the precondition of the repetition statement implies the invariant, and that the invariant and the negated guard imply the postcondition of the repetition (cf. Rule 5). Both are valid because the precondition is equal to the invariant and the postcondition of the repetition statement (in this case it is `Q`) is equal to the negated guard. The last step is to refine the abstract loop statement (`loopSt`) ④. We use an assignment to decrease `i` and get the final

program. We can verify that the invariant holds after each loop iteration. The program terminates because the variant decreases in every step and it is always greater than or equal to zero.

4 Tool Support in CorC

CorC extends KeY's application area by enabling CbC to spread the constructive engineering to areas where post-hoc verification is prevalent. KeY programmers can use both approaches to construct formally correct programs. By using CorC, they develop specification and code that can bootstrap the post-hoc verification. The CorC tool[2] is realized as an Eclipse plug-in in Java. We use the Eclipse Modeling Framework (EMF)[3] to specify a CbC meta model. This meta model is used by two editor views, a textual and a graphical editor. The Hoare triple verification is implemented by the deductive program verification tool KeY [4]. In the following list, we summarize the features of CorC.

- Programs are written as Hoare triple specifications, including pre-/postcondition specifications and abstract statements or assignment/skip statements in concrete triples.
- CorC has eight rules to construct programs: skip, assignment, composition, selection, repetition, weakening precondition, strengthening postcondition, and subroutine (cf. Sect. 2).
- Pre-/postconditions and invariant specification are automatically propagated through the program.
- CorC comprises a graphical and a textual editor that can be used interchangeably.
- Up to now, CorC supports integers, chars, strings, arrays, and subroutine calls without side effects, I/O, and library calls.
- Hoare triples are typically verified by KeY automatically. If the proof cannot be closed automatically, the user can interact with KeY.
- Helper methods written in Java 1.5 can be used in a specification.
- CorC comprises content assist and an automatic generation of intermediate conditions.

4.1 Graphical Editor

The graphical editor represents CbC-based program refinement by a tree structure. A node represents the Hoare triple of a specific CorC language statement. Figure 3 presents the linear search algorithm of Sect. 3 in the graphical editor. The structure of the tree is the same as in Fig. 2. The additional nodes on the right specify used program variables including their type and global invariant

[2] https://github.com/TUBS-ISF/CorC.
[3] https://eclipse.org/emf/.

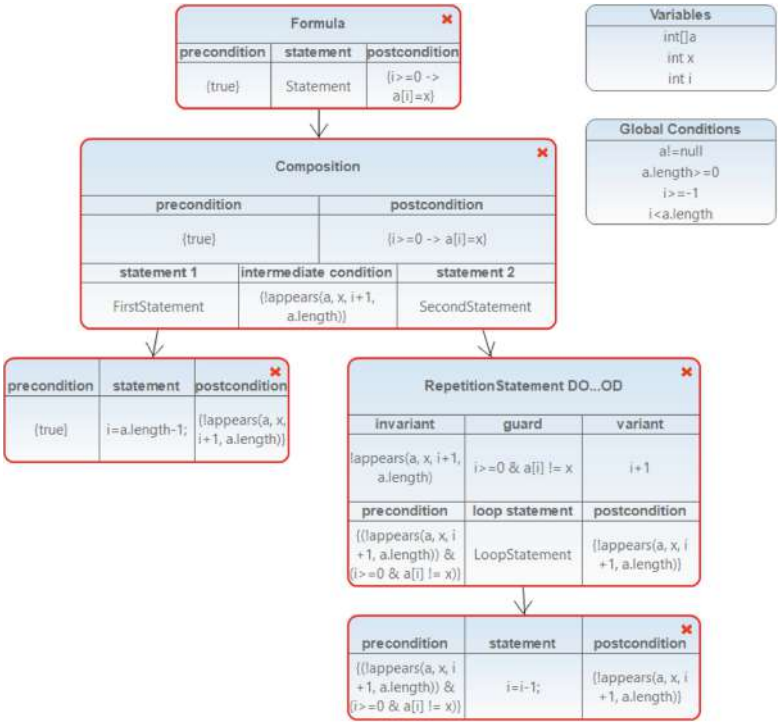


Fig. 3. Linear search example in the graphical editor

conditions. The global invariant conditions are added to every pre- and postcondition of Hoare triples to simplify the construction of the program. In the example, we specify the array `a` and the range of variable `i` to support the verification, as KeY requires this range to be explicit for verification.

The root node of the tree shows the abstract Hoare triple for the overall program with a symbolic name for the abstract statement. In every node, the pre- and postcondition are specified on the left and right of the node under the corresponding header. A composition statement node, the second statement of the tree, contains the pre- and postcondition and additionally defines an intermediate condition. The intermediate condition is the middle term in the bottom line. Both abstract sub-statements of the composition have a symbolic name and can be further refined by adding a connection to another node (i.e., creating a parent-child relation). The repetition node contains fields to specify the invariant, the guard and the variant of the repetition. These fields are in the middle row. The pre- and postcondition are associated to the inner loop statement. An assignment node (cf. both leaf nodes of the figure) contains the precondition, the assignment, and the postcondition. The representations of the nodes for the refinements not illustrated in this example are similar.

Refinement steps are represented by edges. The pre- and postconditions are propagated from parents to their children on drawing the parent/child relation. We explicitly show the propagated conditions in a node to improve readability. The propagated conditions from the parent are unmodifiable because refinement rules determine explicitly how conditions are propagated. An exception are the rules to weaken the precondition or strengthen the postcondition. Here, the conditions can be overridden. At the repetition statement, we only depict the pre-/postconditions of the inner loop statement to reduce the size of this node. The pre-/postconditions of the parent node (in our example the composition statement) are not shown explicitly, but they are propagated internally to verify that the repetition refinement rule is satisfied. To visualize the verification status, the nodes have a green border if proven, a red one otherwise.

By showing the Hoare triples explicitly, problems in the program can be localized. If some leaf node cannot be proven, the user has to check the assignment and the corresponding pre-/postcondition. If an error occurred, the conditions on the refinement path up to pre-/postcondition of the starting Hoare triple can be altered. Other paths do not need to be checked. To prove the program correct, we have to prove that the refinement is correct. Aside from the side conditions of refinement rules (cf. iff conditions in refinement rules), only the leaf nodes of the refinement tree which contain basic Hoare triples with skip or assignment statements need to be verified by a prover, while all composite statements are correct by construction of their conditions.

To support the user in developing intermediate conditions for composition statements, our tool can compute the weakest precondition from a postcondition and a concrete assignment by using the KeY theorem prover. So, the user can create a specific assignment statement and generate the intermediate conditions afterwards. We also support modularization, to cover cases where algorithms become too large. Sub-algorithms can be created using CbC in other CorC programs. We introduce a simple subroutine rule which can be used as a leaf node in the editor. The subroutine has a name and it is connected to a second diagram with the same name as the subroutine. This subroutine call is similar to a classic method call. It can be used to decompose larger CbC developments to multiple smaller programs.

4.2 Textual Editor

The textual editor is an editor for the CorC programming language described above. The user writes code by using keywords for the specific statements and enriches the code with conditions, such as invariants or intermediate conditions, and assignments in our CorC syntax. The syntax of the composed statements in the textual editor is shown in Fig. 4. In the `GlobalConditions` declaration, we enumerate the needed global conditions separated with a comma. The used variables are enumerated after the `JavaVariables` keyword.

The linear search example program presented in Sect. 3 is shown in the syntax of CorC in Listing 1. The program starts with keyword `Formula`. The pre- and postcondition of the abstract Hoare triple are written after the `pre:` and `post:`

Selection statement	*Repetition statement*
if (“guard”) **then** {statement}	**while** (“guard”)
elseif (“guard”) **then** {statement}	**inv**: [“invariant”] **var**: [“variant”]
...	**do** {statement} **od**
fi	

Fig. 4. Syntax of statements in textual editor

```
1  Formula "linearSearch"
2  pre: {"true"}
3  {
4    {
5      i=a.length-1;
6    }
7    intm: ["!appears(a, x, i+1, a.length)"]
8    {
9      while ("i>=0 & a[i]!=x")
10     inv: ["!appears(a, x, i+1, a.length)"]
11     var: ["i+1"] do
12     {
13       i=i-1;
14     } od
15   }
16 }
17 post: {"i>=0 -> a[i]=x"}
18
19 GlobalConditions
20   conditions {"a!=null", "a.length>=0",
21     "i>=-1", "i<a.length"}
22
23 JavaVariables
24   variables {"int[] a", "int x", "int i"}
```

Listing 1. Linear search example in the textual editor

keywords. The abstract statement of the Hoare triple is refined to a composition statement in lines 3–16. The statements are surrounded by curly brackets to establish the refinement structure. We have the first statement in lines 4–6, the intermediate condition in line 7 and the second statement in lines 8–15. The first statement is refined to an assignment (Line 5). The refinement is done by introducing an assignment in Java syntax (`i = a.length − 1;`). The second statement is refined to a repetition statement (cf. the syntax of a repetition statement in Fig. 4). We specify the guard, the invariant, and the variant. Finally, the single statement of the loop body is refined to an assignment in Line 13.

As in the graphical editor, pre-/postconditions are propagated top-down from a parent to a child statement. For example, the intermediate condition of a

```
1 \javaSource "src";
2 \include "helper.key";
3 \programVariables {int x;}
4 \problem {
5   (x = 0) -> \<{x=x+1;}\> (x = 1)
6 }
```

Listing 2. KeY problem file

composition statement which is the postcondition of the first sub-statement and the precondition of the second, appears only once in the editor (e.g., Line 7). To support the user, we implemented syntax highlighting and a content assist. When starting to write a statement, a user may employ auto-completion where the statements are inserted following the syntax in Fig. 4. The user can specify the conditions, then the next statement can be refined. The editor also automatically checks the syntax and highlights syntax errors. Information markers are used to indicate statements which are not proven yet. For example, the Hoare triple of the assignment statement (`i = a.length - 1`) in Listing 1 has to be verified, and CorC marks the statement according to the proof completion results.

4.3 Verification of CorC Programs

To prove the refined program is correct, we have to prove side conditions of refinements correct (e.g., prove that an assignment satiesfies the pre-/postcondition specification). This reduces the proof complexity because the challenge to prove a complete program is decomposed into smaller verification tasks. The intermediate Hoare triples are verified indirectly through the soundness of the refinement rules and the propagation of the specifications from parent nodes to child nodes [19]. Side conditions occur in all refinements (cf. iff conditions in refinement rules). These side conditions, such as the termination of repetition statements or that at least one guard in a selection has to evaluate to true, are proven in separate KeY files.

For the proof of concrete Hoare triples, we use the deductive program verifier KeY [4]. Hoare triples are transformed to KeY's dynamic logic syntax. The syntax of KeY problem files is shown in Listing 2. Using the keyword `javaSource`, we specify the path to Java helper methods which are called in the specifications. These methods have to be verified independently with KeY. A KeY helper file, where the users can define their own FOL predicates for the specification, is included with the keyword `include`. For example, in CorC a predicate $appears(a, x, l, h)$ (cf. the linear search example) can be used which is specified in the helper file as a FOL formula. The variables used in the program are listed after the keyword `programVariables`. After `problem`, we define the Hoare triple to be proven, which is translated to dynamic logic as used by KeY. KeY problem files are verified by KeY. As we are only verifying simple Hoare triples with skip

or assignment statements, KeY is usually able to close the proofs automatically if the Hoare triple is valid.

To verify total correctness of the program, we have to prove that all repetition statements terminate. The termination of repetition statements is shown by proving that the variants in the program monotonically decrease and are bounded. Without loss of generality, we assume this bound to equal 0, as this is what KeY requires. This is done by specifying the problem in the KeY file in the following way: `(invariant & guard) -> {var0:=var} \<{std}\> (invariant & var<var0 & var>=0)`. The code of the loop body is specified at `std` to verify that after one iteration of the loop body the variant `var` is smaller than before but greater than or equal to zero.

To verify Hoare triples in the graphical editor, we implemented a menu entry. The user can right-click on a statement and start the automatic proof. If the proof is not closed, the user can interact with the opened KeY interface. To prove Hoare triples in the textual editor, we automatically generate all needed problem files for KeY whenever the user saves the editor file. The proof of the files is started using a menu button. The user gets feedback which triples are not proven by means of markers in the editor.

4.4 Implementation as Eclipse Plugin

We extended the Eclipse modeling framework with plugins to implement the two editors. We have created a meta model of the CbC language to represent the required constructs (i.e., statements with specification). The statements can be nested to create the CbC refinement hierarchy. The graphical and the textual editor are projections on the same meta model. The graphical editor is implemented using the framework Graphiti.[4] It provides functionality to create nodes and to associate them to domain elements, such as statements and specifications. The nodes can be added from a palette at the side of the editor, so no incorrect statement with its associated specification can be created. We implemented editing functionality to change the text in the node; the background model is changed simultaneously. Graphiti also provides the possibility to update nodes (e.g., to propagate pre- and postconditions), if we connect those nodes by refinement edges. The refinement is checked for compliance with the CbC rules.

The textual editor is implemented using XText.[5] We created a grammar covering every statement and the associated specification. If the user writes a program, the text is parsed and translated to an instance of the meta model. If a program is created in one editor, a model (an instance of our meta model) of the program is created in the background. We can easily transform one view into the other. The transformation is a generation step and not a live synchronization between both views, but it is carried out invisibly for the user when changing the views.

[4] https://eclipse.org/graphiti/.
[5] https://eclipse.org/Xtext/.

Table 1. Evaluation of the example programs

Algorithm	#Nodes in GE	#Lines in TE	#Lines with JML	#Verified CorC triples	CbC Total Proof-Nodes	CbC Total Proof-Time	PhV Total Proof-Nodes	PhV Total Proof-Time
Linear Search	5	12	10	5/5	285	0.4 s	589	1.2 s
Max. Element	9	21	15	9/9	1023	1.2 s	993	1.8 s
Pattern Matching	14	23	20	13/13	21131	54.9 s	201619	1479.3 s
Exponentiation	7	21	17	7/7	6588	15.2 s	7303	20.4 s
Log. Approx.	5	16	12	5/5	13756	42.7 s	18835	68.5 s
Dutch Flag	8	26	24	8/8	4107	5.7 s	4993	13.4 s
Factorial	5	15	13	4/4	1554	3.6 s	1598	4.4 s

(GE) Grahical Editor, (TE) Textual Editor, (PhV) Post-hoc Verification

In implementing CorC, we considered the exchangeability of the host language. The specifications and assignments are saved as strings in the meta model. They are checked by a parser to comply with Java. This parser could be exchanged to support a different language. The verification is done by generating KeY files which are then evaluated by KeY. Here, we have to exchange the generation of the files if another theorem prover should be integrated. The information of the meta model may have to be adopted to fit the needs of the other prover. We also have to implement a programmatic call to the other prover.

5 Evaluation

The tool support offers new chances to evaluate CbC versus post-hoc verification. We quantitatively compare the development and verification of programs with CorC and with post-hoc verification. This is to check the hypothesis that the verification of algorithms is faster with CorC than with post-hoc verification. We created the first eight algorithms from the book by Kourie and Watson [19] in our graphical editor. For comparison purposes, we also wrote each example as a plain Java program with JML specifications in order to directly verify it with KeY. The specifications are the same as in CorC. We measured the verification time and the proof nodes that KeY needed to close the proofs for both approaches. The results of the evaluation are presented in Table 1 (verification time rounded).

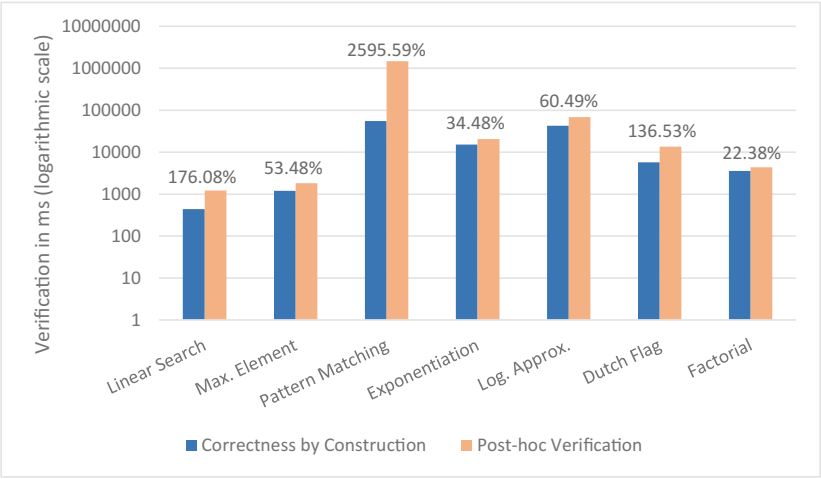


Fig. 5. Proof time of CbC and post-hoc verification in logarithmic scale

The algorithms have 5 to 14 nodes in the graphical editor and 12 to 26 lines of code in the textual editor. The Java version with a JML specification always has fewer lines (between 8% and 29% smaller). The additional specifications, such as the intermediate conditions of composition statements, and the global invariant conditions and variables cause more lines of code in the CbC program.

The verification of the eight algorithms worked nearly without problems. We verified 7 out of 8 examples within CorC. In the cases without problems, every Hoare triple and the termination of the loops could be proven. We had to prove fewer Hoare triples than nodes in the editor, as not every node has to be proven separately. Composition nodes are proven indirectly through the refinement structure. For *exponentiation*, *logarithm*, and *factorial*, we had to implement recursive helper methods which are used in the specification. Therefore, the programs impose upper bounds for integers to shorten the proof. The *binary search* algorithm could not be verified automatically in KeY using post-hoc verification or CorC. In each step, when the element is not found, the algorithm halves the array. KeY could not prove that the searched element is in the new boundaries because verification problems with arithmetic division are hard to prove for KeY automatically.

In the case of measured proof nodes, *maximum element* needs slightly fewer nodes proved with post-hoc verification than with CbC. In the other cases, the proofs for the algorithms constructed with CbC are 3% to 854% smaller. The largest difference was measured for the *pattern matching* algorithm. The proof is reduced to a ninth of the nodes.

The verification time is visualized in Fig. 5. The time is measured in milliseconds and scaled logarithmically. The proofs for the CbC approach are always faster showing lower proof complexity. For *maximum element*, *exponentiation*,

logarithm and *factorial*, the post-hoc verification time requires between 22% and 60% more time. The difference increases for *Dutch flag* and *linear search* to 137% and 176%, respectively. Algorithm *pattern matching* has the biggest difference. Here, the CbC approach needs nearly a minute, but the post-hoc approach needs over 24 min. To verify our hypothesis, we apply the non-parametric paired Wilcoxon-Test [30] with a significance level of 5%. We can reject the null hypothesis that CbC verification and post-hoc verification have no significant difference in verification time (p-value = 0.007813). This rejection of the null hypothesis in an empirical evidence for our hypothesis that verification is faster with CorC than with post-hoc verification.

With our tool support, we were able to compare the CbC approach with post-hoc verification. For our examples, we evaluated that the verification effort is reduced significantly which indicates a reduced proof complexity. It is worthwhile to further investigate the CbC approach, also to profit from synergistic effects in combination with post-hoc verification. As we built CorC on top of KeY, the post-hoc verification of programs constructed with CorC is feasible.

An advantage of CorC is the overview on all Hoare triples during development. In this way, we found some specifications where descriptions in the book by Kourie and Watson [19] were not precise enough to verify the problem in KeY. For example, in the *pattern matching* algorithm, we had to verify two nested loops. At one point, we had to verify that the invariant of the inner loop implies the invariant of the outer loop. This was not possible, so we extended the invariant of the inner loop to be the conjunction of both invariants. In the book of Kourie and Watson [19], this conjunction of both invariants was not explicitly used.

6 Related Work

We compare CorC to other programming languages and tools using specification or refinements. The programming language Eiffel is an object-oriented programming language with a focus on design-by-contract [21,22]. Classes and methods are annotated with pre-/postconditions and invariants. Programs written in Eiffel can be verified using AutoProof [18,28]. The verification tool translates the program with assertions to a logic formula. An SMT-solver proves the correctness and returns the result. Spec# is a similar tool for specifying C# programs with pre-/postcondition contracts. These programs can be verified using Boogie. The code and specification is translated to an intermediate language (BoogiePL) and verified [5,6]. VCC [8] is a tool to annotate and verify C code. For this purpose, it reuses the Spec# tool chain. VeriFast [16] is another tool to verify C and Java programs with the help of contracts. The contracts are written in separation logic (a variant of Hoare logic). As in Eiffel, the focus of Spec#, VCC, and VeriFast is on post-hoc verification and debugging failed proof attempts.

The Event-B framework [2] is a related CbC approach. Automata-based systems including a specification are refined to a concrete implementation.

Atelier B [1] implements the B method by providing an automatic and interactive prover. Rodin [3] is another tool implementing the Event-B method. The main difference to CorC is that CorC works on code and specifications rather than on automata-based systems.

ArcAngel [25] is a tool supporting Morgan's refinement calculus. Rules are applied to an initial specification to produce a correct implementation. The tool implements a tactic language for refinements to apply a sequence of rules. In comparison to our tool, ArcAngel does not offer a graphical editor to visualize the refinement steps. Another difference is that ArcAngel creates a list of proof obligations which have to be proven separately. CRefine [26] is a related tool for the Circus refinement calculus, a calculus for state-rich reactive systems. Like our tool, CRefine provides a GUI for the refinement process. The difference is that we specify and implement source code, but they use a state-based language. ArcAngelC [10] is an extension to CRefine which adds refinement tactics.

The tools iContract [20] and OpenJML [9] apply design-by-contract. They use a special comment tag to insert conditions into Java code. These conditions are translated to assertions and checked at runtime which is a difference to our tool because no formal verification is done. DBC-Python is a similar approach for the Python language which also checks assertions at runtime [27].

To verify the CbC program, we need a theorem prover for Hoare triples, such as KeY [4]. There are other theorem provers which could be used (e.g., Coq [7] or Isabelle/HOL [24]). The Tecton Proof System [17] is a related tool to structure and interactively prove Hoare logic specification. The proofs are represented graphically as a set of linked trees. These interactive provers do not fit our needs because we want to automate the verification process. KeY provides a symbolic execution debugger (SED) that represents all execution paths with specifications of the code to the verification [15]. This visualization is similar to our tree representation of the graphical editor. The SED can be used to debug a program if an error occur during the post-hoc verification process.

7 Conclusion and Future Work

We implemented CorC to support the Correctness-by-Construction process of program development. We created a textual and a graphical editor that can be used interchangeably to enable different styles of CbC-based program development. The program and its specification are written in one of the editors and can be verified using KeY. This reduces the proof complexity with respect to post-hoc verification. We extended the KeY ecosystem with CorC. CorC opens the possibility to utilize CbC in areas where post-hoc verification is used as programmers could benefit from synergistic effects of both approaches. With tool support, CbC can be studied in experiments to determine the value of using CbC in industry.

For future work, we want to extend the tool support, and we want to evaluate empirically the benefits and drawbacks of CorC. To extend the expressiveness, we implement a rule for methods to use method calls in CorC. These methods have to be verified independently by CorC/KeY. We could investigate whether the method call rules of KeY can be used for our CbC approach. Another future work is the inference of conditions to reduce the manual effort. Postconditions can be generated automatically for known statements by using the strongest postcondition calculus. Invariants could be generated by incorporating external tools. As mentioned earlier, other host languages and other theorem provers can be integrated in our IDE.

The second work package for future work comprise the evaluation with a user study. We could compare the effort of creating and verifying algorithms with post-hoc verification and with our tool support. The feedback can be used to improve the usability of the tool.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transfer **12**(6), 447–466 (2010)
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification - The KeY Book: From Theory to Practice, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
5. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011)
6. 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
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-662-07964-5
8. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
9. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35

10. Conserva Filho, M., Oliveira, M.V.M.: Implementing tactics of refinement in CRefine. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 342–351. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_24
11. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
12. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Upper Saddle River (1976)
13. Gries, D.: The Science of Programming. Springer, Heidelberg (1987). https://doi.org/10.1007/978-1-4612-5983-1
14. Hall, A., Chapman, R.: Correctness by construction: developing a commercial secure system. IEEE Softw. **19**(1), 18–25 (2002)
15. Hentschel, M.: Integrating symbolic execution, debugging and verification. Ph.D. thesis, Technische Universität Darmstadt (2016)
16. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the verifast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_21
17. Kapur, D., Nie, X., Musser, D.R.: An overview of the Tecton proof system. Theoret. Comput. Sci. **133**(2), 307–339 (1994)
18. Khazeev, M., Rivera, V., Mazzara, M., Johard, L.: Initial steps towards assessing the usability of a verification tool. In: Ciancarini, P., Litvinov, S., Messina, A., Sillitti, A., Succi, G. (eds.) SEDA 2016. AISC, vol. 717, pp. 31–40. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-70578-1_4
19. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27919-5
20. Kramer, R.: iContract - the Java design by contract tool. In: Proceedings, Technology of Object-Oriented Languages. TOOLS 26 (Cat. No. 98EX176), pp. 295–307. IEEE, August 1998
21. Meyer, B.: Eiffel: a language and environment for software engineering. J. Syst. Softw. **8**(3), 199–246 (1988)
22. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992)
23. Morgan, C.: Programming from Specifications, 2nd edn. Prentice Hall, Upper Saddle River (1994)
24. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
25. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. Formal Aspects Comput. **15**(1), 28–47 (2003)
26. Oliveira, M.V.M., Gurgel, A.C., Castro, C.G.: CRefine: support for the circus refinement calculus. In: 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods, pp. 281–290. IEEE, November 2008
27. Plosch, R.: Tool support for design by contract. In: Proceedings, Technology of Object-Oriented Languages. TOOLS 26 (Cat. No. 98EX176), pp. 282–294. IEEE, August 1998
28. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_53

29. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: Correctness-by-construction and post-hoc verification: a marriage of convenience? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 730–748. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_52
30. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29044-2

Automatic Modeling of Opaque Code for JavaScript Static Analysis

Joonyoung Park[1,2(✉)], Alexander Jordan[1(✉)], and Sukyoung Ryu[2(✉)]

[1] Oracle Labs Australia, Brisbane, Australia
{joonyoung.p.park,alexander.jordan}@oracle.com
[2] KAIST, Daejeon, Republic of Korea
{sryu.cs,gmb55}@kaist.ac.kr

Abstract. Static program analysis often encounters problems in analyzing library code. Most real-world programs use library functions intensively, and library functions are usually written in different languages. For example, static analysis of JavaScript programs requires analysis of the standard built-in library implemented in host environments. A common approach to analyze such *opaque code* is for analysis developers to build models that provide the semantics of the code. Models can be built either manually, which is time consuming and error prone, or automatically, which may limit application to different languages or analyzers. In this paper, we present a novel mechanism to support automatic modeling of opaque code, which is applicable to various languages and analyzers. For a given static analysis, our approach automatically computes analysis results of opaque code via dynamic testing during static analysis. By using testing techniques, the mechanism does not guarantee *sound* over-approximation of program behaviors in general. However, it is fully automatic, is scalable in terms of the size of opaque code, and provides more precise results than conventional over-approximation approaches. Our evaluation shows that although not all functionalities in opaque code can (or should) be modeled automatically using our technique, a large number of JavaScript built-in functions are approximated soundly yet more precisely than existing manual models.

Keywords: Automatic modeling · Static analysis · Opaque code · JavaScript

1 Introduction

Static analysis is widely used to optimize programs and to find bugs in them, but it often faces difficulties in analyzing library code. Since most real-world programs use various libraries usually written in different programming languages, analysis developers should provide analysis results for libraries as well. For example, static analysis of JavaScript apps involves analysis of the builtin functions implemented in host environments like the V8 runtime system written in C++.

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 43–60, 2019.
https://doi.org/10.1007/978-3-030-16722-6_3

A conventional approach to analyze such *opaque code* is for analysis developers to create models that provide the analysis results of the opaque code. Models approximate the behaviors of opaque code, they are often tightly integrated with specific static analyzers to support precise abstract semantics that are compatible with the analyzers' internals.

Developers can create models either manually or automatically. Manual modeling is complex, time consuming, and error prone because developers need to consider all the possible behaviors of the code they model. In the case of JavaScript, the number of APIs to be modeled is large and ever-growing as the language evolves. Thus, various approaches have been proposed to model opaque code automatically. They create models either from specifications of the code's behaviors [2,26] or using dynamic information during execution of the code [8,9,22]. The former approach heavily depends on the quality and format of available specifications, and the latter approach is limited to the capability of instrumentation or specific analyzers.

In this paper, we propose a novel mechanism to model the behaviors of opaque code to be used by static analysis. While existing approaches aim to create general models for the opaque code's behaviors, which can produce analysis results for all possible inputs, our approach computes specific results of opaque code during static analysis. This on-demand modeling is specific to the abstract states of a program being analyzed, and it consists of three steps: sampling, run, and abstraction. When static analysis encounters opaque code with some abstract state, our approach generates samples that are a subset of all possible inputs of the opaque code by concretizing the abstract state. After evaluating the code using the concretized values, it abstracts the results and uses it during analysis. Since the sampling generally covers only a small subset of infinitely many possible inputs to opaque code, our approach does not guarantee the soundness of the modeling results just like other automatic modeling techniques.

The sampling strategy should select well-distributed samples to explore the opaque code's behaviors as much as possible and to avoid redundant ones. Generating too few samples may miss too much behaviors, while redundant samples can cause the performance overhead. As a simple yet effective way to control the number of samples, we propose to use *combinatorial testing* [11].

We implemented the proposed automatic modeling as an extension of SAFE, a JavaScript static analyzer [13,17]. For opaque code encountered during analysis, the extension generates concrete inputs from abstract states, and executes the code dynamically using the concrete inputs via a JavaScript engine (Node.js in our implementation). Then, it abstracts the execution results using the operations provided by SAFE such as lattice-*join* and our over-approximation, and resumes the analysis.

Our paper makes the following contributions:

– We present a novel way to handle opaque code during static analysis by computing a precise on-demand model of the code using (1) input samples that represent analysis states, (2) dynamic execution, and (3) abstraction.

- We propose a combinatorial sampling strategy to efficiently generate well-distributed input samples.
- We evaluate our tool against hand-written models for large parts of JavaScript's builtin functions in terms of precision, soundness, and performance.
- Our tool revealed implementation errors in existing hand-written models, demonstrating that it can be used for automatic testing of static analyzers.

In the remainder of this paper, we present our Sample-Run-Abstract approach to model opaque code for static analysis (Sect. 2) and describe the sampling strategy (Sect. 3) we use. We then discuss our implementation and experiences of applying it to JavaScript analysis (Sect. 4), evaluate the implementation using ECMAScript 5.1 builtin functions as benchmarks (Sect. 5), discuss related work (Sect. 6), and conclude (Sect. 7).

2 Modeling via Sample-Run-Abstract

Our approach models opaque code by designing a universal model, which is able to handle arbitrary opaque code. Rather than generating a specific model for each opaque code statically, it produces a single general model, which produces results for given states using concrete semantics via dynamic execution. We call this universal model the *SRA model*.

In order to create the SRA model for a given static analyzer $\mathcal{A}$ and a dynamic executor $\mathcal{E}$, we assume the following:

- The static analyzer $\mathcal{A}$ is based on abstract interpretation [6]. It provides the abstraction function $\alpha : \wp(S) \to \widehat{S}$ and the concretization function $\gamma : \widehat{S} \to \wp(S)$ for a set of concrete states S and a set of abstract states $\widehat{S}$.
- An abstract domain forms a complete lattice, which has a partial order among its values from $\bot$(bottom) to $\top$(top).
- For a given program point $c \in C$, either $\mathcal{A}$ or $\mathcal{E}$ can identify the code corresponding to the point.

Then, the SRA model consists of the following three steps:

- $\mathit{Sample} : \widehat{S} \to \wp(S)$
  For a given abstract state $\widehat{s} \in \widehat{S}$, *Sample* chooses a finite set of elements from $\gamma(\widehat{s})$, a possible set of values for $\widehat{s}$. Because it is, in the general case, impossible to execute opaque code dynamically with all possible inputs, *Sample* should select representative elements efficiently as we discuss in the next section.
- $\mathit{Run} : C \times S \to S$
  For a given program point and a concrete state at this point, *Run* generates executable code corresponding to the point and state, executes the code, and returns the result state of the execution.
- $\mathit{Abstract} : \wp(S) \to \widehat{S}$
  For a given set of concrete states, *Abstract* produces an abstract state that encompasses the concrete states. One can apply α to each concrete state, join

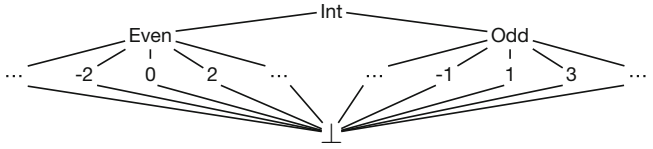


Fig. 1. An abstract domain for even and odd integers

all the resulting abstract states, and optionally apply an over-approximation heuristic, comparable to widening $\mathit{Broaden} : \widehat{S} \to \widehat{S}$ to mitigate missing behaviors of the opaque code due to the under-approximate sampling.

We write the SRA model as $\Downarrow_{SRA}: C \times \widehat{S} \to \widehat{S}$ and define it as follows:

$$\begin{aligned} \Downarrow_{SRA}(c, \widehat{s}) &= \mathit{Abstract}(\{\mathit{Run}(c, s) \mid s \in \mathit{Sample}(\widehat{s})\}) \\ &= \mathit{Broaden}(\bigsqcup\{\alpha(\{\mathit{Run}(c, s)\}) \mid s \in \mathit{Sample}(\widehat{s})\}) \end{aligned}$$

We now describe how $\Downarrow_{SRA}$ works using an example abstract domain for even and odd integers as shown in Fig. 1. Let us consider the code snippet `x := abs(x)` at a program point c where the library function `abs` is opaque. We use maps from variables to their concrete values for concrete states, maps from variables to their abstract values for abstract states, and the identity function for *Broaden* in this example.

Case $\widehat{s}_1 \equiv [\mathtt{x} : n]$ *where* n *is a constant integer:*

$$\begin{aligned} \Downarrow_{SRA}(c, \widehat{s}_1) &= \bigsqcup\{\alpha(\{\mathit{Run}(c, s)\}) \mid s \in \mathit{Sample}(\widehat{s}_1)\} \\ &= \bigsqcup\{\alpha(\{\mathit{Run}(c, s)\}) \mid s \in \{[\mathtt{x} : n]\}\} \\ &= \bigsqcup\{\alpha(\{\mathit{Run}(c, [\mathtt{x} : n])\})\} \\ &= \bigsqcup\{\alpha(\{[\mathtt{x} : |n|]\})\} \\ &= [\mathtt{x} : |n|] \end{aligned}$$

Because the given abstract state $\widehat{s}_1$ contains a single abstract value corresponding to a single concrete value, *Sample* produces the set of all possible states, which makes $\Downarrow_{SRA}$ provide a sound and also the most precise result.

Case $\widehat{s}_2 \equiv [\mathtt{x} : \mathtt{Even}]$*:*

$$\begin{aligned} \Downarrow_{SRA}(c, \widehat{s}_2) &= \bigsqcup\{\alpha(\{\mathit{Run}(c, s)\}) \mid s \in \mathit{Sample}(\widehat{s}_2)\} \\ &= \bigsqcup\{\alpha(\{\mathit{Run}(c, s)\}) \mid s \in \{[\mathtt{x} : -2], [\mathtt{x} : 0], [\mathtt{x} : 2]\}\} \\ &= \bigsqcup\{\alpha(\{[\mathtt{x} : 0], [\mathtt{x} : 2]\})\} \\ &= [\mathtt{x} : \mathtt{Even}] \end{aligned}$$

When *Sample* selects three elements from the set of all possible states represented by $\widehat{s}_2$, executing `abs` results in $\{[\mathtt{x} : 0], [\mathtt{x} : 2]\}$. Since joining these two abstract states produces `Even`, $\Downarrow_{SRA}$ models the correct behavior of `abs` by taking advantage of the abstract domain.

Case $\widehat{s}_3 \equiv [\texttt{x} : \texttt{Int}]$ *:*

$$
\begin{aligned}
&\Downarrow_{SRA}\ (c, \widehat{s}_3) \\
&= \bigsqcup\{\alpha(\{Run(c,s)\}) \mid s \in Sample(\widehat{s}_3)\} \\
&= \bigsqcup\{\alpha(\{Run(c,s)\}) \mid s \in Sample(\widehat{s}_2) \cup Sample([\texttt{x} : \texttt{Odd}])\} \\
&= \bigsqcup\{\alpha(\{Run(c,s)\}) \mid s \in \{[\texttt{x} : -2], [\texttt{x} : -1], [\texttt{x} : 0], [\texttt{x} : 1], [\texttt{x} : 2], [\texttt{x} : 3]\}\} \\
&= \bigsqcup\{\alpha(\{[\texttt{x} : 0], [\texttt{x} : 1], [\texttt{x} : 2], [\texttt{x} : 3]\})\} \\
&= [\texttt{x} : \texttt{Int}]
\end{aligned}
$$

When an abstract value has a finite number of elements that are immediately below it in the abstract domain lattice, our sampling strategy selects samples from them recursively. Thus, in this example, $Sample([\texttt{x} : \texttt{Int}])$ becomes the union of $Sample([\texttt{x} : \texttt{Even}])$ and $Sample([\texttt{x} : \texttt{Odd}])$. We explain this recursive sampling strategy in Sect. 3.

Case $\widehat{s}_4 \equiv [\texttt{x} : \texttt{Odd}]$*:*

$$
\begin{aligned}
\Downarrow_{SRA}\ (c, \widehat{s}_4) &= \bigsqcup\{\alpha(\{Run(c,s)\}) \mid s \in Sample(\widehat{s}_4)\} \\
&= \bigsqcup\{\alpha(\{Run(c,s)\}) \mid s \in \{[\texttt{x} : -1], [\texttt{x} : 1]\}\} \\
&= \bigsqcup\{\alpha(\{[\texttt{x} : 1]\})\} \\
&= [\texttt{x} : 1]
\end{aligned}
$$

While $\Downarrow_{SRA}$ produces sound and precise results for the above three cases, it does not guarantee soundness; it may miss some behaviors of opaque code due to the limitations of the sampling strategy. Let us assume that $Sample([\texttt{x} : \texttt{Odd}])$ selects $\{[\texttt{x} : -1], [\texttt{x} : 1]\}$ this time. Then, the model produces an unsound result $[\texttt{x} : 1]$, which does not cover odd integers, because the selected values explore only partial behaviors of `abs`. When the number of possible states at a call site of opaque code is infinite, the sampling strategy can lead to unsound results. A well-designed sampling strategy is crucial for our modeling approach; it affects the analysis performance and soundness significantly. The approach is precise thanks to under-approximated results from sampling, but entails a tradeoff between the analysis performance and soundness depending on the number of samples. In the next section, we propose a strategy to generate samples for various abstract domains and to control sample sizes effectively.

3 Combinatorial Sampling Strategy

We propose to use a combinatorial sampling strategy (inspired by combinatorial testing) by the types of values that an abstract domain represents. The domains represent either *primitive* values like number and string, or *object* values like tuple, set, and map. Based on combinatorial testing, our strategy is recursively defined on the hierarchy of abstract domains used to represent program states. Assume that $\widehat{a}, \widehat{b} \in \widehat{A}$ are abstract values that we want to concretize using *Sample*.

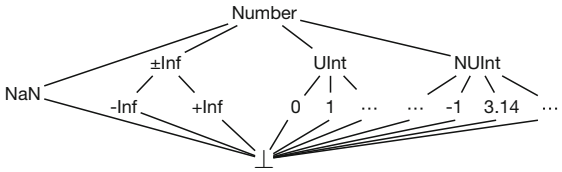


Fig. 2. The SAFE number domain for JavaScript

3.1 Abstract Domains for Primitive Values

To explain our sampling strategy for primitive abstract domains, we use the `DefaultNumber` domain from SAFE as an example. `DefaultNumber` represents JavaScript numbers with subcategories as shown in Fig. 2. The subcategories are `NaN` (not a number), ±`Inf` (positive/negative infinity), `UInt` (unsigned integer), and `NUInt` (not an unsigned integer, which is a negative integer or a floating point number).

Case $|\gamma(\widehat{a})| = constant$*:*

$$Sample(\widehat{a}) = \gamma(\widehat{a})$$

When $\widehat{a}$ represents a finite number of concrete values, *Sample* simply takes all the values. For example, ±`Inf` has two possible values, `+Inf` and `-Inf`. Therefore, $Sample(\pm\texttt{Inf}) = \{\texttt{+Inf}, \texttt{-Inf}\}$.

Case $|\gamma(\widehat{a})| = \infty$ *and* $|\{\widehat{b} \in \widehat{A} \mid \forall \widehat{x} \sqsubset \widehat{a}.\ \widehat{b} \not\sqsubset \widehat{x}\}| = constant$*:*

$$Sample(\widehat{a}) = \bigcup_{\widehat{b}} Sample(\widehat{b})$$

When $\widehat{a}$ represents an infinite number of concrete values, but it *covers* (that is, is immediately preceded by) a finite number of abstract values in the lattice, *Sample* applies to each predecessor recursively and merges the concrete results by set union. Note that, "y covers x" holds whenever $x \sqsubset y$ and there is no z such that $x \sqsubset z \sqsubset y$. The number of samples increases linearly in this step. `Number` falls into this case. It represents infinitely many numbers, but it covers four abstract values in the lattice: `NaN`, ±`Inf`, `UInt`, and `NUInt`.

Case $|\gamma(\widehat{a})| = \infty$ *and* $|\{\widehat{b} \in A \mid \forall \widehat{x} \sqsubset \widehat{a}.\ \widehat{b} \not\sqsubset \widehat{x}\}| = \infty$*:*

$$Sample(\widehat{a}) = H(\gamma(\widehat{a}))$$

When $\widehat{a}$ represents infinitely many concrete values and also covers infinitely many abstract values, we make the number of samples finite by applying a heuristic injection H of seed samples. For seed samples, we propose the following guidelines to manually select them:

- Use a small number of commonly used values. Our conjecture is that common values will trigger the same behavior in opaque code repeatedly.
- Choose values that have special properties for known operators. For example, for each operator, select the minimum, maximum, identity, and inverse elements, if any.

In the `DefaultNumber` domain example, `UInt` and `NUInt` fall into this case. For the evaluation of our modeling approach in Sect. 5, we selected seed samples based on the guidelines as follows:

$$\begin{aligned} \mathit{Sample}(\texttt{UInt}) &= \{0, 1, 3, 10, 9999\} \\ \mathit{Sample}(\texttt{NUInt}) &= \{-10, -3, -1, -0.5, -0, 0.5, 3.14\} \end{aligned}$$

We experimentally show that this simple heuristic works well for automatic modeling of JavaScript builtin functions.

3.2 Abstract Domains for Object Values

Our sampling strategy for object abstract domains consists of four steps. To sample from a given abstract object $\widehat{a} \in \widehat{A}$, we assume the following:

- A concrete object $a \in \gamma(\widehat{a})$ is a map from fields to their values: $\mathit{Map}[F, V]$.
- Abstract domains for fields and values are $\widehat{F}$ and $\widehat{V}$, respectively.
- The abstract domain $\widehat{A}$ provides two helper functions: $\mathit{mustF} : \widehat{A} \to \wp(F)$ and $\mathit{mayF} : \widehat{A} \to \widehat{F}$. The $\mathit{mustF}(\widehat{a})$ function returns a set of fields that $\forall a \in \gamma(\widehat{a})$ must have, and $\mathit{mayF}(\widehat{a})$ returns an abstract value $\widehat{f} \in \widehat{F}$ representing a set of fields that $\exists a \in \gamma(\widehat{a})$ may have.

Then, the sampling strategy follows the next four steps:

1. Sampling fields
   In order to construct sampled objects, it first samples a finite number of fields. JavaScript provides open objects, where fields can be added and removed dynamically, and fields can be referenced not only by string literals but also by arbitrary expressions of string values. Thus, this step collects fields from a finite set of fields that all possible objects should contain (F_{must}) and samples from a possibly infinite set of fields that some possible objects may (but not must) contain (F_{may}):

$$\begin{aligned} F_{must} &= \mathit{mustF}(\widehat{a}) \\ F_{may} &= \mathit{Sample}(\mathit{mayF}(\widehat{a})) \setminus F_{must} \end{aligned}$$

2. Abstracting values for the sampled fields
   For the fields in F_{must} and F_{may} sampled from the given abstract object $\widehat{a}$, it constructs two maps from fields to their abstract values, M_{must} and M_{may}, respectively, of type $\mathit{Map}[F, \widehat{V}]$:

$$\begin{aligned} M_{must} &= \lambda f \in F_{must}.\ \alpha(\{a(f) \mid a \in \gamma(\widehat{a})\}) \\ M_{may} &= \lambda f \in F_{may}\ .\ \alpha(\{a(f) \mid a \in \gamma(\widehat{a})\}) \end{aligned}$$

3. Sampling values
   From M_{must} and M_{may}, it constructs another map $M_s : F \to \wp(V_{\nexists})$, where $V_{\nexists} = V \cup \{\nexists\}$ denotes a set of values and the absence of a field $\nexists$, by applying *Sample* to the value of each field in F_{must} and F_{may}. The value of each field in F_{may} contains $\nexists$ to denote that the field may not exist in M_s:

$$M_s = \lambda f \in F_{must} \cup F_{may}. \begin{cases} Sample(M_{must}(f)) & \text{if } f \in F_{must} \\ Sample(M_{may}(f)) \cup \{\sharp\} & \text{if } f \in F_{may} \end{cases}$$

4. Choosing samples by combinatorial testing
   Finally, since a number of all combinations from M_s, $\prod_{f \in Domain(M_s)} |M_s(f)|$, grows exponentially, the last step limits the number selections. We solve this selection problem by reducing it to a traditional testing problem with combinatorial testing [3]. Combinatorial testing is a well-studied problem and efficient algorithms for generating test cases exist. It addresses a similar problem to ours, increasing dynamic coverage of code under test, but in the context of finding bugs:

   > "The most common bugs in a program are generally triggered by either a single input parameter or an interaction between pairs of parameters."

   Thus, we apply each-used or pair-wise testing (1 or 2-wise) as the last step.

Now, we demonstrate each step using an abstract array object $\widehat{a}$, whose length is greater than or equal to 2 and the elements of which are `true` or `false`. We write $\top_b$ to denote an abstract value such that $\gamma(\top_b) = \{\texttt{true}, \texttt{false}\}$.

- Assumptions
  - A concrete array object a is a map from indices to boolean values: $Map[\texttt{UInt}, \texttt{Boolean}]$.
  - For given abstract object $\widehat{a}$, $mustF(\widehat{a}) = \{0, 1\}$ and $mayF(\widehat{a}) = \texttt{UInt}$.
  - From Sect. 3.1, we sample $\{0, 1, 3, 10, 9999\}$ for `UInt`.
  - $k\text{-}wise(M)$ generates a set of minimum number of test cases satisfying all the requirements of $k\text{-}wise$ testing for a map M. It constructs a test case by choosing one element from a set on each field.
- Step 1: Sampling fields

$$\begin{aligned} F_{must} &= \{0, 1\} \\ F_{may} &= Sample(\texttt{UInt}) \setminus \{0, 1\} = \{3, 10, 9999\} \end{aligned}$$

- Step 2: Abstracting values for the sampled fields

$$\begin{aligned} M_{must} &= [0 \mapsto \top_b, 1 \mapsto \top_b] \\ M_{may} &= [3 \mapsto \top_b, 10 \mapsto \top_b, 9999 \mapsto \top_b] \end{aligned}$$

- Step 3: Sampling values

$$\begin{aligned} M_s = [\quad & 0 \mapsto \{\texttt{true}, \texttt{false}\}, \quad 1 \mapsto \{\texttt{true}, \texttt{false}\}, \\ & 3 \mapsto \{\texttt{true}, \texttt{false}, \sharp\}, 10 \mapsto \{\texttt{true}, \texttt{false}, \sharp\}, \\ & 9999 \mapsto \{\texttt{true}, \texttt{false}, \sharp\} \qquad\qquad\qquad\qquad\qquad\quad] \end{aligned}$$

- Step 4: Choosing samples by combinatorial testing
  The number of all combinations $\prod_{f \in Domain(M_s)} |M_s(f)|$ is 108 even after sampling fields and values in an under-approximate manner. We can avoid such

explosion of samples and manage well-distributed samples by using combinatorial testing. With each-used testing, three combinations can cover every element in a set on each field at least once:

$$
\begin{aligned}
&1\text{-}wise(M_s) = \\
&\quad \{\ [0 \mapsto \texttt{true},\ \ 1 \mapsto \texttt{false}, 3 \mapsto \texttt{true},\ \ 10 \mapsto \sharp,\ \ \ \ 9999 \mapsto \sharp], \\
&\quad\ \ [0 \mapsto \texttt{false}, 1 \mapsto \texttt{true},\ \ 3 \mapsto \texttt{false}, 10 \mapsto \texttt{false}, 9999 \mapsto \texttt{true}], \\
&\quad\ \ [0 \mapsto \texttt{false}, 1 \mapsto \texttt{true},\ \ 3 \mapsto \sharp,\ \ \ \ 10 \mapsto \texttt{true},\ \ 9999 \mapsto \texttt{false}]\ \}
\end{aligned}
$$

With pair-wise testing, 12 samples can cover every pair of elements from different sets at least once.

4 Implementation

We implemented our automatic modeling approach for JavaScript because of its large number of builtin APIs and complex libraries, which are all opaque code for static analysis. They include the functions in the ECMAScript language standard [1] and web standards such as DOM and browser APIs. We implemented the modeling as an extension of SAFE [13,17], a JavaScript static analyzer. When the analyzer encounters calls of opaque code during analysis, it uses the SRA model of the code.

Sample. We applied the combinatorial sampling strategy for the SAFE abstract domains. Of the abstract domains for primitive JavaScript values, `UInt`, `NUInt`, and `OtherStr` represent an infinite number of concrete values (c.f. third case in Sect. 3.1) and thus require the use of heuristics. We describe the details of our heuristics and sample sets in Sect. 5.1.

We implemented the *Sample* step to use "each-used sample generation" for object abstract domains by default. In order to generate more samples, we added three options to apply pair-wise generation:

- `ThisPair` generates pairs between the values of `this` and heap,
- `HeapPair` among objects in the heap, and
- `ArgPair` among property values in an `arguments` object.

As an exception, we use the all-combination strategy for the `DefaultDataProp` domain representing a JavaScript property, consisting of a value and three booleans: `writable`, `enumerable`, and `configurable`. Note that *field* is used for language-independent objects and *property* is for JavaScript objects. The number of their combinations is limited to 2^3. We consider a linear increase of samples as acceptable. The *Sample* step returns a finite set of concrete states, and each element in the set, which in turn contains concrete values only, is passed to the *Run* step.

Run. For each concrete input state, the *Run* step obtains a result state by executing the corresponding opaque code in four steps:

1. Generation of executable code
   First, *Run* populates object values from the concrete state. We currently omit the JavaScript scope-chain information, because the library functions that we analyze as opaque code are independent from the scope of user code. It derives executable code to invoke the opaque code and adds argument values from the static analysis context.
2. Execution of the code using a JavaScript engine
   Run executes the generated code using the JavaScript `eval` function on Node.js. Populating objects and their properties from sample values before invoking the opaque function may throws an exception. In such cases, *Run* executes the code once again with a different sample value. If the second sample value also throws an exception during population of the objects and their properties, it dismisses the code.
3. Serialization of the result state
   After execution, the result state contains the objects from the input state, the return value of the opaque code, and all the values that it might refer to. Also, any mutation of objects of the input state as well as newly created objects are captured in this way. We use a snapshot module of SAFE to serialize the result state into a JSON-like format.
4. Transfer of the state to the analyzer
   The serialized snapshot is then passed to SAFE, where it is parsed, loaded, and combined with other results as a set of concrete result states.

Abstract. To abstract result states, we mostly used existing operations in SAFE, like lattice-*join*, and also implemented an over-approximation heuristic function, *Broaden*, comparable to widening. We use *Broaden* for property name sets in JavaScript objects, because *mayF* of a JavaScript abstract object can produce an abstract value that denotes an infinite set of concrete strings, and because $\Downarrow_{SRA}$ cannot produce such an abstract value from simple sampling and *join*. Thus, we regard all possibly absent properties as sampled properties. Then, we implemented the *Broaden* function merging all possibly absent properties into one abstract property representing any property, when the number of absent properties is greater than a certain threshold proportional to a number of sampled properties.

5 Evaluation

We evaluated the $\Downarrow_{SRA}$ model in two regards, (1) the feasibility of replacing existing manual models (RQ1 and RQ2) and (2) the effects of our heuristic H on the analysis soundness (RQ3). The research questions are as follow:

- **RQ1: Analysis performance of $\Downarrow_{SRA}$**
  Can $\Downarrow_{SRA}$ replace existing manual models for program analysis with decent performance in terms of soundness, precision, and runtime overhead?

- **RQ2: Applicability of $\Downarrow_{SRA}$**
  Is $\Downarrow_{SRA}$ broadly applicable to various builtin functions of JavaScript?
- **RQ3: Dependence on heuristic H**
  How much is the performance of $\Downarrow_{SRA}$ affected by the heuristics?

After describing the experimental setup for evaluation, we present our answers to the research questions with quantitative results, and discuss the limitations of our evaluation.

5.1 Experimental Setup

In order to evaluate the $\Downarrow_{SRA}$ model, we compared the analysis performance and applicability of $\Downarrow_{SRA}$ with those of the existing manual models in SAFE. We used two kinds of subjects: browser benchmark programs and builtin functions. From 34 browser benchmarks included in the test suite of SAFE, a subset of V8 Octane[1], we collected 13 of them that invoke opaque code. Since browser benchmark programs use a small number of opaque functions, we also generated test cases for 134 functions in the ECMAScript 5.1 specification.

Each test case contains abstract values that represent two or more possible values. Because SAFE uses a finite number of abstract domains for primitive values, we used all of them in the test cases. We also generated 10 abstract objects. Five of them are manually created to represent arbitrary objects:

`OBJ1` has an arbitrary property whose value is an arbitrary primitive.
`OBJ2` is a property descriptor whose `"value"` is an arbitrary primitive, and the others are arbitrary booleans.
`OBJ3` has an arbitrary property whose value is `OBJ2`.
`OBJ4` is an empty array whose `"length"` is arbitrary.
`OBJ5` is an arbitrary-length array with an arbitrary property

The other five objects were collected from SunSpider benchmark programs by using Jalangi2 [20] to represent frequently used abstract objects. We counted the number of function calls with object arguments and joined the most used object arguments in each program. Out of 10 programs that have function calls with object arguments, we discarded four programs that use the same objects for every function call, and one program that uses an argument with 2500 properties, which makes manual inspection impossible. We joined the first 10 concrete objects for each argument of the following benchmark to obtain abstract objects: 3d-cube.js, 3d-raytrace.js, access-binary-trees.js, regexp-dna.js, and string-fasta.js. For 134 test functions, when a test function consumes two or more arguments, we restricted each argument to have only an expected type to manage the number of test cases. Also, we used one or minimum number of arguments for functions with variable number of arguments.

In summary, we used 13 programs for RQ1, and 134 functions with 1565 test cases for RQ2 and RQ3. All experiments were on a 2.9 GHz quad-core Intel Core i7 with 16 GB memory machine.

[1] https://github.com/chromium/octane.

5.2 Answers to Research Questions

Answer to RQ1. We compared the precision, soundness, and analysis time of the SAFE manual models and the $\Downarrow_{SRA}$ model. Table 1 shows the precision and soundness for each opaque function call, and Table 2 presents the analysis time and number of samples for each program.

As for the precision, Table 1 shows that $\Downarrow_{SRA}$ produced more precise results than manual models for 9 (19.6%) cases. We manually checked whether each result of a model is sound or not by using the partial order function ($\sqsubseteq$) implemented in SAFE. We found that all the results of the SAFE manual models for the benchmarks were sound. The $\Downarrow_{SRA}$ model produced an unsound result for only one function: `Math.random`. While it returns a floating-point value in the range $[0, 1)$, $\Downarrow_{SRA}$ modeled it as `NUInt`, instead of the expected `Number`, because it missed 0.

As shown in Table 2, on average $\Downarrow_{SRA}$ took 1.35 times more analysis time than the SAFE models. The table also shows the number of context-sensitive opaque function calls during analysis (#Call), the maximum number of samples (#Max), and the total number of samples (#Total). To understand the runtime overhead better, we measured the proportion of elapsed time for each step. On average, *Sample* took 59%, *Run* 7%, *Abstract* 17%, and the rest 17%. The experimental results show that $\Downarrow_{SRA}$ provides high precision while slightly sacrificing soundness with modest runtime overhead.

Answer to RQ2. Because the benchmark programs use only 15 opaque functions as shown in Table 1, we generated abstracted arguments for 134 functions out of 169 functions in the ECMAScript 5.1 builtin library, for which SAFE has manual models. We semi-automatically checked the soundness and precision of the $\Downarrow_{SRA}$ model by comparing the analysis results with their expected results. Table 3 shows the results in terms of test cases (left half) and functions (right half). The **Equal** column shows the number of test cases or functions, for which both models provide equal results that are sound. The **SRA Pre.** column shows the number of such cases where the $\Downarrow_{SRA}$ model provides sound and more precise results than the manual model. The **Man. Uns.** column presents the number of such cases where $\Downarrow_{SRA}$ provides sound results but the manual one provides unsound results, and **SRA Uns.** shows the opposite case of **Man. Uns.** Finally, **Not Comp.** shows the number of cases where the results of $\Downarrow_{SRA}$ and the manual model are incomparable.

The $\Downarrow_{SRA}$ model produced sound results for 99.4% of test cases and 94.0% of functions. Moreover, $\Downarrow_{SRA}$ produced more precise results than the manual models for 33.7% of test cases and 50.0% of functions. Although $\Downarrow_{SRA}$ produced unsound results for 0.6% of test cases and 6.0% of functions, we found soundness bugs in the manual models using 1.3% of test cases and 7.5% of functions. Our experiments showed that the automatic $\Downarrow_{SRA}$ model produced less unsound results than the manual models. We reported the manual models producing unsound results to SAFE developers with the concrete examples that were generated in the *Run* step, which revealed the bugs.

Table 1. Precision and soundness by functions in the benchmarks

Function	Precision and Soundness		
	Equal Precise	More Precise	Unsound
`Array`, `Array.prototype.join`, `Array.prototype.push`	15	5	0
`Date`, `Date.prototype.getTime`	0	4	0
`Error`	5	0	0
`Math.cos`, `Math.max`, `Math.pow`, `Math.sin`, `Math.sqrt`	11	0	0
`Math.random`	0	0	1
`Number.prototype.toString`	1	0	0
`String`, `String.prototype.substring`	4	0	0
Total	36	9	1
Proportion	78.3%	19.6%	2.2%

Table 2. Analysis time overhead by programs in the benchmarks

Program	Manual		$\Downarrow_{SRA}$				Increased
	Time(ms)	#Call	Time(ms)	#Call	#Max	#Total	Time Ratio
3d-morph.js	1,423	50	2,641	50	16	408	1.86
access-binary-trees.js	1,926,132	10	1,784,866	10	16	95	0.93
access-fannkuch.js	1,615	31	2,627	31	15	413	1.63
access-nbody.js	10,125	132	25,564	324	16	4,274	2.52
access-nsieve.js	1,019	6	1,126	6	16	54	1.10
bitops-nsieve-bits.js	282	1	343	1	2	2	1.22
math-cordic.js	574	2	662	2	2	4	1.15
math-partial-sums.js	1,613	99	4,703	99	16	916	2.92
math-spectral-norm.js	10,702	6	10,986	6	16	96	1.03
string-fasta.js	22,170	78	6,147	30	226	2,555	0.28
navier-stokes.js	4,662	20	5,104	20	2	40	1.09
richards.js	86,013	85	88,902	85	54	4,018	1.03
splay.js	259,073	423	217,863	422	56	11,492	0.84
Total	2,325,404	943	2,151,533	1,086	453	24,367	1.35

Answer to RQ3. The sampling strategy plays an important role in the performance of $\Downarrow_{SRA}$ especially for soundness. Our sampling strategy depends on two factors: (1) manually sampled sets via the heuristic H and (2) each-used or pair-wise selection for object samples. We used manually sampled sets for three abstract values: `UInt`, `NUInt`, and `OtherStr`. To sample concrete values from them, we used three methods: Base simply follows the guidelines described in Sect. 3.1, Random generates samples randomly, and Final denotes the heuristics determined by our trials and errors to reach the highest ratio of sound results. For object samples, we used three pair-wise options: HeapPair, ThisPair, and ArgPair. For various sampling configurations, Table 4 summarizes the ratio of sound

Table 3. Precision and soundness for the builtin functions

Object	#Test Case							#Function						
	Equal	SRA Pre.	Man. Uns.	Man. Pre.	SRA Uns.	Not Comp.	Total	Equal	SRA Pre.	Man. Uns.	Man. Pre.	SRA Uns.	Not Comp.	Total
`Array`	59	144	1	0	0	0	174	8	7	1	0	0	0	16
`Boolean`	37	2	3	0	0	0	42	1	0	3	0	0	0	4
`Date`	74	241	0	2	1	1	319	8	35	0	2	1	1	47
`Global`	7	1	0	0	0	0	8	1	1	0	0	0	0	2
`Math`	106	5	0	0	6	0	117	11	2	0	0	5	1	18
`Number`	41	71	0	3	0	1	116	1	6	0	0	0	0	8
`Object`	370	24	7	1	3	5	410	12	2	5	0	2	0	21
`String`	300	70	9	0	0	0	379	3	14	1	0	0	0	18
Total	994	528	20	6	10	7	1565	45	67	10	2	8	2	134
Proportion	63.5%	33.7%	1.3%	0.4%	0.6%	0.4%	100%	33.6%	50.0%	7.5%	1.5%	6.0%	1.5%	100%

Table 4. Soundness and sampling cost for the builtin functions

Sampling Configuration						Builtin Function		
Set Heuristic			Pair Option			Sound Result Ratio	#Ave.	#Max
`UInt`	`NUInt`	`Other`	HeapPair	ThisPair	ArgPair			
Base	Base	Base	F	F	F	85.0%	17.4	41
Random	Random	Random	F	F	F	84.9%	17.4	41
Final	Final	Final	F	F	F	92.1%	32.6	98
			F	F	T	93.5%	38.1	226
			F	T	F	95.0%	181.9	4312
			F	T	T	95.5%	276.8	11752
			T	F	F	96.2%	323.0	7220
			T	F	T	97.4%	397.5	16498
			T	T	F	99.2%	513.7	11988
			T	T	T	99.4%	677.6	16498

results, the average and maximum numbers of samples for the test cases used in RQ2.

The table shows that Base and Random produced sound results for 85.0% and 84.9% (the worst case among 10 repetitions) of the test cases, respectively. Even without any sophisticated heuristics or pair-wise options, $\Downarrow_{SRA}$ achieved a decent amount of sound results. Using more samples collected by trials and errors with Final and all three pair-wise options, $\Downarrow_{SRA}$ generated sound results for 99.4% of the test cases by observing more behaviors of opaque code.

5.3 Limitations

A fundamental limitation of our approach is that the $\Downarrow_{SRA}$ model may produce unsound results when the behavior of opaque code depends on values that $\Downarrow_{SRA}$ does not support via sampling. For example, if a sampling strategy calls the `Date` function without enough time intervals, it may not be able to sample different

results. Similarly, if a sampling strategy does not use 4-wise combinations for property descriptor objects that have four components, it cannot produce all the possible combinations. However, at the same time, simply applying more complex strategies like 4-wise combinations may lead to an explosion of samples, which is not scalable.

Our experimental evaluation is inherently limited to a specific use case, which poses a threat to validity. While our approach itself is not dependent on a particular programming language or static analysis, the implementation of our approach depends on the abstract domains of SAFE. Although the experiments used well-known benchmark programs as analysis subjects, they may not be representative of all common uses of opaque functions in JavaScript applications.

6 Related Work

When a textual specification or documentation is available for opaque code, one can generate semantic models by mining them. Zhai *et al.* [26] showed that natural language processing can successfully generate models for Java library functions and used them in the context of taint analysis for Android applications. Researchers also created models automatically from types written in WebIDL or TypeScript declarations to detect Web API misuses [2,16].

Given an executable (e.g. binary) version of opaque code, researchers also synthesized code by sampling the inputs and outputs of the code [7,10,12,19]. Heule *et al.* [8] collected partial execution traces, which capture the effects of opaque code on user objects, followed by code synthesis to generate models from these traces. This approach works in the absence of any specification and has been demonstrated on array-manipulating builtins.

While all of these techniques are a-priori attempts to generate general-purpose models of opaque code, to be usable for other analyses, researchers also proposed to construct models during analysis. Madsen *et al.*'s approach [14] infers models of opaque functions by combining pointer analysis and use analysis, which collects expected properties and their types from given application code. Hirzel *et al.* [9] proposed an online pointer analysis for Java, which handles native code and reflection via dynamic execution that ours also utilizes. While both approaches use only a finite set of pointers as their abstract values, ignoring primitive values, our technique generalizes such online approaches to be usable for all kinds of values in a given language.

Opaque code does matter in other program analyses as well such as model checking and symbolic execution. Shafiei and Breugel [22] proposed *jpf-nhandler*, an extension of Java PathFinder (JPF), which transfers execution between JPF and the host JVM by on-the-fly code generation. It does not need concretization and abstraction since a JPF object represents a concrete value. In the context of symbolic execution, concolic testing [21] and other hybrid techniques that combine path solving with random testing [18] have been used to overcome the problems posed by opaque code, albeit sacrificing completeness [4].

Even when source code of external libraries is available, substituting external code with models rather than analyzing themselves is useful to reduce time

and memory that an analysis takes. Palepu *et al.* [15] generated summaries by abstracting concrete data dependencies of library functions observed on a training execution to avoid heavy execution of instrumented code. In model checking, Tkachuk *et al.* [24,25] generated over-approximated summaries of environments by points-to and side-effect analyses and presented a static analysis tool OCSEGen [23]. Another tool Modgen [5] applies a program slicing technique to reduce complexities of library classes.

7 Conclusion

Creating semantic models for static analysis by hand is complex, time-consuming and error-prone. We present a Sample-Run-Abstract approach ($\Downarrow_{SRA}$) as a promising way to perform static analysis in the presence of opaque code using automated on-demand modeling. We show how $\Downarrow_{SRA}$ can be applied to the abstract domains of an existing JavaScript static analyzer, SAFE. For benchmark programs and 134 builtin functions with 1565 abstracted inputs, a tuned $\Downarrow_{SRA}$ produced more sound results than the manual models and concrete examples revealing bugs in the manual models. Although not all opaque code may be suitable for modeling with $\Downarrow_{SRA}$, it reduces the amount of hand-written models a static analyzer should provide. Future work on $\Downarrow_{SRA}$ could focus on orthogonal testing techniques that can be used for sampling complex objects, and practical optimizations, such as caching of computed model results.

Acknowledgment. This work has received funding from National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

References

1. ECMAScript Language Specification. Edition 5.1. http://www.ecma-international.org/publications/standards/Ecma-262.htm
2. Bae, S., Cho, H., Lim, I., Ryu, S.: SAFEWAPI: web API misuse detector for web applications. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 507–517. ACM (2014)
3. Black, R.: Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional. Wiley, Hoboken (2007)
4. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
5. Ceccarello, M., Tkachuk, O.: Automated generation of model classes for Java PathFinder. ACM SIGSOFT Softw. Eng. Notes **39**(1), 1–5 (2014)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM (1977)
7. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Commun. ACM **55**(8), 97–105 (2012)

8. Heule, S., Sridharan, M., Chandra, S.: Mimic: computing models for opaque code. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 710–720. ACM (2015)
9. Hirzel, M., Dincklage, D.V., Diwan, A., Hind, M.: Fast online pointer analysis. ACM Trans. Program. Lang. Syst. (TOPLAS) **29**(2), 11 (2007)
10. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 215–224. ACM (2010)
11. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. **30**(6), 418–421 (2004)
12. Lau, T., Domingos, P., Weld, D.S.: Learning programs from traces using version space algebra. In: Proceedings of the 2nd International Conference on Knowledge Capture, pp. 36–43. ACM (2003)
13. Lee, H., Won, S., Jin, J., Cho, J., Ryu, S.: SAFE: formal specification and implementation of a scalable analysis framework for ECMAScript. In: FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages, p. 96. Citeseer (2012)
14. Madsen, M., Livshits, B., Fanning, M.: Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 499–509. ACM (2013)
15. Palepu, V.K., Xu, G., Jones, J.A.: Improving efficiency of dynamic analysis with dynamic dependence summaries. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 59–69. IEEE Press (2013)
16. Park, J.: JavaScript API misuse detection by using TypeScript. In: Proceedings of the Companion Publication of the 13th International Conference on Modularity, pp. 11–12. ACM (2014)
17. Park, J., Ryou, Y., Park, J., Ryu, S.: Analysis of JavaScript web applications using SAFE 2.0. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 59–62. IEEE (2017)
18. Păsăreanu, C.S., Rungta, N., Visser, W.: Symbolic execution with mixed concrete-symbolic solving. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 34–44. ACM (2011)
19. Qi, D., Sumner, W.N., Qin, F., Zheng, M., Zhang, X., Roychoudhury, A.: Modeling software execution environment. In: 2012 19th Working Conference on Reverse Engineering (WCRE), pp. 415–424. IEEE (2012)
20. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 488–498. ACM (2013)
21. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ACM SIGSOFT Software Engineering Notes, vol. 30, pp. 263–272. ACM (2005)
22. Shafiei, N., Breugel, F.V.: Automatic handling of native methods in Java PathFinder. In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, pp. 97–100. ACM (2014)
23. Tkachuk, O.: OCSEGen: open components and systems environment generator. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis, pp. 9–12. ACM (2013)
24. Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking, vol. 28. ACM (2003)

25. Tkachuk, O., Dwyer, M.B., Pasareanu, C.S.: Automated environment generation for software model checking. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, pp. 116–127. IEEE (2003)
26. Zhai, J., Huang, J., Ma, S., Zhang, X., Tan, L., Zhao, J., Qin, F.: Automatic model generation from documentation for Java API functions. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 380–391. IEEE (2016)

SMT-Based Bounded Schedulability Analysis of the Clock Constraint Specification Language

Min Zhang[1], Fu Song[2(✉)], Frédéric Mallet[3], and Xiaohong Chen[1]

[1] Shanghai Key Laboratory of Trustworthy Computing, ECNU, Shanghai, China
[2] ShanghaiTech University, Shanghai, China
songfu@shanghaitech.edu.cn
[3] Université Cote d'Azur, CNRS, Inria, I3S, Nice, France

Abstract. The Clock Constraint Specification Language (CCSL) is a formalism for specifying logical-time constraints on events for the design of real-time embedded systems. A central verification problem of CCSL is to check whether events are schedulable under logical constraints. Although many efforts have been made addressing this problem, the problem is still open. In this paper, we show that the bounded scheduling problem is NP-complete and then propose an efficient SMT-based decision procedure which is sound and complete. Based on this decision procedure, we present a sound algorithm for the general scheduling problem. We implement our algorithm in a prototype tool and illustrate its utility in schedulability analysis in designing real-world systems and automatic proving of algebraic properties of CCSL constraints. Experimental results demonstrate its effectiveness and efficiency.

Keywords: SMT · CCSL · Schedulability · Logical time · Real-time system

1 Introduction

Model-based design has been widely used, particularly in the design of safety-critical real-time embedded systems. It has achieved industrial successes through languages such as SCADE [12], AADL [15] and UML MARTE [26]. For example, UML MARTE provides syntactic annotations to implement, when the context allows, classical real-time scheduling algorithms such as EDF (Earliest Deadline First). It also provides a domain-specific language–Clock Constraint Specification Language (CCSL) [3], to express the real-time behaviors of a system under development as logical constraints on system events, but independently of any physical time and classical real-time scheduling algorithms. CCSL has been used on several industrial scenarios such as vehicle systems [16] and cyber-physical systems [10,22].

This work is supported by NSFC grants 61872146, 61532019 and 61761136011.

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 61–78, 2019.
https://doi.org/10.1007/978-3-030-16722-6_4

Model-based design usually starts with coarse-grained logical models that are progressively refined into more concrete ones until the final code deployment. It is well-known that the earlier one can detect and fix bugs in the refinement process, the better [7]. Therefore, it is critical to provide efficient methods and tools to check safety, liveness and schedulability on the logical models and not only on the definite deployed system. This has motivated a large body of works on verifying whether events are schedulable under a set of constraints expressed in CCSL [11,21,28,33,35,36,38], though its decidability is still open. These works first transform CCSL constraints into other formal representations such as transition systems [21], Promela [35], Büchi automata [36], timed automata [33], rewriting logics [38], instant relations [28], or timed-interval logics [11], and then apply existing tools. However, their approaches usually suffer from the state explosion problem. Moreover, most of these works only deal with the so-called safe subset of CCSL and the other ones only provide semi-algorithms. In our earlier work [39], we proposed an SMT-based verification approach to CCSL and demonstrated several applications of the approach to finding schedules, verifying temporal properties, proving constraint entailment, and analyzing the validity of system traces. Based on the approach, we implemented an efficient tool for verifying LTL properties of CCSL [40].

In this work we are focused on the scheduling problem of CCSL, a fundamental problem to which the aforementioned verification problems of CCSL can be reduced. We first prove that the *bounded* scheduling problem of CCSL with fixed bounds is NP-complete. To our knowledge, this is the first result regarding the complexity of the scheduling problem with CCSL. Then, we propose a decision procedure for the bounded scheduling problem with a given bound. The decision procedure is based on the transformation of CCSL into SMT formulas [39]. Our decision procedure is sound, complete, and efficient in practice. Based on this decision procedure, we turn to the general (i.e. unbounded) scheduling problem and present a binary-search based algorithm. Our algorithm is sound, i.e., if it proves either schedulable or unschedulable, then the result is conclusive. We implemented our algorithms in a prototype tool. The tool was used to analyze a real-world interlocking system in a rail transit system. Using the proposed approach, we also prove some algebraic properties of CCSL. The experimental results demonstrate the effectiveness and efficiency of the SMT-based approach.

The rest of this paper is organised as follows: Section 2 introduces CCSL. Section 3 defines the (bounded) scheduling problem of CCSL and shows that the bounded case is NP-complete. Section 4 presents an SMT-based decision procedure for the bounded scheduling problem and a sound algorithm for the general scheduling problem. Section 5 shows a case study and experimental results. Section 6 discusses related work, and Section 7 concludes the paper.

2 The Clock Constraint Specification Language

2.1 Logical Clock, History and Schedule

In CCSL, clocks are used to model occurrences of events, where a clock ticks when the corresponding event occurs. For instance, a clock may represent an

event that is dispatch of a task, communications between tasks or acquisition of a shared resource by a task. Constraints over clocks are used to specify causal and temporal relations between system events. No global physical time is presumed for the clocks and their constraints. This feature allows CCSL to define a polychronous specification of a system at a logical level.

Definition 1 (Logical clock). *A* (logical) clock *c is an infinite sequence of ticks $(c^i)_{i\in\mathbb{N}^+}$ with each c^i being* tick *or* idle*, where $\mathbb{N}^+$ denotes the set of all the non-zero natural numbers.*

The value of c^i denotes whether an event associated with c occurs or not at step i. If c^i is *tick*, then the event occurs, otherwise not. In particular, we denote by **1** a global reference logical clock that always ticks at each step.

Definition 2 (Schedule). *Given a set C of clocks, a* schedule *of C is a total function $\delta : \mathbb{N}^+ \to 2^C$ such that $\forall i \in \mathbb{N}^+$, $\delta(i) = \{c \in C \mid c^i = tick\}$ and $\delta(i) \neq \emptyset$.*

Intuitively, a schedule δ defines a partial order between the ticks of the clocks. $\delta(i)$ is a subset of C such that $c \in \delta(i)$ iff c ticks at step i. The condition $\delta(i) \neq \emptyset$ expresses that step i cannot be empty. This forbids stuttering steps in schedules. As one can add or remove finite number of empty steps without effect on schedulability, we exclude them from schedules for succinctness.

A clock can memorize the number of ticks that it has made. We use *history* to represent the memorization.

Definition 3 (History). *Given a schedule δ for a set C of clocks, a* history *of δ is a function $\chi_\delta : C \times \mathbb{N}^+ \to \mathbb{N}$ such that for each $c \in C$ and $i \in \mathbb{N}^+$:*

$$\chi_\delta(c,i) = \begin{cases} 0, & \text{if } i = 1; \\ \chi_\delta(c,i-1), & \text{if } i > 1 \wedge c \notin \delta(i-1); \\ \chi_\delta(c,i-1)+1, & \text{if } i > 1 \wedge c \in \delta(i-1). \end{cases}$$

$\chi_\delta(c,i)$ represents the number of the ticks that the clock c has made immediately before step i. (Note that the tick of c at step i is excluded in $\chi_\delta(c,i)$.) For simplicity, we may write χ for χ_δ if it is clear from the context.

2.2 Syntax and Semantics of CCSL

CCSL consists of 11 kinds of constraints, 4 of them are binary relations for specifying the *precedence*, *causality*, *subclocking*, and *exclusion* relations between clocks, and the others are used to define clocks from existing ones. Clocks defined by constraints may correspond to system events or are just introduced as auxiliary clocks without corresponding to any events.

Table 1. Semantics of CCSL with respect to schedules

	ϕ	$\delta \models \phi$
Precedence	$c_1\ [b] \prec c_2$	$\forall n \in \mathbb{N}^+.\chi(c_2,n) - \chi(c_1,n) = b \Rightarrow c_2 \notin \delta(n)$
Causality	$c_1 \preccurlyeq c_2$	$\forall n \in \mathbb{N}^+.\chi(c_1,n) \geq \chi(c_2,n)$
Subclock	$c_1 \subseteq c_2$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Rightarrow c_2 \in \delta(n)$
Exclusion	$c_1 \# c_2$	$\forall n \in \mathbb{N}^+.c_1 \notin \delta(n) \vee c_2 \notin \delta(n)$
Union	$c_1 \triangleq c_2 + c_3$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow c_2 \in \delta(n) \vee c_3 \in \delta(n)$
Intersection	$c_1 \triangleq c_2 * c_3$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow c_2 \in \delta(n) \wedge c_3 \in \delta(n)$
Infimum	$c_1 \triangleq c_2 \wedge c_3$	$\forall n \in \mathbb{N}^+.\chi(c_1,n) = \max(\chi(c_2,n), \chi(c_3,n))$
Supremum	$c_1 \triangleq c_2 \vee c_3$	$\forall n \in \mathbb{N}^+.\chi(c_1,n) = \min(\chi(c_2,n), \chi(c_3,n))$
Periodicity	$c_1 \triangleq c_2 \propto p$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow (c_2 \in \delta(n) \wedge \exists m \in \mathbb{N}^+.\chi(c_2,n) = m \times p - 1)$
Filtering	$c_1 \triangleq c_2 \blacktriangledown w$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow (c_2 \in \delta(n) \wedge w[n])$
DelayFor	$c_1 \triangleq c_2\ \$\ d\ on\ c_3$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow (c_3 \in \delta(n) \wedge \exists m \in \mathbb{N}^+.(c_2 \in \delta(m) \wedge \chi(c_3,n) - \chi(c_3,m) = d))$

Definition 4 (Syntax). *A* CCSL *constraint ϕ is defined by the following form:*

Precedence:	$c_1\ [b] \prec c_2$	*Causality:*	$c_1 \preccurlyeq c_2$
Subclock:	$c_1 \subseteq c_2$	*Exclusion:*	$c_1 \# c_2$
Union:	$c_1 \triangleq c_2 + c_3$	*Intersection:*	$c_1 \triangleq c_2 * c_3$
Infimum:	$c_1 \triangleq c_2 \wedge c_3$	*Supremum:*	$c_1 \triangleq c_2 \vee c_3$
Periodicity:	$c_1 \triangleq c_2 \propto p$	*Filtering:*	$c_1 \triangleq c_2 \blacktriangledown w$
DelayFor:	$c_1 \triangleq c_2\ \$\ d\ on\ c_3$		

where $b \geq 0$, $d \geq 0$ and $p > 0$ are natural numbers, c_1, c_2, c_3 are logical clocks and w is a (possibly infinite) word over $\{0,1\}$ expressed as a (ω-)regular expression.

For simplifying presentation, we denote by $c_1 \prec c_2$ the constraint $c_1\ [0] \prec c_2$, and $c_1 \triangleq c_2\ \$\ d$ the constraint $c_1 \triangleq c_2\ \$\ d\ on\ c_3$ such that $c_2 = c_3$.

The semantics of CCSL constraints is defined over schedules. Given a CCSL constraint ϕ and a schedule δ, the satisfiability relation $\delta \models \phi$ (i.e., δ satisfies constraint ϕ) is defined in Table 1.

The precedence constraint $c_1 \prec c_2$ (i.e., $c_1\ [0] \prec c_2$) expresses that the clock c_1 precedes the clock c_2. Suppose there is an unbounded buffer with two operations *fetch* and *store*, which respectively fetch data from and store data into the buffer. Fetch is only allowed when the buffer is nonempty. If the buffer is initially empty, store operation must strictly precede fetch operation. This behavior can be expressed by the constraint: *store* $\prec$ *fetch*. Likewise, the precedence constraint can be used to represent reentrant tasks by replacing *store* with *start* and *fetch* with *finish*.

The general precedence constraint $c_1\ [b] \prec c_2$ that can specify the differences b between the number of occurrences of two clocks before the precedence takes effect. Hence, it is able to express more complicated relations. For instance, if the buffer initially is nonempty, fetch operations can be performed prior to any

store operation. Figure 1 shows such a scenario where 4 elements are initially presented in the buffer. This behavior can be represented as: $store\ [4] \prec fetch$.

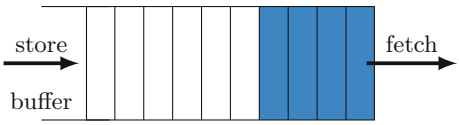


Fig. 1. Example for $store\ [4] \prec fetch$

The causality, subclock and exclusion constraints are straightforward. The causality constraint $c_1 \preccurlyeq c_2$ specifies that the occurrence of c_2 must be caused by the occurrence of c_1, namely at any moment c_1 must have ticked at least as many times as c_2 has. The subclock constraint $c_1 \subseteq c_2$ expresses that c_1 occurs at some step only if c_2 occur at this step as well. The exclusion constraint $c_1 \# c_2$ specifies that two clocks c_1 and c_2 are exclusive, i.e., they cannot occur simultaneously at the same step.

The union and intersection constraints are used to define clocks. $c_1 \triangleq c_2 + c_3$ defines a clock c_1 such that c_1 ticks iff c_2 or c_3 ticks. Similarly, $c_1 \triangleq c_2 * c_3$ defines a clock c_1 such that c_1 ticks iff both c_2 and c_3 tick. The infimum (resp. supremum) constraint $c_1 \triangleq c_2 \wedge c_3$ (resp. $c_1 \triangleq c_2 \vee c_3$) is used to define a clock c_1 that is the slowest (resp. fastest) clock that is faster (resp. slower) than both c_2 and c_3. These two constraints are useful for expressing delay requirements between two events. Remark that clocks c_1 defined by constraints may correspond to system events, otherwise are auxiliary clocks. In the former case, these constraints can be seen as constraints specifying relations between clocks c_1, c_2 and c_3.

The periodicity constraint $c_1 \triangleq c_2 \propto p$ defines a clock c_1 such that c_1 has to be performed once every p occurrences of clock c_2. It is worth mentioning that the periodicity constraint defined in such a way is relative because of the logical nature of CCSL clocks. That is, clock c_1 is relatively periodic with respect to clock c_2. CCSL does not assume the existence of a global reference clock, most relations are defined relative to other clocks. These notions extend the equivalent behaviors which are usually defined relative to physical time. If c_2 represents a sensor that measures physical time, then c_1 becomes physically periodic.

The filtering constraint $c_1 \triangleq c_2 \blacktriangledown w$ is used to define a clock c_1 which can be seen as snapshots of the clock c_2 at some steps according to the (ω-)regular expression w. For instance, $c_1 \triangleq c_2 \blacktriangledown (01)^\omega$ expresses that c_1 simulates c_2 at every even step. It defines a logically periodic behavior of c_1 with respect to c_2.

The delayFor constraint $c_1 \triangleq c_2\ \$\ d$ (i.e., $c_1 \triangleq c_2\ \$\ d\ on\ c_2$) defines a new clock c_1 that is delayed by the clock c_2 with d steps. The general form $c_1 \triangleq c_2\ \$\ d\ on\ c_3$ defines a new clock c_1 that is delayed by c_2 with d times of the ticks of c_3. c_1 can be seen as a *sampled* clock of c_2 on the basis of c_3. For instance, $c_1 \triangleq c_2\ \$\ 1\ on\ c_3$, denotes that whenever c_2 ticks at least once between two successive ticks of c_3 at steps m and n, c_1 must tick at step n.

3 Scheduling Problem of CCSL

3.1 Schedulability

Given a set Φ of CCSL constraints, a schedule δ satisfies Φ, denoted by $\delta \models \Phi$, iff $\delta \models \phi$ for all constraints $\phi \in \Phi$.

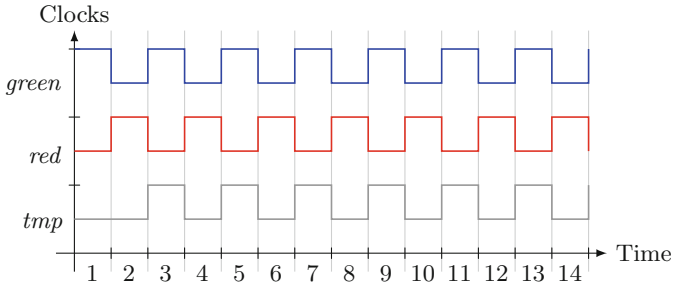


Fig. 2. The unique schedule that satisfies the three constraints in the example

Definition 5 (Logical time scheduling problem). *Given a set Φ of* CCSL *constraints,* the (logical time) scheduling problem *of* CCSL *is to determine whether there exists a schedule δ such that $\delta \models \Phi$.*

We illustrate the scheduling problem by a simple example. Consider alternative flickering between the green and red light using CCSL. We assume that green light starts first. The timing requirements can be formalized by the following three constraints:

$$green \prec red, \qquad tmp \triangleq green \$ 1, \qquad red \prec tmp,$$

where *green* and *red* are clocks respectively representing whether the green (resp. red) light is turned on, the clock *tmp* is an auxiliary clock used to help specify the constraints on clocks.

There exists exactly one schedule satisfying the three constraints, as shown in Fig. 2. In this schedule, the clock *tmp* has the same behavior as *green* from step 2, while the clock *red* has the opposite behavior to *green*. Namely, *red* and *green* operates in an alternative manner. For simplicity, we also write $green \sim red$ to denote the *alternation* relation of the two clocks.

Although one may be able to find one or more schedules for some simple constraints, to our knowledge, there is no generally applicable decision procedure solving the scheduling problem of full CCSL. There are two main challenges. First, schedules are essentially *infinite*, i.e., defined on all the natural numbers. Second, the *precedence* is *stateful*, i.e., it depends on the history, and there is no upper bound on how far in the history one must go back. It may then require an infinite memory to store the history. As a first step to tackle this challenging problem, in this work, we first consider the *bounded* scheduling problem.

3.2 Bounded Scheduling Problem

Given a bound $k \in \mathbb{N}^+$, let $\sigma : \mathbb{N}^+_{\leq k} \to 2^C$ be a function. σ is an *k-bounded schedule* of a set Φ of CCSL constraints, denoted by $\sigma \models_k \Phi$, iff there exists a schedule δ such that $\delta(i) = \sigma(i)$ for every $i \in \mathbb{N}^+_{\leq k}$ and $\delta \models \Phi$ from step 1 up to k, where $\mathbb{N}^+_{\leq k} := \{1, \cdots, k\}$.

Definition 6 (Bounded scheduling problem). *The* bounded scheduling problem *is to determine, for a given set Φ of* CCSL *constraints and a bound k, whether there is an k-bounded schedule σ for Φ, i.e., $\sigma \models_k \Phi$.*

Theorem 1 (Sufficient condition of unschedulability). *If a set Φ of constraints has no k-bounded schedule for some $k \in \mathbb{N}^+$, then Φ is unschedulable.*

The proof is straightforward by contradiction.

It is easy to see that the bounded scheduling problem is decidable, as there are finitely many potential k-bounded schedules, i.e., $(2^{|C|}-1)^k$, where $|C|$ denotes the number of clocks. Furthermore, the satisfiability problem of Boolean formulas can be reduced to the bounded scheduling problem in polynomial time.

Theorem 2. *The k-bounded scheduling problem of* CCSL *is* NP*-complete, even if $k = 1$.*

Proof. The NP upper bound can be proved easily based on the facts that the number of possible k-bounded schedules is finite and the universal quantification $\forall n \in \mathbb{N}^+_{\leq k}$ can be eliminated by enumerating all the possible values in $\mathbb{N}^+_{\leq k}$.

We prove the NP-hardness by a reduction from the satisfiability problem of Boolean formulas which is known NP-complete. Consider the Boolean formula $\phi = \bigwedge_{i=1}^{m}(l_i^1 \vee l_i^2 \vee l_i^3)$, where $m \in \mathbb{N}^+$ and l_i^j for $j \in \{1,2,3\}$ is either a Boolean variable x or its negation $\neg x$. Let $\mathsf{Var}(\phi)$ denote the set of Boolean variables appearing in ϕ. We construct a set of CCSL constraints Φ as follows.

For each $x \in \mathsf{Var}(\phi)$, we have two clocks x^+ and x^-. Let $\mathsf{enc}(x) = x^+$ and $\mathsf{enc}(\neg x) = x^-$. Each clause $l_i^1 \vee l_i^2 \vee l_i^3$ in ϕ is encoded as the CCSL constraint $c_i \triangleq \mathsf{enc}(l_i^1)+\mathsf{enc}(l_i^2)+\mathsf{enc}(l_i^3)$, denoted by ψ_i. Note that $c_i \triangleq \mathsf{enc}(l_i^1)+\mathsf{enc}(l_i^2)+\mathsf{enc}(l_i^3)$ can be transformed into CCSL constraints by introducing one auxiliary clock c, i.e., $\{c_i \triangleq \mathsf{enc}(l_i^1)+\mathsf{enc}(l_i^2)+\mathsf{enc}(l_i^3)\} \equiv \{c_i \triangleq \mathsf{enc}(l_i^1)+c, c \triangleq \mathsf{enc}(l_i^2)+\mathsf{enc}(l_i^3)\}$.

Let $\mathsf{enc}(\phi)$ denote the following set of CCSL constraints

$$\{\mathbf{1} \triangleq *_{i=1}^{m} c_i, \psi_1, ..., \psi_m, x^+ \,\#\, x^-, \mathbf{1} \triangleq x^+ + x^- \mid x \in \mathsf{Var}(\phi)\}$$

where $x^+ \,\#\, x^-$ and $\mathbf{1} \triangleq x^+ + x^-$ enforce that either x^+ or x^- ticks at each step, but not both. This encodes that either x is true or $\neg x$ is true. Note that $\tau \triangleq *_{i=1}^{m} c_i$ is a shorthand of $\tau \triangleq c_1 * \cdots * c_m$, and can also be expressed in CCSL constraints by introducing polynomial number of auxiliary clocks. For instance, $\{c \triangleq c_1 * c_2 * c_3\} \equiv \{c \triangleq c_1 * c', c' \triangleq c_2 * c_3\}$. We can show that ϕ is satisfiable iff $\mathsf{enc}(\phi)$ is 1-bounded schedulable. The satisfiability problem of Boolean formulas is NP-complete, we get that the 1-bounded scheduling problem of CCSL is NP-hard. The k-bounded scheduling problem for $k > 1$ immediately follows by repeating the ticks of clocks at the first step. □

Theorem 2 indicates the time complexity of the bounded scheduling problem. Thus, we need to find practical solutions that are algorithmically efficient for it. In the next section, we propose an SMT-based decision procedure for the bounded scheduling problem and a sound algorithm for the scheduling problem. Thanks to advances in state-of-the-art SMT solvers such as Z3 [25], our approach is usually efficient in practice.

4 Decision Procedure for the Scheduling Problem

4.1 Transformation from CCSL into SMT

Let us fix a set of CCSL constraints Φ defined over a set C of clocks. Each clock $c \in C$ is interpreted as a predicate $t_c : \mathbb{N}^+ \rightarrow \mathsf{Bool}$ such that for all $i \in \mathbb{N}^+$, $t_c(i)$ is true iff the clock c ticks at i, where Bool denotes Boolean sort. A schedule δ of Φ is encoded as a set of predicates $\mathcal{T}_C = \{t_c | c \in C\}$ such that the following condition holds: for all $t_c \in \mathcal{T}_C$,

$$\forall i \in \mathbb{N}^+ . t_c(i) \Leftrightarrow c \in \delta(i).$$

Recalling that schedules forbid stuttering steps, this condition is enforced by restricting the predicates t_c in $\mathcal{T}_C$ to satisfy the following condition:

$$\forall i \in \mathbb{N}^+ . \vee_{c \in C} t_c(i) \tag{F1}$$

Formula F1 specifies that at each step i at least one clock c ticks, i.e., $t_c(i)$ holds.

For each clock $c \in C$, we introduce an auxiliary function $h_c : \mathbb{N}^+ \rightarrow \mathbb{N}$ to encode its history. For each $i \in \mathbb{N}^+$,

$$h_c(i) := \begin{cases} 0, & \text{if } i = 1; \\ h_c(i-1), & \text{if } i > 1 \wedge \neg t_c(i-1); \\ h_c(i-1) + 1, & \text{if } i > 1 \wedge t_c(i-1). \end{cases} \tag{F2}$$

Intuitively, $h_c(i)$ is equivalent to $\chi(c, i)$ for each $i \in \mathbb{N}^+$. The set of all the auxiliary functions is denoted by $\mathcal{H}_C$.

By replacing each occurrence of clock c in $\delta(n)$ (resp. $c \notin \delta(n)$) with $t_c(n)$ (resp. $\neg t_c(n)$) and $\chi(c, n)$ with $h_c(n)$ in the definition of each CCSL constraint, each CCSL constraint ϕ can be encoded as an SMT formula $[\![\phi]\!]$.

We use $[\![\Phi]\!]$ to denote the conjunction of Formulas F1, F2 and the SMT encodings of CCSL constraints in Φ. Formally,

$$[\![\Phi]\!] := \text{F1} \wedge \text{F2} \wedge (\wedge_{\phi \in \Phi} [\![\phi]\!]).$$

Finding a schedule for Φ amounts to finding a solution, i.e., definitions of predicates in $\mathcal{T}_C$, which satisfies $[\![\Phi]\!]$.

Proposition 1. *Φ has a schedule iff $[\![\Phi]\!]$ is satisfiable.*

The scheduling problem of Φ is transformed into the satisfiability problem of the formula $[\![\Phi]\!]$. However, according to the SMT-LIB standard [4], $[\![\Phi]\!]$ belongs to the logic of UFLIA (formulas with Uninterpreted Functions and Linear Integer Arithmetic), whose satisfiability problem is undecidable in general. Nevertheless, the SMT encoding is still useful to solve the bounded scheduling problem, which we will present in the next subsection.

4.2 Decision Procedure for the Bounded Scheduling Problem

For k-bounded scheduling problem, it suffices to consider schedules $\delta : \mathbb{N}^+_{\leq k} \to 2^C$. Moreover, the quantifiers in $[\![\Phi]\!]$ can be eliminated once the bound k is fixed. Hence, we can resort to state-of-the-art SMT solvers. Formally, let $[\![\Phi]\!]_k$ be the formula obtained from $[\![\Phi]\!] = F1 \wedge F2 \wedge (\bigwedge_{\phi \in \Phi} [\![\phi]\!])$ by

- restricting the domain of predicates $t_c \in \mathcal{T}_C$ and functions $h_c \in \mathcal{H}_C$ to $\mathbb{N}^+_{\leq k}$;
- replacing quantifications $\forall n \in \mathbb{N}^+$ and $\exists m \in \mathbb{N}^+$ with $\forall n \in \mathbb{N}^+_{\leq k}$ and $\exists m \in \mathbb{N}^+_{\leq k}$ in $(\bigwedge_{\phi \in \Phi} [\![\phi]\!])$.

Proposition 2. *Φ is k-bounded schedulable iff $[\![\Phi]\!]_k$ is satisfiable.*
Moreover, if $[\![\Phi]\!]_k$ is satisfiable, then $[\![\Phi]\!]_{k'}$ is satisfiable for all $k' \leq k$.

4.3 A Sound Algorithm for the Scheduling Problem

According to Theorem 1, Propositions 1 and 2, (1) if $[\![\Phi]\!]$ is satisfiable, then Φ is schedulable, and (2) if $[\![\Phi]\!]_k$ for some $k \in \mathbb{N}^+$ is unsatisfiable, then Φ is unschedulable. We can deduce a sound algorithm for checking the general scheduling problem. However, randomly choosing a bound k and checking whether or not $[\![\Phi]\!]_k$ is unsatisfiable may be inefficient, as the k-bounded scheduling problem is NP-hard (cf. Theorem 2), and larger bound k may result in time out, but smaller bound k may result in that $[\![\Phi]\!]_k$ is satisfiable. Indeed, if we consider the maximal bound B, then the random approach may have to call SMT solving $\mathbf{O}(B)$ times. Alternatively, we propose a binary-search based approach as shown in Algorithm 1 for a given maximal bound B, which invokes SMT solving at most $\mathbf{O}(|\log_2 B|)$ times.

Algorithm 1: A sound algorithm for the scheduling problem

Input : a set of constraints Φ, a timeout threshold T, a maximal bound B
Output: $\{\texttt{SAT}, \texttt{UNSAT}, \texttt{Timeout}\} \times \mathbb{N}^+$

```
1  result_1 ← SMTSolver([[Φ]], T);
2  if result_1 = SAT then                                   /* Schedulable */
3      return (SAT, 0)
4  l ← 0; u ← B;
5  while l ≤ u do                                           /* Binary search */
6      k ← ⌊(l+u)/2⌋;
7      result_2 ← SMTSolver([[Φ]]_k, T);
8      if result_2 = SAT then l ← k + 1;                    /* Upper half */
9      else                                                 /* Lower half */
10         u ← k − 1;
11         if result_1 = UNSAT ∨ result_2 = UNSAT then
12             result_1 ← UNSAT;
13 if result_2 ≠ SAT then k ← k − 1;
14 return (result_1, k);
```

Given a set Φ of constraints in CCSL, a timeout threshold T and a maximal bound B, Algorithm 1 first invokes an `SMTSolver` to decide whether $[\![\Phi]\!]$ is satisfiable or not within `T` time. If $[\![\Phi]\!]$ is satisfiable, then Algorithm 1 returns (`SAT,0`), meaning that Φ is schedulable. Otherwise, it binary searches a bound $k \leq B$ such that $[\![\Phi]\!]_k$ is satisfiable while $[\![\Phi]\!]_{k+1}$ (if $k+1 \leq B$) is unsatisfiable or cannot be verified in time `T`.

Theorem 3. *Algorithm 1 has the following three properties:*

1. *If it returns* (`SAT,0`)*, then* Φ *is schedulable.*
2. *If it returns* (`UNSAT,k`)*, then* Φ *is unschedulable. If* $k \neq 0$*, then* Φ *has k-bounded schedulable, otherwise does not have any bounded schedulable.*
3. *If it returns* (`Timeout,k`)*, then* Φ *is k-bounded schedulable if* $k \neq 0$*, otherwise no bounded schedule is found for* Φ*.*

5 Case Study and Performance Evaluation

We implemented our approach in a prototype tool with Z3 [25] as its underlying SMT solver. We conduct a case study on expressing requirements of an interlocking system in CCSL constraints and analyzing its schedulability. Then, we prove 12 algebraic properties of CCSL constraints using the tool. Finally, we evaluate the performance of the tool using 9 sets of CCSL constraints.

5.1 Schedulability of an Interlocking System

The interlocking system is a subsystem of a rail transit system. It is used to prevent trains from collisions and derailments when they are moving under the control of signal lights. As shown in Fig. 3, the interlocking system monitors the occupancy status of the individual track section, and sends signals to inform drivers whether they are allowed to enter the route or not. The railway tracks are divided into sections. Each section is associated with a track circuit for detecting whether it is occupied by a train or not. Signal lights are placed between track sections. They can be red and green to indicate proceeding and stopping, respectively.

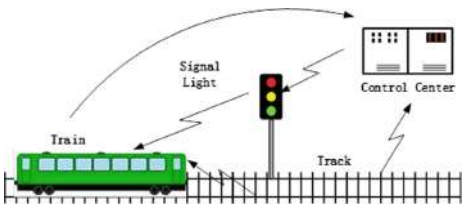


Fig. 3. Interlocking system

The mechanism and operation procedure of the interlocking system are summarized as follows.

1. To enter a track, a train first sends a request to the control center.
2. On receiving the request, the control center sends an inquiry to the track circuit to detect the status of the track.

Table 2. CCSL constraints of the interlocking system

request $\prec$ inquiry	responseOfTrack $\triangleq$ checkSucc + checkFail
checkFail $\prec$ redPulse	responseOfTrain $\triangleq$ enter + wait
redPulse $\preccurlyeq$ showRed	inquiry $\prec$ responseOfTrack
showRed $\prec$ wait	getOccupied $\sim$ getUnoccupied
checkSucc $\prec$ greenPulse	getOccupied # getUnoccupied
greenPulse $\preccurlyeq$ showGreen	request $\sim$ responseOfTrain
showGreen $\prec$ enter	inquiry $-$ responseOfTrack ≤ 40
enter $\prec$ leave	greenPulse $-$ showGreen ≤ 30
enter $\subseteq$ getOccupied	redPulse $-$ showRed ≤ 30
leave $\subseteq$ getUnoccupied	request $-$ responseOfTrain ≤ 50
getOccupied $\sim$ tmp_1	checkFail $-$ showRed ≤ 40
getUnoccupied $\sim$ tmp_1	checkSucc $-$ showGreen ≤ 40
checkFail $\subseteq$ tmp_1	getUnoccupied $\prec$ tmp_2
$tmp_2 \prec$ getOccupied	checkSucc $\subseteq$ tmp_2

3. If the track is occupied, it sends *checkFail* to the control center, and otherwise *checkSucc*.
4. On receiving the message *checkFail* (*resp. checkSucc*), the control center sends a red (*resp.* green) signal pulse to the signal light.
5. The signal light turns red (*resp.* green) on receiving the red (*resp.* green) signal pulse.
6. The train will enter after seeing the light is green, and the track becomes occupied. In case of the red light, the train must stop and wait.
7. The track becomes unoccupied after the train leaves. If the train is waiting, it must send a request again after some time.

There are time constraints on the above operations. For instance, the control center needs to get a response from the track circuit within 30 ms after sending an inquiry to it. The train must make decision within 50 ms after it sends a request to the control center. The light should turn to the corresponding color within 30 ms after it receives a pulse. After the track becomes occupied (*resp.* unoccupied), the light must turn red (*resp.* green) within 40 ms.

Table 2 shows the main logical constraints on the operations in the system and their timing constraints. We use some non-standard constraint expressions for the sake of compactness. Constraint $a - b \leq n$ denotes that b must tick within n steps after a ticks. It equals the set of the following three constraints:

$$a \prec b, \quad t \triangleq a \$ n \; on \; \mathbf{1}, \quad b \preccurlyeq t.$$

Note that in this example the unit of time is millisecond (ms). Thus, there is an implicit assumption in the constraints that every tick of a logic clock means the elapse of one millisecond.

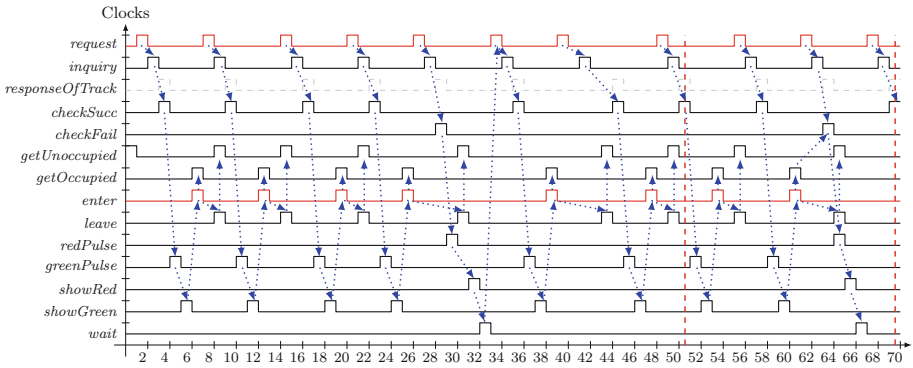


Fig. 4. A bounded schedule for the CCSL constraints in the case study

Most constraints in Table 2 are straightforward, except the six constraints marked with waved underlines. The first three constraints specify that `checkFail` only can occur between the occurrences of `getUnoccupied` and `getOccupied`. The others specify the following two requirements:

1. `checkSucc` only can occur after `getUnoccupied` and before `getOccupied`;
2. `getUnoccupied` precedes `getOccupied`.

Given these constraints, our tool found a bounded schedule as depicted in Fig. 4. From step 1 to step 7, one complete process is finished. Initially, the track gets unoccupied. At step 2, a request is made, which causes subsequent operations to occur from step 3 to step 7. At step 29, a fail case occurs because another train enters (step 26) but has not left (step 31). The train that made the request has to wait (step 33).

If we extend the bounded schedule by infinitely repeating the behaviors of all the clocks between step 51 and 69 from step 70, we obtain an infinite schedule. The extended schedule satisfies all the constraints, and thus it is a witness of the schedulability of designed mechanism for the interlocking system.

In this paper, we are only concerned with the schedulability of the constraints in the example. Some other kinds of temporal properties also need to verify. For instance, we must guarantee that whenever a train requests to enter the station, it must eventually enter. We also need to verify the system is deadlock-free. Such temporal properties can be verified by LTL model checking of CCSL constraints using SMT technique [40]. We omit it because it is beyond the scope of this paper.

5.2 Automatic Proof of CCSL Algebraic Properties

Using the proposed approach, we can also prove automatically algebraic properties of CCSL constraints such as the commutativity of exclusion and transitivity of causality. Algebraic properties of CCSL constraints can be represented as $\Phi \Rightarrow \phi$, where Φ is a set of CCSL constraints and ϕ is a constraint derived from Φ. Proving $\Phi \Rightarrow \phi$ is valid equals proving the unsatisfiability of $[\![\Phi]\!] \wedge \neg[\![\phi]\!]$, which can be solved by Algorithm 1.

Table 3. Proved algebraic properties of CCSL constraints

Algebraic property	Definition
Commutativity of exclusion	$c1 \# c2 \Rightarrow c2 \# c1$
Transitivity of causality	$c_1 \preccurlyeq c_2$, $c_2 \preccurlyeq c_3 \Rightarrow c_1 \preccurlyeq c_3$
Antisymmetry of causality	$c_1 \preccurlyeq c_2$, $c_2 \preccurlyeq c_1 \Rightarrow c_1 = c_2$
Fastness of infimum	$c_1 \triangleq c_2 \wedge c_3 \Rightarrow c_1 \preccurlyeq c_2, c_1 \preccurlyeq c_3$
Slowestness of infimum	$c_1 \triangleq c_2 \wedge c_3, c_4 \preccurlyeq c_2, c_4 \preccurlyeq c_3 \Rightarrow c_4 \preccurlyeq c_1$
Slowness of supremum	$c_1 \triangleq c_2 \vee c_3 \Rightarrow c_2 \preccurlyeq c_1, c_3 \preccurlyeq c_1$
Fastestness of supremum	$c_1 \triangleq c_2 \vee c_3, c_2 \preccurlyeq c_4, c_3 \preccurlyeq c_4 \Rightarrow c_1 \preccurlyeq c_4$
Causality of subclock	$c_1 \subseteq c_2 \Rightarrow c_2 \preccurlyeq c_1$
Causality of union	$c_1 \triangleq c_2 + c_3 \Rightarrow c_1 \preccurlyeq c_2, c_1 \preccurlyeq c_3$
Causality of intersection	$c_1 \triangleq c_2 * c_3 \Rightarrow c_2 \preccurlyeq c_1, c_3 \preccurlyeq c_1$
Subclocking of sampling	$c_1 \triangleq c_2 \lightning c_3 \Rightarrow c_1 \subseteq c_3$
Subclocking of union	$c_1 \triangleq c_2 + c_3 \Rightarrow c_2 \subseteq c_1, c_3 \subseteq c_1$
Subclocking of intersection	$c_1 \triangleq c_2 * c_3 \Rightarrow c_1 \subseteq c_2, c_1 \subseteq c_3$

Let us consider the proof of the slowestness of infimum as an example. The slowestness of infimum means that an infimum constraint $c_1 \triangleq c_2 \wedge c_3$ defines the slowest clock c_1 among those that are faster than both c_2 and c_3.

Proposition 3 (Slowestness of infimum). *Given two clocks c_2, c_3, let $c_1 \triangleq c_2 \wedge c_3$ and c_4 be an arbitrary clock such that $c_4 \preccurlyeq c_2$ and $c_4 \preccurlyeq c_3$, then $c_4 \preccurlyeq c_1$.*

This is proved by transforming CCSL constraints into the following SMT formula according the SMT encoding method:

$$[\![c_1 \triangleq c_2 \wedge c_3]\!] \wedge [\![c_4 \preccurlyeq c_2]\!] \wedge [\![c_4 \preccurlyeq c_3]\!] \wedge \neg[\![c_4 \preccurlyeq c_1]\!].$$

Algorithm 1 returns (`UNSAT`, 0), which means that the formula is proved unsatisfiable. The proposition is proved.

Table 3 lists the algebraic properties that have been successfully proved in our approach. Algebraic properties are useful to help understand the relation among CCSL constraints. Using them we can also verify whether some CCSL constraints are redundant or inconsistent for a given set of CCSL constraints.

5.3 Performance Evaluation

To evaluate the performance our tool, we collected 9 sets of CCSL constraints from the literature and real-world applications, and analyzed their schedulability using our tool. Under different time thresholds, we calculate the maximal bounds under which the constraints are schedulable.

Table 4 shows all the experimental results including the corresponding execution time. All the experiments were conducted on a Win 10 running on an i7 CPU with 2.70 GHz and 16 GB memory. The numbers followed by asterisks

Table 4. Experimental results of bounded schedulability analysis

CS	Clks.	Cons.	THD: 10 s		THD: 20 s		THD: 30 s		THD: 40 s	
			BD	TM	BD	TM	BD	TM	BD	TM
CS1	3	3	8	0.06	8	0.06	8	0.06	8	0.06
CS2	3	4	2^*	0.06	2^*	0.06	2^*	0.06	2^*	0.06
CS3	8	9	48	6.20	59	15.88	70	28.72	75	39.82
CS4	8	7	70	7.12	70	7.12	70	7.12	70	7.12
CS5	9	9	80	8.29	90	19.95	110	26.81	111	39.84
CS6	10	6	95	9.40	113	14.26	113	14.26	113	14.26
CS7	12	9	69	8.80	76	19.42	89	27.69	95	40.00
CS8	17	20	16	0.81	16	0.81	16^*	27.36	16^*	27.36
CS9	27	51	30	9.94	41	17.19	45	29.78	45	29.78

Remarks: CS: constraint set, Cons: the number of constraints, Clks: the number of clocks, THD: timeout threshold, TM: Time (second), BD: upper bound.

are the maximal bounds such that the corresponding constraints are bounded schedulable, but unschedulable in the next step. It is interesting to observe from Table 4 that time cost is loosely related to size (the number of clocks and constraints), thanks to efficient search strategies of SMT solvers. This is in striking contrasts to automata-based [29,35] and the rewriting-based approaches [38], whose scalability suffers from both the numbers of clocks and constraints.

6 Related Work

CCSL is directly derived from the family of synchronous languages, such as Lustre [9], Esterel [6] and Signal [5], and its the scheduling problem of CCSL is akin to what synchronous languages call clock calculus. The main differences are: CCSL is a specification language, while others are programming languages; and CCSL partially describes what is expected to happen in a declarative way and does not give a direct operational deterministic description of what must happen. Furthermore, CCSL only deals with pure clocks while the others deal with signals and extract the clocks when needed.

The Esterel compiler [31] applies a constructive approach to decide when a signal must occur (compute its clock) and what its value should be. This requires a detection of *causality cycles*, or intra-cycle data dependencies, which are also naturally addressed by our approach. However, the Esterel compiler compiles an imperative program into a Boolean circuit, or equivalently a finite state machine. Consequently, it cannot deal with CCSL unbounded schedules.

The clock calculus in Signal attempts to detect whether the specification is endochronous [30], in which case it can generate some efficient code. This analysis is mainly based on the subclock relationship that also exists in CCSL. In CCSL, we consider the problem whether there is at least one possible schedule or not.

In Lustre and its extensions, clocks are regarded as abstract types [13] and the clock calculus computes the relative rates of clocks while rejecting the program when computing the rates is not possible. In most cases, the compiler attempts to build bounded buffers and to ensure that the functional determinism can be preserved with a finite memory. In our case, we do not seek to reach a finite representation, as in the first specification steps this is not a primary goal for the designers. Indeed, this might lead to an over-specification of the problem.

Classical real-time scheduling problem [32] usually relies on task models, arrival patterns and constraints (e.g., precedence, resources) to propose algorithms for the scheduling problem with analytical results [19] or heuristics depending on the specific model (e.g., priorities, preemptive). Other solutions, based on timed automata [1,2,17] or timed Petri nets [8,18], propose a general framework for describing all the relevant aspects without assuming a specific task model. CCSL offers an alternative method based on logical time. It is believed that logical time and multiform time bases offer some flexibility to unify functional requirements and performance constraints. We rely on CCSL and we claim that after encoding a task model in CCSL, finding a schedule for the CCSL model also gives a schedule for the encoded task model [24].

There have been many efforts made towards the scheduling problem of CCSL, though no conclusion is drawn on its decidability. TimeSquare [14] is a simulation tool for CCSL which can produce a possible schedule for a given set of CCSL, up to a given user-defined bound. It also supports different simulation strategies for producing desired execution traces. Some earlier work [20] define the notion of *safe* CCSL specifications that can be encoded with a finite-state machine. The scheduling problem is decidable for safe specifications, as one can merely enumerate all the (finite) solutions. A semi-algorithm can build the finite representation when the specification is safe [21]. In [37], Zhang et al. proposed a state-based approach and a sufficient condition to decide whether safe and unsafe specifications accept a so-called *periodic schedule* [39]. This allows to build a finite solution for unsafe specifications, while there may also exist infinite solutions. Xu et al. proposed a notion of *divergence* of CCSL to study the schedulability of CCSL, and proved that a set of CCSL constraints is schedulable if all the constraints are divergent [34]. They resorted to the theorem prover PVS [27] to assist the divergence proof.

The scheduling problem of CCSL constraints in this work resorts to SMT solving to deal with the bounded and unbounded schedules. Using SMT solving has two advantages: (1) it is usually efficient in practice, and (2) it can deal with unsafe CCSL constraints such as infimum and supremum [21].

Some basic algebraic properties on CCSL relations have been established manually before [23] but we provide here an automatic framework to do so.

7 Conclusion and Future Work

In this work, we proved that the bounded scheduling problem of CCSL is NP-complete, and proposed an SMT-based decision procedure for the bounded

scheduling problem. The procedure is sound and complete. The experimental results also show its efficiency in practice. Based on this decision procedure, we devised a sound algorithm for the general scheduling problem. We evaluated the effectiveness of the proposed approach on an interlocking system. We also showed our approach can be used to prove algebraic properties of CCSL constraints.

Our approach to the bounded scheduling problem of CCSL makes us one step closer to tackling the general (i.e. unbounded) scheduling problem. As the case study demonstrates, one may find an infinite schedule by extending a bounded one such that the extended infinite schedule still satisfies the constraints. This observation inspires future work to investigate mechanisms of finding such bounded schedules, hopefully with SMT solvers by extending our algorithm. In our earlier work [37], we proposed a similar approach to search for periodical schedules in bounded steps. In that approach, CCSL constraints are transformed into finite state machine and consequently suffers from the state explosion problem. We believe our SMT-based approach can be extended to their work while still avoiding state explosion. We leave it to future work.

References

1. Abdeddaïm, Y., Asarin, E., Maler, O.: Scheduling with timed automata. Theor. Comput. Sci. **354**(2), 272–300 (2006)
2. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 60–72. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-40903-8_6
3. André, C., Mallet, F., de Simone, R.: Modeling time(s). In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 559–573. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_38
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard (2016)
5. Benveniste, A., Guernic, P.L., Jacquemot, C.: Synchronous programming with events and relations: the SIGNAL language and its semantics. Sci. Comput. Program. **16**(2), 103–149 (1991)
6. Berry, G., Gonthier, G.: The esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
7. Boehm, B., Basili, V.R.: Software defect reduction top 10 list. Computer **34**(1), 135–137 (2001)
8. Bucci, G., Fedeli, A., Sassoli, L., Vicario, E.: Modeling flexible real time systems with preemptive time petri nets. In: Proceedings of the 15th ECRTS, Porto, Portugal, pp. 279–286. IEEE (2003)
9. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: LUSTRE: a declarative language for programming synchronous systems. In: Proceedings of 14th POPL, Tucson, USA, pp. 178–188. ACM Press (1987)
10. Chen, X., Yin, L., Yu, Y., Jin, Z.: Transforming timing requirements into CCSL constraints to verify cyber-physical systems. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 54–70. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_4
11. Chen, Y., Chen, Y., Madelaine, E.: Timed-pNets: a communication behavioural semantic model for distributed systems. Front. Comput. Sci. **9**(1), 87–110 (2015)

12. Colaço, J., Pagano, B., Pouzet, M.: SCADE 6: a formal language for embedded critical software development. In: Proceedings of the 11th TASE, Sophia Antipolis, France, pp. 1–11. IEEE (2017)
13. Colaço, J.-L., Pouzet, M.: Clocks as first class abstract types. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 134–155. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45212-6_10
14. Deantoni, J., Mallet, F.: TimeSquare: treat your models with logical time. In: Proceedings of the 50th TOOLS, Prague, Czech Republic, pp. 34–41. IEEE (2012)
15. Feiler, P.H., Gluch, D.P.: Model-based engineering with AADL - an introduction to the SAE architecture analysis and design language. SEI, Addison-Wesley (2012)
16. Kang, E., Schobbens, P.: Schedulability analysis support for automotive systems: from requirement to implementation. In: Proceedings of the 29th SAC, Gyeongju, Korea, pp. 1080–1085. ACM (2014)
17. Krčál, P., Yi, W.: Decidable and undecidable problems in schedulability analysis using timed automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 236–250. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_20
18. Lime, D., Roux, O.: A translation based method for the timed analysis of scheduling extended time petri nets. In: Proceedings of the 25th RTSS, pp. 187–196. IEEE (2004)
19. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
20. Mallet, F., Millo, J.-V.: Boundness issues in CCSL specifications. In: Groves, L., Sun, J. (eds.) ICFEM 2013. LNCS, vol. 8144, pp. 20–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41202-8_3
21. Mallet, F., de Simone, R.: Correctness issues on MARTE/CCSL constraints. Sci. Comput. Program. **106**, 78–92 (2015)
22. Mallet, F., Villar, E., Herrera, F.: MARTE for CPS and CPSoS. In: Nakajima, S., Talpin, J.-P., Toyoshima, M., Yu, H. (eds.) Cyber-Physical System Design from an Architecture Analysis Viewpoint, pp. 81–108. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-4436-6_4
23. Mallet, F., Millo, J., de Simone, R.: Safe CCSL specifications and marked graphs. In: Proceedings of the 11th MEMOCODE, Portland, OR, USA, pp. 157–166. IEEE (2013)
24. Mallet, F., Zhang, M.: Work-in-progress: from logical time scheduling to real-time scheduling. In: Proceedings of the 39th RTSS, Nashville, USA, pp. 143–146. IEEE (2018)
25. 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
26. OMG: UML profile for MARTE: modeling and analysis of real-time embedded systems (2015)
27. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
28. Peters, J., Przigoda, N., Wille, R., Drechsler, R.: Clocks vs. instants relations: verifying CCSL time constraints in UML/MARTE models. In: Proceedings of the 14th MEMOCODE, Kanpur, India, pp. 78–84. IEEE (2016)
29. Peters, J., Wille, R., Przigoda, N., Kühne, U., Drechsler, R.: A generic representation of CCSL time constraints for UML/MARTE models. In: Proceedings of the 52nd DAC, pp. 122:1–122:6. ACM (2015)

30. Potop-Butucaru, D., Caillaud, B., Benveniste, A.: Concurrency in synchronous systems. Formal Methods Syst. Des. **28**(2), 111–130 (2006)
31. Potop-Butucaru, D., Edwards, S.A., Berry, G.: Compiling Esterel. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-70628-3
32. Sha, L., et al.: Real time scheduling theory: a historical perspective. Real-Time Syst. **28**(2–3), 101–155 (2004)
33. Suryadevara, J., Seceleanu, C., Mallet, F., Pettersson, P.: Verifying MARTE/CCSL mode behaviors using UPPAAL. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 1–15. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40561-7_1
34. Xu, Q., de Simone, R., DeAntoni, J.: Divergence detection for CCSL specification via clock causality chain. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 18–37. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47677-3_2
35. Yin, L., Mallet, F., Liu, J.: Verification of MARTE/CCSL time requirements in Promela/SPIN. In: Proceedings of the 16th ICECCS, USA, pp. 65–74. IEEE (2011)
36. Yu, H., Talpin, J., Besnard, L., et al.: Polychronous controller synthesis from MARTE/CCSL timing specifications. In: Proceedings of the 9th MEMOCODE, Cambridge, UK, pp. 21–30. IEEE (2011)
37. Zhang, M., Dai, F., Mallet, F.: Periodic scheduling for MARTE/CCSL: theory and practice. Sci. Comput. Program. **154**, 42–60 (2018)
38. Zhang, M., Mallet, F.: An executable semantics of clock constraint specification language and its applications. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2015. CCIS, vol. 596, pp. 37–51. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29510-7_2
39. Zhang, M., Mallet, F., Zhu, H.: An SMT-based approach to the formal analysis of MARTE/CCSL. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 433–449. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_27
40. Zhang, M., Ying, Y.: Towards SMT-based LTL model checking of clock constraint specification language for real-time and embedded systems. In: Proceedings of the 18th LCTES, Barcelona, Spain, pp. 61–70. ACM (2017)

A Hybrid Dynamic Logic for Event/Data-Based Systems

Rolf Hennicker[1], Alexandre Madeira[2,3(✉)], and Alexander Knapp[4]

[1] Ludwig-Maximilians-Universität München, Munich, Germany
hennicke@pst.ifi.lmu.de
[2] CIDMA, University of Aveiro, Aveiro, Portugal
madeira@ua.pt
[3] QuantaLab, University of Minho, Braga, Portugal
[4] Universität Augsburg, Augsburg, Germany
knapp@informatik.uni-augsburg.de

Abstract. We propose $\mathcal{E}^{\downarrow}$-logic as a formal foundation for the specification and development of event-based systems with local data states. The logic is intended to cover a broad range of abstraction levels from abstract requirements specifications up to constructive specifications. Our logic uses diamond and box modalities over structured actions adopted from dynamic logic. Atomic actions are pairs $e/\!\!/\psi$ where e is an event and ψ a state transition predicate capturing the allowed reactions to the event. To write concrete specifications of recursive process structures we integrate (control) state variables and binders of hybrid logic. The semantic interpretation relies on event/data transition systems; specification refinement is defined by model class inclusion. For the presentation of constructive specifications we propose operational event/data specifications allowing for familiar, diagrammatic representations by state transition graphs. We show that $\mathcal{E}^{\downarrow}$-logic is powerful enough to characterise the semantics of an operational specification by a single $\mathcal{E}^{\downarrow}$-sentence. Thus the whole development process can rely on $\mathcal{E}^{\downarrow}$-logic and its semantics as a common basis. This includes also a variety of implementation constructors to support, among others, event refinement and parallel composition.

1 Introduction

Event-based systems are an important kind of software systems which are open to the environment to react to certain events. A crucial characteristics of such systems is that not any event can (or should) be expected at any time. Hence the control flow of the system is significant and should be modelled by appropriate means. On the other hand components administrate data which may change upon the occurrence of an event. Thus also the specification of admissible data changes caused by events plays a major role.

A. Madeira—Supported by ERDF through COMPETE 2020 and by National Funds through FCT with POCI-01-0145-FEDER-016692 and UID/MAT/04106/2019, in a contract foreseen in nos. 4–6 of art. 23 of the DL 57/2016, changed by DL 57/2017.

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 79–97, 2019.
https://doi.org/10.1007/978-3-030-16722-6_5

There is quite a lot of literature on modelling and specification of event-based systems. Many approaches, often underpinned by graphical notations, provide formalisms aiming at being constructive enough to suggest particular designs or implementations, like e.g., Event-B [1,7], symbolic transition systems [17], and UML behavioural and protocol state machines [12,16]. On the other hand, there are logical formalisms to express desired properties of event-based systems. Among them are temporal logics integrating state and event-based styles [4], and various kinds of modal logics involving data, like first-order dynamic logic [10] or the modal μ-calculus with data and time [9]. The gap between logics and constructive specification is usually filled by checking whether *the* model of a constructive specification satisfies certain logical formulae.

In this paper we are interested in investigating a logic which is capable to express properties of event/data-based systems on various abstraction levels in a common formalism. For this purpose we follow ideas of [15], but there data states, effects of events on them and constructive operational specifications (see below) were not considered. The advantage of an expressive logic is that we can split the transition from system requirements to system implementation into a series of gradual refinement steps which are more easy to understand, to verify, and to adjust when certain aspects of the system are to be changed or when a product line of similar products has to be developed.

To that end we propose $\mathcal{E}^{\downarrow}$-logic, a dynamic logic enriched with features of hybrid logic. The dynamic part uses diamond and box modalities over structured actions. Atomic actions are of the form $e/\!\!/\psi$ with e an event and ψ a state transition predicate specifying the admissible effects of e on the data. Using sequential composition, union, and iteration we obtain complex actions that, in connection with the modalities, can be used to specify required and forbidden behaviour. In particular, if E is a finite set of events, though data is infinite we are able to capture all reachable states of the system and to express safety and liveness properties. But $\mathcal{E}^{\downarrow}$-logic is also powerful enough to specify concrete, recursive process structures by integrating state variables and binders from hybrid logic [6] with the subtle difference that our state variables are used to denote control states only. We show that the dynamic part of the logic is bisimulation invariant while the hybrid part, due to the ability to bind names to states, is not.

An axiomatic specification $Sp = (\Sigma, Ax)$ in $\mathcal{E}^{\downarrow}$ is given by an event/data signature $\Sigma = (E, A)$, with a set E of events and a set A of attributes to model local data states, and a set of $\mathcal{E}^{\downarrow}$-sentences Ax, called axioms, expressing requirements. For the semantic interpretation we use event/data transition systems (edts). Their states are reachable configurations $\gamma = (c, \omega)$ where c is a control state, recording the current state of execution, and ω is a local data state, i.e., a valuation of the attributes. Transitions between configurations are labelled by events. The semantics of a specification Sp is "loose" in the sense that it consists of *all* edts satisfying the axioms of the specification. Such structures are called models of Sp. Loose semantics allows us to define a simple refinement notion: Sp_1 refines to Sp_2 if the model class of Sp_2 is included in the model class of Sp_1. We may also say that Sp_2 is an implementation of Sp_1.

Our refinement process starts typically with axiomatic specifications whose axioms involve only the dynamic part of the logic. Hybrid features will successively be added in refinements when specifying more concrete behaviours, like loops. Aiming at a concrete design, the use of an axiomatic specification style may, however, become cumbersome since we have to state explicitly also all negative cases, what the system should not do. For a convenient presentation of constructive specifications we propose operational event/data specifications, which are a kind of symbolic transition systems equipped again with a model class semantics in terms of edts. We will show that $\mathcal{E}^{\downarrow}$-logic, by use of the hybrid binder, is powerful enough to characterise the semantics of an operational specification. Therefore we have not really left $\mathcal{E}^{\downarrow}$-logic when refining axiomatic by operational specifications. Moreover, since several constructive notations in the literature, including (essential parts of) Event-B, symbolic transition systems, and UML protocol state machines, can be expressed as operational specifications, $\mathcal{E}^{\downarrow}$-logic provides a logical umbrella under which event/data-based systems can be developed.

In order to consider more complex refinements we take up an idea of Sannella and Tarlecki [18,19] who have proposed the notion of constructor implementation. This is a generic notion applicable to specification formalisms based on signatures and semantic structures for signatures. As both are available in the context of $\mathcal{E}^{\downarrow}$-logic, we complement our approach by introducing a couple of constructors, among them event refinement and parallel composition. For the latter we provide a useful refinement criterion relying on a relationship between syntactic and semantic parallel composition. The logic and the use of the implementation constructors will be illustrated by a running example.

Hereafter, in Sect. 2, we introduce syntax and semantics of $\mathcal{E}^{\downarrow}$-logic. In Sect. 3, we consider axiomatic as well as operational specifications and demonstrate the expressiveness of $\mathcal{E}^{\downarrow}$-logic. Refinement of both types of specifications using several implementation constructors is considered in Sect. 4. Section 5 provides some concluding remarks. Proofs of theorems and facts can be found in [11].

2 A Hybrid Dynamic Logic for Event/Data Systems

We propose the logic $\mathcal{E}^{\downarrow}$ to specify and reason about event/data-based systems. $\mathcal{E}^{\downarrow}$-logic is an extension of the hybrid dynamic logic considered in [15] by taking into account changing data. Therefore, we first summarise our underlying notions used for the treatment of data. We then introduce the syntax and semantics of $\mathcal{E}^{\downarrow}$ with its hybrid and dynamic logic features applied to events and data.

2.1 Data States

We assume given a universe $\mathcal{D}$ of *data values*. A *data signature* is given by a set A of *attributes*. An *A-data state* ω is a function $\omega : A \to \mathcal{D}$. We denote by $\Omega(A)$ the set of all A-data states. For any data signature A, we assume given a set $\Phi(A)$ of *state predicates* to be interpreted over single A-data states, and a set

$\Psi(A)$ of *transition predicates* to be interpreted over pairs of pre- and post-A-data states. The concrete syntax of state and transition predicates is of no particular importance for the following. For an attribute $a \in A$, a state predicate may be $a > 0$; and a transition predicate e.g. $a' = a + 1$, where a refers to the value of attribute a in the pre-data state and a' to its value in the post-data state. Still, both types of predicates are assumed to contain true and to be closed under negation (written $\neg$) and disjunction (written $\vee$); as usual, we will then also use false, $\wedge$, etc. Furthermore, we assume for each $A_0 \subseteq A$ a transition predicate $\mathrm{id}_{A_0} \in \Psi(A)$ expressing that the values of attributes in A_0 are the same in pre- and post-A-data states.

We write $\omega \models_A^{\mathcal{D}} \varphi$ if $\varphi \in \Phi(A)$ is satisfied in data state ω; and $(\omega_1, \omega_2) \models_A^{\mathcal{D}} \psi$ if $\psi \in \Psi(A)$ is satisfied in the pre-data state ω_1 and post-data state ω_2. In particular, $(\omega_1, \omega_2) \models_A^{\mathcal{D}} \mathrm{id}_{A_0}$ if, and only if, $\omega_1(a_0) = \omega_2(a_0)$ for all $a_0 \in A_0$.

2.2 $\mathcal{E}^{\downarrow}$-Logic

Definition 1. *An* event/data signature *(*ed signature*, for short)* $\Sigma = (E, A)$ *consists of a finite set of* events E *and a data signature* A*. We write* $E(\Sigma)$ *for* E *and* $A(\Sigma)$ *for* A*. We also write* $\Omega(\Sigma)$ *for* $\Omega(A(\Sigma))$*,* $\Phi(\Sigma)$ *for* $\Phi(A(\Sigma))$*, and* $\Psi(\Sigma)$ *for* $\Psi(A(\Sigma))$*. The class of ed signatures is denoted by* $Sig^{\mathcal{E}^{\downarrow}}$*.*

Any ed signature Σ determines a class of semantic structures, the *event/data transition systems* which are reachable transition systems with sets of initial states and events as labels on transitions. The states are pairs $\gamma = (c, \omega)$, called *configurations*, where c is a *control state* recording the current execution state and ω is an $A(\Sigma)$-data state; we write $c(\gamma)$ for c and $\omega(\gamma)$ for ω.

Definition 2. *A* Σ-event/data transition system *(*Σ-edts*, for short)* $M = (\Gamma, R, \Gamma_0)$ *over an ed signature* Σ *consists of a set of* configurations $\Gamma \subseteq C \times \Omega(\Sigma)$ *for a set of* control states C*; a family of* transition relations $R = (R_e \subseteq \Gamma \times \Gamma)_{e \in E(\Sigma)}$*; and a non-empty set of* initial configurations $\Gamma_0 \subseteq \{c_0\} \times \Omega_0$ *for an* initial control state $c_0 \in C$ *and a set of* initial data states $\Omega_0 \subseteq \Omega(\Sigma)$ *such that* Γ *is* reachable *via* R*, i.e., for all* $\gamma \in \Gamma$ *there are* $\gamma_0 \in \Gamma_0$*,* $n \geq 0$*,* $e_1, \ldots, e_n \in E(\Sigma)$*, and* $(\gamma_i, \gamma_{i+1}) \in R_{e_{i+1}}$ *for all* $0 \leq i < n$ *with* $\gamma_n = \gamma$*. We write* $\Gamma(M)$ *for* Γ*,* $C(M)$ *for* C*,* $R(M)$ *for* R*,* $c_0(M)$ *for* c_0*,* $\Omega_0(M)$ *for* Ω_0*, and* $\Gamma_0(M)$ *for* Γ_0*. The class of* Σ*-edts is denoted by* $Edts^{\mathcal{E}^{\downarrow}}(\Sigma)$*.*

Atomic actions are given by expressions of the form $e /\!\!/ \psi$ with e an event and ψ a state transition predicate. The intuition is that the occurrence of the event e causes a state transition in accordance with ψ, i.e., the pre- and post-data states satisfy ψ, and ψ specifies the possible effects of e. Following the ideas of dynamic logic we also use complex, structured actions formed over atomic actions by union, sequential composition and iteration. All kinds of actions over an ed signature Σ are called Σ*-event/data actions* (Σ*-ed actions*, for short). The set $\Lambda(\Sigma)$ of Σ-ed actions is defined by the grammar

$$\lambda ::= e /\!\!/ \psi \mid \lambda_1 + \lambda_2 \mid \lambda_1; \lambda_2 \mid \lambda^*$$

where $e \in E(\Sigma)$ and $\psi \in \Psi(\Sigma)$. We use the following shorthand notations for actions: For a subset $F = \{e_1, \ldots, e_k\} \subseteq E(\Sigma)$, we use the notation F to denote the complex action $e_1 /\!\!/ \text{true} + \ldots + e_k /\!\!/ \text{true}$ and $-F$ to denote the action $E(\Sigma) \setminus F$. For the action $E(\Sigma)$ we will write $\boldsymbol{E}$. For $e \in E(\Sigma)$, we use the notation e to denote the action $e /\!\!/ \text{true}$ and $-e$ to denote the action $\boldsymbol{E} \setminus \{e\}$. Hence, if $E(\Sigma) = \{e_1, \ldots, e_n\}$ and $e_i \in E(\Sigma)$, the action $-e_i$ stands for $e_1 /\!\!/ \text{true} + \ldots + e_{i-1} /\!\!/ \text{true} + e_{i+1} /\!\!/ \text{true} + \ldots + e_n /\!\!/ \text{true}$.

The actions $\Lambda(\Sigma)$ are *interpreted* over a Σ-edts M as the family of relations $(R(M)_\lambda \subseteq \Gamma(M) \times \Gamma(M))_{\lambda \in \Lambda(\Sigma)}$ defined by

- $R(M)_{e/\!\!/\psi} = \{(\gamma, \gamma') \in R(M)_e \mid (\omega(\gamma), \omega(\gamma')) \models^{\mathcal{D}}_{A(\Sigma)} \psi\}$,
- $R(M)_{\lambda_1 + \lambda_2} = R(M)_{\lambda_1} \cup R(M)_{\lambda_2}$, i.e., union of relations,
- $R(M)_{\lambda_1 ; \lambda_2} = R(M)_{\lambda_1} ; R(M)_{\lambda_2}$, i.e., sequential composition of relations,
- $R(M)_{\lambda^*} = (R(M)_\lambda)^*$, i.e., reflexive-transitive closure of relations.

To define the event/data formulae of $\mathcal{E}^\downarrow$ we assume given a countably infinite set X of control state variables which are used in formulae to denote the control part of a configuration. They can be bound by the binder operator $\downarrow x$ and accessed by the jump operator $@x$ of hybrid logic. The dynamic part of our logic is due to the modalities which can be formed over any ed action over a given ed signature. $\mathcal{E}^\downarrow$ thus retains from hybrid logic the use of binders, but omits free nominals. Thus sentences of the logic become restricted to express properties of configurations reachable from the initial ones.

Definition 3. *The set* $\text{Frm}^{\mathcal{E}^\downarrow}(\Sigma)$ *of* Σ-ed formulae *over an ed signature* Σ *is given by*

$$\varrho ::= \varphi \mid x \mid \downarrow x \,.\, \varrho \mid @x \,.\, \varrho \mid \langle \lambda \rangle \varrho \mid \text{true} \mid \neg \varrho \mid \varrho_1 \vee \varrho_2$$

where $\varphi \in \Phi(\Sigma)$, $x \in X$, *and* $\lambda \in \Lambda(\Sigma)$. *We write* $[\lambda]\varrho$ *for* $\neg\langle\lambda\rangle\neg\varrho$ *and we use the usual boolean connectives as well as the constant* false *to denote* $\neg$true.[1] *The set* $\text{Sen}^{\mathcal{E}^\downarrow}(\Sigma)$ *of* Σ-ed sentences *consists of all* Σ*-ed formulae without free variables, where the free variables are defined as usual with* $\downarrow x$ *being the unique operator binding variables.*

Given an ed signature Σ and a Σ-edts M, the satisfaction of a Σ-ed formula ϱ is inductively defined w.r.t. valuations $v : X \to C(M)$, mapping variables to control states, and configurations $\gamma \in \Gamma(M)$:

- $M, v, \gamma \models^{\mathcal{E}^\downarrow}_\Sigma \varphi$ iff $\omega(\gamma) \models^{\mathcal{D}}_{A(\Sigma)} \varphi$;
- $M, v, \gamma \models^{\mathcal{E}^\downarrow}_\Sigma x$ iff $c(\gamma) = v(x)$;
- $M, v, \gamma \models^{\mathcal{E}^\downarrow}_\Sigma \downarrow x \,.\, \varrho$ iff $M, v\{x \mapsto c(\gamma)\}, \gamma \models^{\mathcal{E}^\downarrow}_\Sigma \varrho$;
- $M, v, \gamma \models^{\mathcal{E}^\downarrow}_\Sigma @x \,.\, \varrho$ iff $M, v, \gamma' \models^{\mathcal{E}^\downarrow}_\Sigma \varrho$ for all $\gamma' \in \Gamma(M)$ with $c(\gamma') = v(x)$;
- $M, v, \gamma \models^{\mathcal{E}^\downarrow}_\Sigma \langle\lambda\rangle\varrho$ iff $M, v, \gamma' \models^{\mathcal{E}^\downarrow}_\Sigma \varrho$ for some $\gamma' \in \Gamma(M)$ with $(\gamma, \gamma') \in R(M)_\lambda$;

[1] We use true and false for predicates and formulae; their meaning will always be clear from the context. For boolean values we will use instead the notations *tt* and *ff*.

- $M, v, \gamma \models^{\mathcal{E}^\downarrow}_{\Sigma}$ true always holds;
- $M, v, \gamma \models^{\mathcal{E}^\downarrow}_{\Sigma} \neg\varrho$ iff $M, v, \gamma \not\models^{\mathcal{E}^\downarrow}_{\Sigma} \varrho$;
- $M, v, \gamma \models^{\mathcal{E}^\downarrow}_{\Sigma} \varrho_1 \vee \varrho_2$ iff $M, v, \gamma \models^{\mathcal{E}^\downarrow}_{\Sigma} \varrho_1$ or $M, v, \gamma \models^{\mathcal{E}^\downarrow}_{\Sigma} \varrho_2$.

If ϱ is a sentence then the valuation is irrelevant. M *satisfies* a sentence $\varrho \in \mathrm{Sen}^{\mathcal{E}^\downarrow}(\Sigma)$, denoted by $M \models^{\mathcal{E}^\downarrow}_{\Sigma} \varrho$, if $M, \gamma_0 \models^{\mathcal{E}^\downarrow}_{\Sigma} \varrho$ for all $\gamma_0 \in \Gamma_0(M)$.

By borrowing the modalities from dynamic logic [9,10], $\mathcal{E}^\downarrow$ is able to express liveness and safety requirements as illustrated in our running ATM example below. There we use the fact that we can state properties over all reachable states by sentences of the form $[\boldsymbol{E}^*]\varphi$. In particular, deadlock-freedom can be expressed by $[\boldsymbol{E}^*]\langle\boldsymbol{E}\rangle$true. The logic $\mathcal{E}^\downarrow$, however, is also suited to directly express process structures and, thus, the implementation of abstract requirements. The binder operator is essential for this. For example, we can specify a process which switches a boolean value, denoted by the attribute val, from tt to ff and back by the following sentence:

$$\downarrow x_0 \,.\, \mathsf{val} = tt \wedge \langle \mathsf{switch} /\!\!/ \mathsf{val}' = ff \rangle \langle \mathsf{switch} /\!\!/ \mathsf{val}' = tt \rangle x_0.$$

2.3 Bisimulation and Invariance

Bisimulation is a crucial notion in both behavioural systems specification and in modal logics. On the specification side, it provides a standard way to identify systems with the same behaviour by abstracting the internal specifics of the systems; this is also reflected at the logic side, where bisimulation frequently relates states that satisfy the same formulae. We explore some properties of $\mathcal{E}^\downarrow$ w.r.t. bisimilarity. Let us first introduce the notion of bisimilarity in the context of $\mathcal{E}^\downarrow$:

Definition 4. *Let M_1, M_2 be Σ-edts. A relation $B \subseteq \Gamma(M_1) \times \Gamma(M_2)$ is a* bisimulation relation *between M_1 and M_2 if for all $(\gamma_1, \gamma_2) \in B$ the following conditions hold:*

(atom) for all $\varphi \in \Phi(\Sigma)$, $\omega(\gamma_1) \models^{\mathcal{D}}_{A(\Sigma)} \varphi$ iff $\omega(\gamma_2) \models^{\mathcal{D}}_{A(\Sigma)} \varphi$;

(zig) for all $e /\!\!/ \psi \in \Lambda(\Sigma)$ and for all $\gamma_1' \in \Gamma(M_1)$ with $(\gamma_1, \gamma_1') \in R(M_1)_{e/\!\!/\psi}$, there is a $\gamma_2' \in \Gamma(M_2)$ such that $(\gamma_2, \gamma_2') \in R(M_2)_{e/\!\!/\psi}$ and $(\gamma_1', \gamma_2') \in B$;

(zag) for all $e /\!\!/ \psi \in \Lambda(\Sigma)$ and for all $\gamma_2' \in \Gamma(M_2)$ with $(\gamma_2, \gamma_2') \in R(M_2)_{e/\!\!/\psi}$, there is a $\gamma_1' \in \Gamma(M_1)$ such that $(\gamma_1, \gamma_1') \in R(M_1)_{e/\!\!/\psi}$ and $(\gamma_1', \gamma_2') \in B$.

M_1 and M_2 are bisimilar*, in symbols $M_1 \sim M_2$, if there exists a bisimulation relation $B \subseteq \Gamma(M_1) \times \Gamma(M_2)$ between M_1 and M_2 such that*

(init) for any $\gamma_1 \in \Gamma_0(M_1)$, there is a $\gamma_2 \in \Gamma_0(M_2)$ such that $(\gamma_1, \gamma_2) \in B$ and for any $\gamma_2 \in \Gamma_0(M_2)$, there is a $\gamma_1 \in \Gamma_0(M_1)$ such that $(\gamma_1, \gamma_2) \in B$.

Now we are able to establish a Hennessy-Milner like correspondence for a fragment of $\mathcal{E}^\downarrow$. Let us call *hybrid-free sentences of* $\mathcal{E}^\downarrow$ the formulae obtained by the grammar

$$\varrho ::= \varphi \mid \langle\lambda\rangle\varrho \mid \text{true} \mid \neg\varrho \mid \varrho_1 \vee \varrho_2.$$

Theorem 1. *Let M_1, M_2 be bisimilar Σ-edts. Then $M_1 \models_\Sigma^{\mathcal{E}^\downarrow} \varrho$ iff $M_2 \models_\Sigma^{\mathcal{E}^\downarrow} \varrho$ for all hybrid-free sentences ϱ.*

The converse of Theorem 1 does not hold, in general, and the usual image-finiteness assumption has to be imposed: A Σ-edts M is *image-finite* if, for all $\gamma \in \Gamma(M)$ and all $e \in E(\Sigma)$, the set $\{\gamma' \mid (\gamma, \gamma') \in R(M)_e\}$ is finite. Then:

Theorem 2. *Let M_1, M_2 be image-finite Σ-edts and $\gamma_1 \in \Gamma(M_1)$, $\gamma_2 \in \Gamma(M_2)$ such that $M_1, \gamma_1 \models_\Sigma^{\mathcal{E}^\downarrow} \varrho$ iff $M_2, \gamma_2 \models_\Sigma^{\mathcal{E}^\downarrow} \varrho$ for all hybrid-free sentences ϱ. Then there exists a bisimulation B between M_1 and M_2 such that $(\gamma_1, \gamma_2) \in B$.*

3 Specifications of Event/Data Systems

3.1 Axiomatic Specifications

Sentences of $\mathcal{E}^\downarrow$-logic can be used to specify properties of event/data systems and thus to write system specifications in an axiomatic way.

Definition 5. *An* axiomatic ed specification $Sp = (\Sigma(Sp), Ax(Sp))$ *in* $\mathcal{E}^\downarrow$ *consists of an ed signature* $\Sigma(Sp) \in Sig^{\mathcal{E}^\downarrow}$ *and a set of* axioms $Ax(Sp) \subseteq \mathrm{Sen}^{\mathcal{E}^\downarrow}(\Sigma(Sp))$.

The semantics of *Sp is given by the pair* $(\Sigma(Sp), \mathrm{Mod}(Sp))$ *where* $\mathrm{Mod}(Sp) = \{M \in Edts^{\mathcal{E}^\downarrow}(\Sigma(Sp)) \mid M \models_{\Sigma(Sp)}^{\mathcal{E}^\downarrow} Ax(Sp)\}$. *The edts in* $\mathrm{Mod}(Sp)$ *are called* models *of Sp and* $\mathrm{Mod}(Sp)$ *is the* model class *of Sp.*

As a direct consequence of Theorem 1 we have:

Corollary 1. *The model class of an axiomatic ed specification exclusively expressed by hybrid-free sentences is closed under bisimulation.*

This result does not hold for sentences with hybrid features. For instance, consider the specification $Sp = \big((\{e\}, \{a\}), \{\downarrow x \,.\, \langle e /\!\!/ a' = a \rangle x\}\big)$: An edts with a single control state c_0 and a loop transition $R_e = \{(\gamma_0, \gamma_0)\}$ for $c(\gamma_0) = c_0$ is a model of Sp. However, this is obviously not the case for its bisimilar edts with two control states c_0 and c and the relation $R'_e = \{(\gamma_0, \gamma), (\gamma, \gamma_0)\}$ with $c(\gamma_0) = c_0$, $c(\gamma) = c$ and $\omega(\gamma_0) = \omega(\gamma)$.

Example 1. As a running example we consider an ATM. We start with an abstract specification Sp_0 of fundamental requirements for its interaction behaviour based on the set of events $E_0 = \{\mathsf{insertCard}, \mathsf{enterPIN}, \mathsf{ejectCard}, \mathsf{cancel}\}$[2] and on the singleton set of attributes $A_0 = \{\mathsf{chk}\}$ where chk is boolean valued and records the correctness of an entered PIN. Hence our first ed signature is $\Sigma_0 = (E_0, A_0)$ and $Sp_0 = (\Sigma_0, Ax_0)$ where Ax_0 requires the following properties expressed by corresponding axioms (0.1–0.3):

[2] For shortening the presentation we omit further events like withdrawing money, etc.

– "Whenever a card has been inserted, a correct PIN can eventually be entered and also the transaction can eventually be cancelled."

$$[\boldsymbol{E}^*; \mathsf{insertCard}](\langle \boldsymbol{E}^*; \mathsf{enterPIN} /\!\!/ \mathsf{chk}' = tt\rangle \mathrm{true} \wedge \langle \boldsymbol{E}^*; \mathsf{cancel}\rangle \mathrm{true}) \tag{0.1}$$

– "Whenever either a correct PIN has been entered or the transaction has been cancelled, the card can eventually be ejected."

$$[\boldsymbol{E}^*; (\mathsf{enterPIN} /\!\!/ \mathsf{chk}' = tt) + \mathsf{cancel}]\langle \boldsymbol{E}^*; \mathsf{ejectCard}\rangle \mathrm{true} \tag{0.2}$$

– "Whenever an incorrect PIN has been entered three times in a row, the current card is not returned." This means that the card is kept by the ATM which is not modelled by an extra event. It may, however, still be possible that another card is inserted afterwards. So an ejectCard can only be forbidden as long as no next card is inserted.

$$[\boldsymbol{E}^*; (\mathsf{enterPIN} /\!\!/ \mathsf{chk}' = ff)^3; (-\mathsf{insertCard})^*; \mathsf{ejectCard}]\mathrm{false} \tag{0.3}$$

where λ^n abbreviates the n-fold sequential composition $\lambda; \ldots; \lambda$.

The semantics of an axiomatic ed specification is loose allowing usually for many different realisations. A refinement step is therefore understood as a restriction of the model class of an abstract specification. Following the terminology of Sannella and Tarlecki [18,19], we call a specification refining another one an *implementation*. Formally, a specification Sp' is a *simple implementation* of a specification Sp over the same signature, in symbols $Sp \rightsquigarrow Sp'$, whenever $\mathrm{Mod}(Sp) \supseteq \mathrm{Mod}(Sp')$. Transitivity of the inclusion relation ensures gradual step-by-step development by a series of refinements.

Example 2. We provide a refinement $Sp_0 \rightsquigarrow Sp_1$ where $Sp_1 = (\Sigma_0, Ax_1)$ has the same signature as Sp_0 and Ax_1 are the sentences (1.1–1.4) below; the last two use binders to specify a loop. As is easily seen, all models of Sp_1 must satisfy the axioms of Sp_0.

– "At the beginning a card can be inserted with the effect that chk is set to ff; nothing else is possible at the beginning."

$$\begin{aligned} &\langle \mathsf{insertCard} /\!\!/ \mathsf{chk}' = ff\rangle \mathrm{true} \wedge \\ &[\mathsf{insertCard} /\!\!/ \neg(\mathsf{chk}' = ff)]\mathrm{false} \wedge [-\mathsf{insertCard}]\mathrm{false} \end{aligned} \tag{1.1}$$

– "Whenever a card has been inserted, a PIN can be entered (directly afterwards) and also the transaction can be cancelled; but nothing else."

$$\begin{aligned} [\boldsymbol{E}^*; \mathsf{insertCard}](&\langle \mathsf{enterPIN}\rangle \mathrm{true} \wedge \langle \mathsf{cancel}\rangle \mathrm{true} \wedge \\ &[-\{\mathsf{enterPIN}, \mathsf{cancel}\}]\mathrm{false}) \end{aligned} \tag{1.2}$$

– "Whenever either a correct PIN has been entered or the transaction has been cancelled, the card can eventually be ejected and the ATM starts from the beginning."

$$\downarrow x_0 \,.\, [\boldsymbol{E}^*; (\mathsf{enterPIN} /\!\!/ \mathsf{chk}' = tt) + \mathsf{cancel}] \langle \boldsymbol{E}^*; \mathsf{ejectCard} \rangle x_0 \tag{1.3}$$

– "Whenever an incorrect PIN has been entered three times in a row the ATM starts from the beginning." Hence the current card is kept.

$$\downarrow x_0 \,.\, [\boldsymbol{E}^*; (\mathsf{enterPIN} /\!\!/ \mathsf{chk}' = ff)^3] x_0 \tag{1.4}$$

3.2 Operational Specifications

Operational event/data specifications are introduced as a means to specify in a more constructive style the properties of event/data systems. They are not appropriate for writing abstract requirements for which axiomatic specifications are recommended. Though $\mathcal{E}^{\downarrow}$-logic is able to specify concrete models, as discussed in Sect. 2, the use of operational specifications allows a graphic representation close to familiar formalisms in the literature, like UML protocol state machines, cf. [12,16]. As will be shown in Sect. 3.3, finite operational specifications can be characterised by a sentence in $\mathcal{E}^{\downarrow}$-logic. Therefore, $\mathcal{E}^{\downarrow}$-logic is still the common basis of our development approach. Transitions in an operational specification are tuples $(c, \varphi, e, \psi, c')$ with c a source control state, φ a precondition, e an event, ψ a state transition predicate specifying the possible effects of the event e, and c' a target control state. In the semantic models an event must be enabled whenever the respective source data state satisfies the precondition. Thus isolating preconditions has a semantic consequence that is not expressible by transition predicates only. The effect of the event must respect ψ; no other transitions are allowed.

Definition 6. *An* operational ed specification $O = (\Sigma, C, T, (c_0, \varphi_0))$ *is given by an ed signature* Σ*, a set of* control states C*, a* transition relation specification $T \subseteq C \times \Phi(\Sigma) \times E(\Sigma) \times \Psi(\Sigma) \times C$*, an* initial control state $c_0 \in C$*, and an* initial state predicate $\varphi_0 \in \Phi(\Sigma)$*, such that* C *is* syntactically reachable*, i.e., for every* $c \in C \setminus \{c_0\}$ *there are* $(c_0, \varphi_1, e_1, \psi_1, c_1), \ldots, (c_{n-1}, \varphi_n, e_n, \psi_n, c_n) \in T$ *with* $n > 0$ *such that* $c_n = c$*. We write* $\Sigma(O)$ *for* Σ*, etc.*

A Σ*-edts* M *is a* model *of* O *if* $C(M) = C$ *up to a bijective renaming,* $c_0(M) = c_0$*,* $\Omega_0(M) \subseteq \{\omega \mid \omega \models^{\mathcal{D}}_{A(\Sigma)} \varphi_0\}$*, and if the following conditions hold:*

– *for all* $(c, \varphi, e, \psi, c') \in T$ *and* $\omega \in \Omega(A(\Sigma))$ *with* $\omega \models^{\mathcal{D}}_{A(\Sigma)} \varphi$*, there is a* $((c, \omega), (c', \omega')) \in R(M)_e$ *with* $(\omega, \omega') \models^{\mathcal{D}}_{A(\Sigma)} \psi$*;*

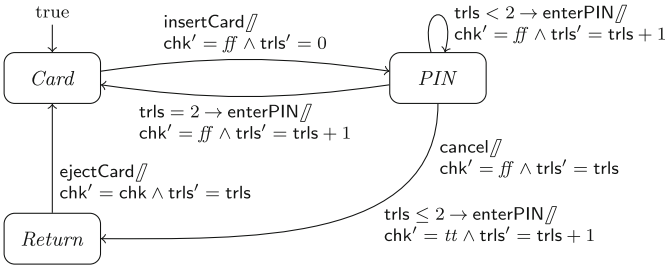


Fig. 1. Operational ed specification *ATM*

– *for all* $((c,\omega),(c',\omega')) \in R(M)_e$ *there is a* $(c,\varphi,e,\psi,c') \in T$ *with* $\omega \models^{\mathcal{D}}_{A(\Sigma)} \varphi$ *and* $(\omega,\omega') \models^{\mathcal{D}}_{A(\Sigma)} \psi$.

The class of all models of O *is denoted by* $\mathrm{Mod}(O)$. *The* semantics *of* O *is given by the pair* $(\Sigma(O), \mathrm{Mod}(O))$ *where* $\Sigma(O) = \Sigma$.

Example 3. We construct an operational ed specification, called *ATM*, for the ATM example. The signature of *ATM* extends the one of Sp_1 (and Sp_0) by an additional integer-valued attribute trls which counts the number of attempts to enter a correct PIN (with the same card). *ATM* is graphically presented in Fig. 1. The initial control state is *Card*, and the initial state predicate is true. Preconditions are written before the symbol $\rightarrow$. If no precondition is explicitly indicated it is assumed to be true. Due to the extended signature, *ATM* is not a simple implementation of Sp_1, and we will only formally justify the implementation relationship in Example 5.

Operational specifications can be composed by a syntactic parallel composition operator which synchronises shared events. Two ed signatures Σ_1 and Σ_2 are *composable* if $A(\Sigma_1) \cap A(\Sigma_2) = \emptyset$. Their parallel composition is given by $\Sigma_1 \otimes \Sigma_2 = (E(\Sigma_1) \cup E(\Sigma_2), A(\Sigma_1) \cup A(\Sigma_2))$.

Definition 7. *Let* Σ_1 *and* Σ_2 *be composable ed signatures and let* O_1 *and* O_2 *be operational ed specifications with* $\Sigma(O_1) = \Sigma_1$ *and* $\Sigma(O_2) = \Sigma_2$. *The* parallel composition *of* O_1 *and* O_2 *is given by the operational ed specification* $O_1 \parallel O_2 = (\Sigma_1 \otimes \Sigma_2, C, T, (c_0, \varphi_0))$ *with* $c_0 = (c_0(O_1), c_0(O_2))$, $\varphi_0 = \varphi_0(O_1) \wedge \varphi_0(O_2)$, *and* C *and* T *are inductively defined by* $c_0 \in C$ *and*

– *for* $e_1 \in E(\Sigma_1) \setminus E(\Sigma_2)$, $c_1, c_1' \in C(O_1)$, *and* $c_2 \in C(O_2)$, *if* $(c_1,c_2) \in C$ *and* $(c_1,\varphi_1,e_1,\psi_1,c_1') \in T(O_1)$, *then* $(c_1',c_2) \in C$ *and* $((c_1,c_2),\varphi_1,e_1,\psi_1 \wedge \mathrm{id}_{A(\Sigma_2)}, (c_1',c_2)) \in T$;
– *for* $e_2 \in E(\Sigma_2) \setminus E(\Sigma_1)$, $c_2, c_2' \in C(O_2)$, *and* $c_1 \in C(O_1)$, *if* $(c_1,c_2) \in C$ *and* $(c_2,\varphi_2,e_2,\psi_2,c_2') \in T(O_2)$, *then* $(c_1,c_2') \in C$ *and* $((c_1,c_2),\varphi_2,e_2,\psi_2 \wedge \mathrm{id}_{A(\Sigma_1)}, (c_1,c_2')) \in T$;

– *for* $e \in E(\Sigma_1) \cap E(\Sigma_2)$, $c_1, c_1' \in C(O_1)$, *and* $c_2, c_2' \in C(O_2)$, *if* $(c_1, c_2) \in C$, $(c_1, \varphi_1, e, \psi_1, c_1') \in T(O_1)$, *and* $(c_2, \varphi_2, e, \psi_2, c_2') \in T(O_2)$, *then* $(c_1', c_2') \in C$ *and* $((c_1, c_2), \varphi_1 \wedge \varphi_2, e, \psi_1 \wedge \psi_2, (c_1', c_2')) \in T$.[3]

3.3 Expressiveness of $\mathcal{E}^{\downarrow}$-Logic

We show that the semantics of an operational ed specification O with finitely many control states can be characterised by a single $\mathcal{E}^{\downarrow}$-sentence ϱ_O, i.e., an edts M is a model of O iff $M \models^{\mathcal{E}^{\downarrow}}_{\Sigma(O)} \varrho_O$. Using Algorithm 1, such a characterising sentence is

$$\varrho_O = \downarrow c_0 \,.\, \varphi_0 \wedge \mathrm{sen}(c_0, Im_O(c_0), C(O), \{c_0\}) \;,$$

where $c_0 = c_0(O)$ and $\varphi_0 = \varphi_0(O)$. Algorithm 1 closely follows the procedure in [15] for characterising a finite structure by a sentence of $\mathcal{D}^{\downarrow}$-logic. A call $\mathrm{sen}(c, I, V, B)$ performs a recursive breadth-first traversal through O starting from c, where I holds the unprocessed quadruples (φ, e, ψ, c') of transitions outgoing from c, V the remaining states to visit, and B the set of already bound states. The function first requires the existence of each outgoing transition of I, provided its precondition holds, in the resulting formula, binding any newly reached state. Then it requires that no other transitions with source state c exist using calls to fin. Having visited all states in V, it finally requires all states in $C(O)$ to be pairwise different.

Algorithm 1. Constructing a sentence from an operational ed specification

Require: $O \equiv$ finite operational ed specification
$Im_O(c) = \{(\varphi, e, \psi, c') \mid (c, \varphi, e, \psi, c') \in T(O)\}$ for $c \in C(O)$
$Im_O(c, e) = \{(\varphi, \psi, c') \mid (c, \varphi, e, \psi, c') \in T(O)\}$ for $c \in C(O)$, $e \in E(\Sigma(O))$

function $\mathrm{sen}(c, I, V, B)$ ▷ c: state, I: image to visit, V: states to visit, B: bound states
  if $I \neq \emptyset$ **then**
    $(\varphi, e, \psi, c') \leftarrow$ **choose** I
    if $c' \in B$ **then**
      return $@c \,.\, \varphi \rightarrow \langle e /\!\!/ \psi \rangle (c' \wedge \mathrm{sen}(c, I \setminus \{(\varphi, e, \psi, c')\}, V, B))$
    else
      return $@c \,.\, \varphi \rightarrow \langle e /\!\!/ \psi \rangle (\downarrow c' \,.\, \mathrm{sen}(c, I \setminus \{(\varphi, e, \psi, c')\}, V, B \cup \{c'\}))$
  $V \leftarrow V \setminus \{c\}$
  if $V \neq \emptyset$ **then**
    $c' \leftarrow$ **choose** $B \cap V$
    return $\mathrm{fin}(c) \wedge \mathrm{sen}(c', Im_O(c'), V, B)$
  return $\mathrm{fin}(c) \wedge \bigwedge_{c_1 \in C(O), c_2 \in C(O) \setminus \{c_1\}} \neg @c_1 \,.\, c_2$

function $\mathrm{fin}(c)$
  return $@c \,.\, \bigwedge_{e \in E(\Sigma(O))} \bigwedge_{P \subseteq Im_O(c,e)} [e /\!\!/ \big(\bigwedge_{(\varphi,\psi,c') \in P} (\varphi \wedge \psi)\big) \wedge \neg\big(\bigvee_{(\varphi,\psi,c') \in Im_O(c,e) \setminus P} (\varphi \wedge \psi)\big)]\big(\bigvee_{(\varphi,\psi,c') \in P} c'\big)$

[3] Note that joint moves with e cannot become inconsistent due to composability of ed signatures.

It is $\mathrm{fin}(c)$ where this algorithm mainly deviates from [15]: To ensure that no other transitions from c exist than those specified in O, $\mathrm{fin}(c)$ produces the requirement that at state c, for every event e and for every subset P of the transitions outgoing from c, whenever an e-transition can be done with the combined effect of P but not adhering to any of the effects of the currently not selected transitions, the e-transition must have one of the states as its target that are target states of P. The rather complicated formulation is due to possibly overlapping preconditions where for a single event e the preconditions of two different transitions may be satisfied simultaneously. For a state c, where all outgoing transitions for the same event have disjoint preconditions, the $\mathcal{E}^{\downarrow}$-formula returned by $\mathrm{fin}(c)$ is equivalent to

$$@c\,.\,\bigwedge_{e\in E(\Sigma(O))}\bigwedge_{(\varphi,\psi,c')\in Im_O(c,e)}[e/\!\!/\varphi\wedge\psi]c'\wedge [e/\!\!/\neg(\bigvee_{(\varphi,\psi,c')\in Im_O(c,e)}(\varphi\wedge\psi))]\mathrm{false}.$$

Example 4. We show the first few steps of representing the operational ed specification ATM of Fig. 1 as an $\mathcal{E}^{\downarrow}$-sentence ϱ_{ATM}. This top-level sentence is

$$\downarrow Card\,.\,\mathrm{true}\wedge \mathrm{sen}(Card,\{(\mathrm{true},\mathsf{insertCard},\mathsf{chk}'=\mathit{ff}\wedge\mathsf{trls}'=0,PIN)\},\{Card,PIN,Return\},\{Card\}).$$

The first call of $\mathrm{sen}(Card,\ldots)$ explores the single outgoing transition from $Card$ to PIN, adds PIN to the bound states, and hence expands to

$$@Card\,.\,\mathrm{true}\rightarrow\langle\mathsf{insertCard}/\!\!/\mathsf{chk}'=\mathit{ff}\wedge\mathsf{trls}'=0\rangle\downarrow PIN.\ \mathrm{sen}(Card,\emptyset,\{Card,PIN,Return\},\{Card,PIN\}).$$

Now all outgoing transitions from $Card$ have been explored and the next call of $\mathrm{sen}(Card,\emptyset,\ldots)$ removes $Card$ from the set of states to be visited, resulting in

$$\mathrm{fin}(Card)\wedge\mathrm{sen}(PIN,\{(\mathsf{trls}<2,\mathsf{enterPIN},\ldots),(\mathsf{trls}=2,\mathsf{enterPIN},\ldots),(\mathsf{trls}\leq 2,\mathsf{enterPIN},\ldots),(\mathrm{true},\mathsf{cancel},\ldots)\},\{PIN,Return\},\{Card,PIN\}).$$

As there is only a single outgoing transition from $Card$, the special case of disjoint preconditions applies for the finalisation call, and $\mathrm{fin}(Card)$ results in

$$@Card\,.\,[\mathsf{insertCard}/\!\!/\mathsf{chk}'=\mathit{ff}\wedge\mathsf{trls}'=0]PIN\wedge [\mathsf{insertCard}/\!\!/\mathsf{chk}'=\mathit{tt}\vee\mathsf{trls}'\neq 0]\mathrm{false}\wedge [\mathsf{enterPIN}/\!\!/\mathrm{true}]\mathrm{false}\wedge[\mathsf{cancel}/\!\!/\mathrm{true}]\mathrm{false}\wedge[\mathsf{ejectCard}/\!\!/\mathrm{true}]\mathrm{false}.$$

4 Constructor Implementations

The implementation notion defined in Sect. 3.1 is too simple for many practical applications. It requires the same signature for specification and implementation and does not support the process of constructing an implementation. Therefore,

Sannella and Tarlecki [18,19] have proposed the notion of constructor implementation which is a generic notion applicable to specification formalisms which are based on signatures and semantic structures for signatures. We will reuse the ideas in the context of $\mathcal{E}^{\downarrow}$-logic.

The notion of *constructor* is the basis: for signatures $\Sigma_1, \ldots, \Sigma_n, \Sigma \in Sig^{\mathcal{E}^{\downarrow}}$, a *constructor* κ from $(\Sigma_1, \ldots, \Sigma_n)$ to Σ is a (total) function $\kappa : Edts^{\mathcal{E}^{\downarrow}}(\Sigma_1) \times \ldots \times Edts^{\mathcal{E}^{\downarrow}}(\Sigma_n) \to Edts^{\mathcal{E}^{\downarrow}}(\Sigma)$. Given a constructor κ from $(\Sigma_1, \ldots, \Sigma_n)$ to Σ and a set of constructors κ_i from $(\Sigma_i^1, \ldots, \Sigma_i^{k_i})$ to Σ_i, $1 \leq i \leq n$, the constructor $(\kappa_1, \ldots, \kappa_n); \kappa$ from $(\Sigma_1^1, \ldots, \Sigma_1^{k_1}, \ldots, \Sigma_n^1, \ldots, \Sigma_n^{k_n})$ to Σ is obtained by the usual composition of functions. The following definitions apply to both axiomatic and operational ed specifications since the semantics of both is given in terms of ed signatures and model classes of edts. In particular, the implementation notion allows to implement axiomatic specifications by operational specifications.

Definition 8. *Given specifications $Sp, Sp_1, \ldots, Sp_n$ and a constructor κ from $(\Sigma(Sp_1), \ldots, \Sigma(Sp_n))$ to $\Sigma(Sp)$, the tuple $\langle Sp_1, \ldots, Sp_n \rangle$ is a* constructor implementation via *κ of Sp, in symbols $Sp \rightsquigarrow_\kappa \langle Sp_1, \ldots, Sp_n \rangle$, if for all $M_i \in \mathrm{Mod}(Sp_i)$ we have $\kappa(M_1, \ldots, M_n) \in \mathrm{Mod}(Sp)$. The implementation involves a* decomposition *if $n > 1$.*

The notion of simple implementation in Sect. 3.1 is captured by choosing the identity. We now introduce a set of more advanced constructors in the context of ed signatures and edts. Let us first consider two central notions for constructors: signature morphisms and reducts. For data signatures A, A' a *data signature morphism* $\sigma : A \to A'$ is a function from A to A'. The *σ-reduct* of an A'-data state $\omega' : A' \to \mathcal{D}$ is given by the A-data state $\omega'|\sigma : A \to \mathcal{D}$ defined by $(\omega'|\sigma)(a) = \omega'(\sigma(a))$ for every $a \in A$. If $A \subseteq A'$, the injection of A into A' is a particular data signature morphism and we denote the reduct of an A'-data state ω' to A by $\omega' \restriction A$. If $A = A_1 \cup A_2$ is the disjoint union of A_1 and A_2 and ω_i are A_i-data states for $i \in \{1, 2\}$ then $\omega_1 + \omega_2$ denotes the unique A-data state ω with $\omega \restriction A_i = \omega_i$ for $i \in \{1, 2\}$. The σ-reduct $\gamma|\sigma$ of a configuration $\gamma = (c, \omega')$ is given by $(c, \omega'|\sigma)$, and is lifted to a set of configurations Γ' by $\Gamma'|\sigma = \{\gamma'|\sigma \mid \gamma' \in \Gamma'\}$.

Definition 9. *An* ed signature morphism *$\sigma = (\sigma_E, \sigma_A) : \Sigma \to \Sigma'$ is given by a function $\sigma_E : E(\Sigma) \to E(\Sigma')$ and a data signature morphism $\sigma_A : A(\Sigma) \to A(\Sigma')$. We abbreviate both σ_E and σ_A by σ.*

Definition 10. *Let $\sigma : \Sigma \to \Sigma'$ be an ed signature morphism and M' a Σ'-edts. The* σ-reduct *of M' is the Σ-edts $M'|\sigma = (\Gamma, R, \Gamma_0)$ such that $\Gamma_0 = \Gamma_0(M')|\sigma$, and Γ and $R = (R_e)_{e \in E(\Sigma)}$ are inductively defined by $\Gamma_0 \subseteq \Gamma$ and for all $e \in E(\Sigma)$, $\gamma', \gamma'' \in \Gamma(M')$: if $\gamma'|\sigma \in \Gamma$ and $(\gamma', \gamma'') \in R(M')_{\sigma(e)}$, then $\gamma''|\sigma \in \Gamma$ and $(\gamma'|\sigma, \gamma''|\sigma) \in R_e$.*

Definition 11. *Let $\sigma : \Sigma \to \Sigma'$ be an ed signature morphism. The* reduct constructor *κ_σ from Σ' to Σ maps any $M' \in Edts^{\mathcal{E}^{\downarrow}}(\Sigma')$ to its reduct $\kappa_\sigma(M') = M'|\sigma$. Whenever σ_A and σ_E are bijective functions, κ_σ is a* relabelling constructor. *If σ_E and σ_A are injective, κ_σ is a* restriction constructor.

Example 5. The operational specification ATM is a constructor implementation of Sp_1 via the restriction constructor κ_ι determined by the inclusion signature morphism $\iota : \Sigma(Sp_1) \to \Sigma(ATM)$, i.e., $Sp_1 \rightsquigarrow_{\kappa_\iota} ATM$.

A further refinement technique for reactive systems (see, e.g., [8]), is the implementation of simple events by complex events, like their sequential composition. To formalise this as a constructor we use *composite events* $\Theta(E)$ over a given set of events E, given by the grammar $\theta ::= e \mid \theta + \theta \mid \theta;\theta \mid \theta^*$ with $e \in E$. They are *interpreted* over an (E, A)-edts M by $R(M)_{\theta_1+\theta_2} = R(M)_{\theta_1} \cup R(M)_{\theta_2}$, $R(M)_{\theta_1;\theta_2} = R(M)_{\theta_1}; R(M)_{\theta_2}$, and $R(M)_{\theta^*} = (R(M)_\theta)^*$. Then we can introduce the intended constructor by means of reducts over signature morphisms mapping atomic to composite events:

Definition 12. *Let Σ, Σ' be ed signatures, D' a finite subset of $\Theta(E(\Sigma'))$, $\Delta' = (D', A(\Sigma'))$, and $\alpha : \Sigma \to \Delta'$ an ed signature morphism. The* event refinement *constructor κ_α from Δ' to Σ maps any $M' \in Edts^{\mathcal{E}^\downarrow}(\Delta')$ to its reduct $M'|\alpha \in Edts^{\mathcal{E}^\downarrow}(\Sigma)$.*

Finally, we consider a semantic, synchronous parallel composition constructor that allows for decomposition of implementations into components which synchronise on shared events. Given two composable signatures Σ_1 and Σ_2, the *parallel composition* $\gamma_1 \otimes \gamma_2$ of two configurations $\gamma_1 = (c_1, \omega_1)$, $\gamma_2 = (c_2, \omega_2)$ with $\omega_1 \in \Omega(A(\Sigma_1))$, $\omega_2 \in \Omega(A(\Sigma_2))$ is given by $((c_1, c_2), \omega_1 + \omega_2)$, and lifted to two sets of configurations Γ_1 and Γ_2 by $\Gamma_1 \otimes \Gamma_2 = \{\gamma_1 \otimes \gamma_2 \mid \gamma_1 \in \Gamma_1,\ \gamma_2 \in \Gamma_2\}$.

Definition 13. *Let Σ_1, Σ_2 be composable ed signatures. The* parallel composition constructor $\kappa_\otimes$ *from (Σ_1, Σ_2) to $\Sigma_1 \otimes \Sigma_2$ maps any $M_1 \in Edts^{\mathcal{E}^\downarrow}(\Sigma_1)$, $M_2 \in Edts^{\mathcal{E}^\downarrow}(\Sigma_2)$ to $M_1 \otimes M_2 = (\Gamma, R, \Gamma_0) \in Edts^{\mathcal{E}^\downarrow}(\Sigma_1 \otimes \Sigma_2)$, where $\Gamma_0 = \Gamma_0(M_1) \otimes \Gamma_0(M_2)$, and Γ and $R = (R_e)_{E(\Sigma_1)\cup E(\Sigma_2)}$ are inductively defined by $\Gamma_0 \subseteq \Gamma$ and*

- *for all $e_1 \in E(\Sigma_1) \setminus E(\Sigma_2)$, $\gamma_1, \gamma_1' \in \Gamma(M_1)$, and $\gamma_2 \in \Gamma(M_2)$, if $\gamma_1 \otimes \gamma_2 \in \Gamma$ and $(\gamma_1, \gamma_1') \in R(M_1)_{e_1}$, then $\gamma_1' \otimes \gamma_2 \in \Gamma$ and $(\gamma_1 \otimes \gamma_2, \gamma_1' \otimes \gamma_2) \in R_{e_1}$;*
- *for all $e_2 \in E(\Sigma_2) \setminus E(\Sigma_1)$, $\gamma_2, \gamma_2' \in \Gamma(M_2)$, and $\gamma_1 \in \Gamma(M_1)$, if $\gamma_1 \otimes \gamma_2 \in \Gamma$ and $(\gamma_2, \gamma_2') \in R(M_2)_{e_2}$, then $\gamma_1 \otimes \gamma_2' \in \Gamma$ and $(\gamma_1 \otimes \gamma_2, \gamma_1 \otimes \gamma_2') \in R_{e_2}$;*
- *for all $e \in E(\Sigma_1) \cap E(\Sigma_2)$, $\gamma_1, \gamma_1' \in \Gamma(M_1)$, and $\gamma_2, \gamma_2' \in \Gamma(M_2)$, if $\gamma_1 \otimes \gamma_2 \in \Gamma$, $(\gamma_1, \gamma_1') \in R(M_1)_{e_1}$, and $(\gamma_2, \gamma_2') \in R(M_2)_{e_2}$, then $\gamma_1' \otimes \gamma_2' \in \Gamma$ and $(\gamma_1 \otimes \gamma_2, \gamma_1' \otimes \gamma_2') \in R_e$.*

An obvious question is how the semantic parallel composition constructor is related to the syntactic parallel composition of operational ed specifications.

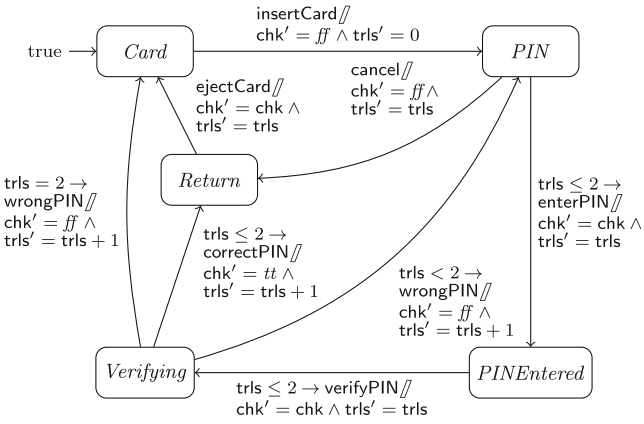
Proposition 1. *Let O_1, O_2 be operational ed specifications with composable signatures. Then* $\mathrm{Mod}(O_1) \otimes \mathrm{Mod}(O_2) \subseteq \mathrm{Mod}(O_1 \parallel O_2)$, *where* $\mathrm{Mod}(O_1) \otimes \mathrm{Mod}(O_2)$ *denotes* $\kappa_\otimes(\mathrm{Mod}(O_1), \mathrm{Mod}(O_2))$.

The converse $\mathrm{Mod}(O_1 \parallel O_2) \subseteq \mathrm{Mod}(O_1) \otimes \mathrm{Mod}(O_2)$ does not hold: Consider the ed signature $\Sigma = (E, A)$ with $E = \{e\}$, $A = \emptyset$, and the operational ed specifications $O_i = (\Sigma, C_i, T_i, (c_{i,0}, \varphi_{i,0}))$ for $i \in \{1, 2\}$ with $C_1 = \{c_{1,0}\}$, $T_1 = \{(c_{1,0}, \mathrm{true}, e, \mathrm{false}, c_{1,0})\}$, $\varphi_{1,0} = \mathrm{true}$; and $C_2 = \{c_{2,0}\}$, $T_2 = \emptyset$, $\varphi_{2,0} = \mathrm{true}$. Then $\mathrm{Mod}(O_1) = \emptyset$, but $\mathrm{Mod}(O_1 \parallel O_2) = \{M\}$ with M showing just the initial configuration.

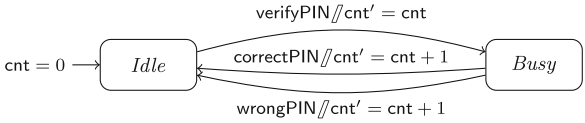
The next theorem shows the usefulness of the syntactic parallel composition operator for proving implementation correctness when a (semantic) parallel composition constructor is involved. The theorem is a direct consequence of Proposition 1 and Definition 8.

Theorem 3. *Let Sp be an (axiomatic or operational) ed specification, O_1, O_2 operational ed specifications with composable signatures, and κ an implementation constructor from $\Sigma(O_1) \otimes \Sigma(O_2)$ to $\Sigma(Sp)$: If $Sp \rightsquigarrow_\kappa O_1 \parallel O_2$, then $Sp \rightsquigarrow_{\kappa_\otimes;\kappa} \langle O_1, O_2 \rangle$.*

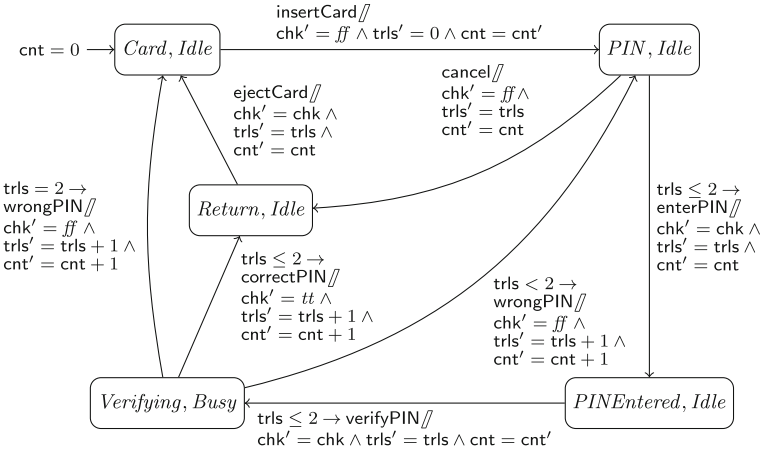
Example 6. We finish the refinement chain for the ATM specifications by applying a decomposition into two parallel components. The operational specification *ATM* of Example 3 (and Example 5) describes the interface behaviour of an ATM interacting with a user. For a concrete realisation, however, an ATM will also interact internally with other components, like, e.g., a clearing company which supports the ATM for verifying PINs. Our last refinement step hence realises the *ATM* specification by two parallel components, represented by the operational specification ATM' in Fig. 2a and the operational specification *CC* of a clearing company in Fig. 2b. Both communicate (via shared events) when an ATM sends a verification request, modelled by the event verifyPIN, to the clearing company. The clearing company may answer with correctPIN or wrongPIN and then the ATM continues following its specification. For the implementation construction we use the parallel composition constructor $\kappa_\otimes$ from $(\Sigma(ATM'), \Sigma(CC))$ to $\Sigma(ATM') \otimes \Sigma(CC)$. The signature of *CC* consists of the events shown on the transitions in Fig. 2b. Moreover, there is one integer-valued attribute cnt counting the number of verification tasks performed. The signature of ATM' extends $\Sigma(ATM)$ by the events verifyPIN, correctPIN and wrongPIN. To fit the signature and the behaviour of the parallel composition of ATM' and *CC* to the specification *ATM* we must therefore compose $\kappa_\otimes$ with an event refinement constructor κ_α such that $\alpha(\mathsf{enterPIN}) = (\mathsf{enterPIN}; \mathsf{verifyPIN}; (\mathsf{correctPIN} + \mathsf{wrongPIN}))$; for the other events α is the identity and for the attributes the inclusion. The idea is therefore that the refinement looks like $ATM \rightsquigarrow_{\kappa_\otimes;\kappa_\alpha} \langle ATM', CC \rangle$. To prove this refinement relation we rely on the syntactic parallel composition $ATM' \parallel CC$ shown in Fig. 2c, and on Theorem 3. It is easy to see that $ATM \rightsquigarrow_{\kappa_\alpha} ATM' \parallel CC$. In fact, all transitions for event enterPIN in Fig. 1 are split into several transitions in Fig. 2c according to the event refinement defined by α. For instance, the loop transition from *PIN* to *PIN* with precondition $\mathsf{trls} < 2$ in Fig. 1 is split into



(a) Operational ed specification ATM'



(b) Operational specification CC of a clearing company



(c) Syntactic parallel composition $ATM' \parallel CC$

Fig. 2. Operational ed specifications ATM', CC and their parallel composition

the cycle from $(PIN, Idle)$ via $(PINEntered, Idle)$ and $(Verifying, Busy)$ back to $(PIN, Idle)$ in Fig. 2c. Thus, we have $ATM \rightsquigarrow_{\kappa_\alpha} ATM' \parallel CC$ and can apply Theorem 3 such that we get $ATM \rightsquigarrow_{\kappa_\otimes ; \kappa_\alpha} \langle ATM', CC \rangle$.

5 Conclusions

We have presented a novel logic, called $\mathcal{E}^{\downarrow}$-logic, for the rigorous formal development of event-based systems incorporating changing data states. To the best of our knowledge, no other logic supports the full development process for this kind of systems ranging from abstract requirements specifications, expressible by the dynamic logic features, to the concrete specification of implementations, expressible by the hybrid part of the logic.

The temporal logic of actions (TLA [13]) supports also stepwise refinement where state transition predicates are considered as actions. In contrast to TLA we model also the events which cause data state transitions. For writing concrete specifications we have proposed an operational specification format capturing (at least parts of) similar formalisms, like Event-B [1], symbolic transition systems [17], and UML protocol state machines [16]. A significant difference to Event-B machines is that we distinguish between control and data states, the former being encoded as data in Event-B. On the other hand, Event-B supports parameters of events which could be integrated in our logic as well. An institution-based semantics of Event-B has been proposed in [7] which coincides with our semantics of operational specifications for the special case of deterministic state transition predicates. Similarly, our semantics of operational specifications coincides with the unfolding of symbolic transition systems in [17] if we instantiate our generic data domain with algebraic specifications of data types (and consider again only deterministic state transition predicates). The syntax of UML protocol state machines is about the same as the one of operational event/data specifications. As a consequence, all of the aforementioned concrete specification formalisms (and several others) would be appropriate candidates for integration into a development process based on $\mathcal{E}^{\downarrow}$-logic.

There remain several interesting tasks for future research. First, our logic is not yet equipped with a proof system for deriving consequences of specifications. This would also support the proof of refinement steps which is currently achieved by purely semantic reasoning. A proof system for $\mathcal{E}^{\downarrow}$-logic must cover dynamic and hybrid logic parts at the same time, like the proof system in [15], which, however, does not consider data states, and the recent calculus of [5], which extends differential dynamic logic but does not deal with events and reactions to events. Both proof systems could be appropriate candidates for incorporating the features of $\mathcal{E}^{\downarrow}$-logic. Another issue concerns the separation of events into input and output as in I/O-automata [14]. Then also communication compatibility (see [2] for interface automata without data and [3] for interface theories with data) would become relevant when applying a parallel composition constructor.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2013)
2. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Tjoa, A.M., Gruhn, V. (eds.) Proceedings 8th European Software Engineering Conference & 9th ACM SIGSOFT International Symposium Foundations of Software Engineering, pp. 109–120. ACM (2001)
3. Bauer, S.S., Hennicker, R., Wirsing, M.: Interface theories for concurrency and data. Theoret. Comput. Sci. **412**(28), 3101–3121 (2011)
4. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 133–148. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79707-4_11
5. Bohrer, B., Platzer, A.: A hybrid, dynamic logic for hybrid-dynamic information flow. In: Dawar, A., Grädel, E. (eds.) Proceedings of 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 115–124. ACM (2018)
6. Braüner, T.: Hybrid Logic and its Proof-Theory. Applied Logic Series. Springer, Heidelberg (2010). https://doi.org/10.1007/978-94-007-0002-4
7. Farrell, M., Monahan, R., Power, J.F.: An institution for Event-B. In: James, P., Roggenbach, M. (eds.) WADT 2016. LNCS, vol. 10644, pp. 104–119. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72044-9_8
8. Gorrieri, R., Rensink, A.: Action refinement. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 1047–1147. Elsevier, Amsterdam (2000)
9. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
10. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
11. Hennicker, R., Madeira, A., Knapp, A.: A hybrid dynamic logic for event/data-based systems (2019). https://arxiv.org/abs/1902.03074
12. Knapp, A., Mossakowski, T., Roggenbach, M., Glauer, M.: An institution for simple UML state machines. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 3–18. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_1
13. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2003)
14. Lynch, N.A.: Input/output automata: basic, timed, hybrid, probabilistic, dynamic, ... In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 191–192. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45187-7_12
15. Madeira, A., Barbosa, L.S., Hennicker, R., Martins, M.A.: A logic for the stepwise development of reactive systems. Theoret. Comput. Sci. **744**, 78–96 (2018)
16. Object Management Group: Unified Modeling Language 2.5. Standard formal/2015-03-01, OMG (2015)
17. Poizat, P., Royer, J.C.: A formal architectural description language based on symbolic transition systems and modal logic. J. Univ. Comp. Sci. **12**(12), 1741–1782 (2006)

18. Sannella, D., Tarlecki, A.: Toward formal development of programs from algebraic specifications: implementations revisited. Acta Inf. **25**(3), 233–281 (1988)
19. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-17336-3

Model-Driven Development and Model Transformation

Pyro: Generating Domain-Specific Collaborative Online Modeling Environments

Philip Zweihoff(✉), Stefan Naujokat, and Bernhard Steffen

Chair for Programming Systems, TU Dortmund University, Dortmund, Germany
{philip.zweihoff,stefan.naujokat,bernhard.steffen}@tu-dortmund.de

Abstract. We present Pyro, a framework for enabling domain-specific modeling via the internet. Provided with an adequate metamodel specification, Pyro turns your browser into a collaborative, domain-specific, graphical development environment with features reminiscent of desktop IDEs for textual programming languages. The required metamodeling is supported in a high-level, simplicity-driven fashion, and the entire ready-to-run browser-based domain-specific development environment is generated fully automatically. We will illustrate the steps of this development along the realization of a graphical IDE for the Architecture Analysis and Design Language (AADL).

1 Introduction

Domain-specific languages (DSLs) aim at closing the gap between domain knowledge and software development by explicitly supporting the required domain concepts. Graphical domain-specific languages have turned out to be particularly suitable for domain experts without any programming background. The bottleneck in practice is the enormous effort to develop the required domain-specific graphical modeling tools. The Cinco *SCCE Meta Tooling Suite* [26] has been designed to overcome this bottleneck by providing a holistic, simplicity-driven [22] approach for the creation of such domain-specific graphical modeling tools. A key feature of Cinco is that it generates the entire graphical modeling environment (referred to as 'Cinco Products' in the remainder of the paper) from high-level specifications of the defined model structures and functionalities. The (translational) semantics of the specified modeling language is defined in terms of code generation, model transformation, evaluation, and/or interpretation [20]. Cinco Products are Eclipse-based, graphical modeling tools that are realized via a number of Eclipse plug-ins [13]. Thus, setting up a Cinco Product involves some technical aspects that are beyond the competence of typical domain experts, and it becomes even more tedious when one wants to enable a cooperative development.

In this paper, we present *Pyro*, a tool that enables one to generate Cinco Products for collaborative modeling that run in a web browser. Conceptually, Pyro borrows from modern online editors for collaborative work, like Google

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 101–115, 2019.
https://doi.org/10.1007/978-3-030-16722-6_6

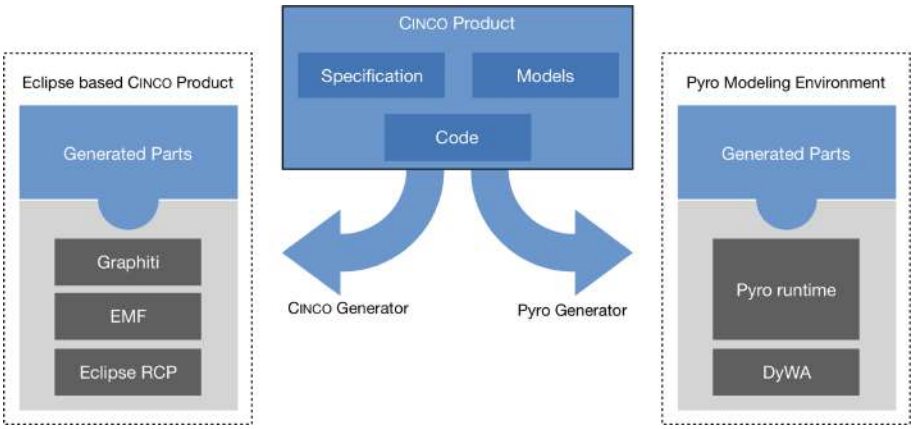


Fig. 1. Cinco generation architecture.

Docs, Microsoft Office 365, or solutions like ShareLaTeX/Overleaf that even free one from maintaining a corresponding build and runtime environment.

Key to the realization of Pyro is that CINCO follows a fully generative approach on the meta level, which allows one to modularly 'retarget' the CINCO Product Generation for the web (cf. Fig. 1). Technically, Pyro web modeling environments utilize *DyWA* [27] (Dynamic Web Application) for data modeling, empowering prototype-driven application development.

In order to achieve this retargeting and to enable collaborative work, Pyro needs to, in particular, compensate for all the required functionality provided by the Eclipse platform, like the EMF framework with GMF or Graphiti for graphical editors. Altogether, this poses the following three key challenges:

- Developing an adequate web solution for the metamodel-based model handling (API, persistence, event system, etc.) that in the Eclipse world is provided by the EMF framework [33] (see *Architecture Backend*, Sect. 3.1).
- Developing a frontend on top of these model structures that feels like a modern integrated development environment with a graphical editor for the models, which in the Eclipse world is provided by the Rich Client Platform (RCP) [24] and the Graphiti editor framework [2] (see *Architecture Frontend*, Sect. 3.2).
- Enabling real-time live collaborative working on models, which is not foreseen in an offline client like Eclipse (see *Collaborative Editing*, Sect. 4).

In the course of this tool paper, Pyro is illustrated along the development of a graphical modeling environment for the *Architecture Analysis and Design Language* (AADL), an SAE standard for modeling the architecture of embedded real-time systems [29]. CINCO was used to develop a graphical AADL modeling tool supporting a subset of AADL's features tailored to be used in teaching [28],

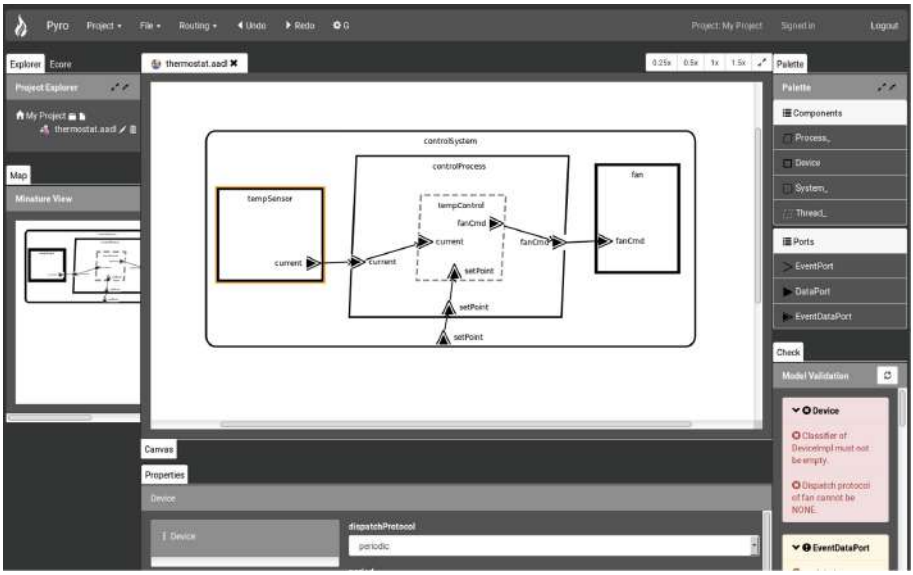


Fig. 2. Pyro web-based modeling environment for the AADL language.

where it replaces the graphical editor of the OSATE tool [8] (AADL's reference implementation). Furthermore, a dedicated code generator was developed to support verification with behavior specified with the BLESS language [17]. Another example for Pyro realizing a DSL for point and click adventures can be found in [21].

Figure 2 shows the web-based graphical AADL editor in Pyro[1]. We will use this editor in the remainder of this paper to illustrate CINCO's and Pyro's core ideas and concepts. The user interface is designed after commonly known concepts from integrated development environments, like Eclipse or IntelliJ. The main area in the center is covered by the *modeling canvas* showing the currently edited model. On the right, there is the *palette* showing the available types of modeling elements. They can be placed onto the canvas just by drag&drop. The attributes of the currently selected element in the editor can be set via the *properties* view at the bottom. The *validation* view (bottom right corner) constantly checks for the syntax and static semantics of the model in the canvas and provides appropriate error or warning messages. Finally, a *project explorer* and a *menu bar* complete the IDE-like appearance.

The remainder of the paper is organized as follows: While Sect. 2 briefly describes the use of CINCO's specification languages to define a sophisticated graphical

[1] The editor is available for experimentation on the Pyro website: https://pyro.scce.info.

modeling language, the generation to a web-based environment and the resulting architecture is explained in Sect. 3. The mechanisms and techniques used to enable simultaneous collaboration are explained in Sect. 4. The paper closes with a summary, related work, and an outlook of the future development in Sect. 5.

2 DSL Development with Cinco

Cinco is a language workbench [11] for the simplicity-driven development of graphical modeling environments that are domain-specific [12], support full code generation [10,15], and easily integrate existing solutions in the form of services [23]. As Cinco is itself a meta-level application of these principles [25], it is specialized to the domain of 'graph-based graphical modeling tools' and fully generates such tools from meta-level descriptions (models) – the key enabling factor for the whole Pyro approach. Primarily relevant in this regard are two Cinco metamodeling languages:[2]

1. The *Meta Graph Language* (MGL) allows for the definition of the abstract syntax of the developed language, i.e., which types of language elements exist and how they can be related. In the context of AADL, this means, for instance, that a *system* model consists of *devices*, *processes* and *threads*, and that all of them have *ports* (of different types) that can be connected with *data/information flow* edges.
2. The *Meta Style Language* (MSL) is used to specify the concrete graphical syntax of those MGL-defined concepts by means of simple hierarchical shapes and their appearance (such as color, line type/width, etc.). As can be seen in Fig. 2, for instance, *devices* are depicted by a black thick line rectangle, while *threads* appear as a grey dashed line parallelogram.

With these meta-level specification files, the Cinco Product Generator (which is part of Cinco) generates plug-ins for the Eclipse Rich Client Platform (RCP) that realize the editor based on the Eclipse Modeling Framework (EMF) and the Graphiti graphical editor framework. Further additions to the editor, which are not covered by these two specification files, can be injected in an aspect-oriented fashion [16]: Cinco provides a so-called mechanism of *hooks* that are triggered on the occurrence of certain events, for instance, when a node is created, moved, or deleted. Hooks are inserted into the MGL file with *annotations* on the model elements defined therein. The effect of a hook can either be modeled in a transformation language [20] or directly be written as Java code using the generated model API. In the context of the AADL editor, e.g., a `postMoveHook` is used to move a port to the nearest border within its container after it has been moved by the user. This results in a very natural 'snapping to the border' effect during modeling.

As Cinco follows a fully generative approach, the very same specification files are utilized by Pyro to generate a web-based modeling editor that runs in

[2] For a more elaborate introduction on how to define a graphical editor with Cinco, as well as other case studies and exemplary modeling languages, please refer to [26].

the browser (cf. Fig. 1). Of course, in this context, the running platform won't be based on Eclipse anymore, but based on common web frameworks like Angular for the frontend and Java EE for the backend. The aspects of a CINCO Product included in a service-oriented fashion via native components written in Java (for instance a code generator or editor-assisting features like the hooks discussed above) can thus directly be run also in the backend of the Pyro editor.

In the following, we will focus on two particularly important aspects of Pyro: After discussing the frontend/backend architecture of the generated Pyro modeling environments in Sect. 3, we will take a deeper look at the communication pattern between the involved components that facilitates synchronous collaborative modeling (cf. Sect. 4).

3 Architecture

In contrast to developing an Eclipse-based modeling environment, for the realization of a web-based solution one nearly has to start from scratch. Eclipse itself is built on a huge amount of plug-ins, developed over the past seventeen years. In particular, the Eclipse Modeling Project provides many frameworks for developing modeling languages based on metamodels and bundling them into a rich IDE. In the context of the web, development of integrated environments has just started, so that only a few best practices, plug-ins, and frameworks are available. This means, even fundamental features often have to be implemented to enable basic functionalities. The main difference between local desktop IDEs and a web-based environment like Pyro is the opportunity to provide distributed access to a centralized instance by multiple users at the same time. This results in new challenges and requirements regarding the synchronization between multiple users and conflict resolution for oppositional modifications.

Thus, the Pyro architecture must be built in a way that adequately substitutes what Eclipse already provides in the desktop application context, but also be prepared for the distributed setting with multiple users – in particular for supporting live collaborative editing on the same models. In this section, the generation of Pyro web-based modeling environments is described in a way that shows how the needed information is collected from CINCO's high-level specification metamodels and where the generated code is placed and distributed in the overall context to build the Pyro architecture.

The previously introduced specification of the AADL modeling language constitutes the source for the tool generation step. After the Pyro generator is triggered, all MGL and MSL files for a CINCO-based modeling tool are collected to gather the required information. At this point, all modeling languages, including their available node and edge types, are visible for the generator.

In the next step, a template of the modeling environment web application is created. The gray parts with dotted borders in Fig. 3 show the static elements independent of the given language specification, whereas the blue parts with solid borders are specifically generated from this specification. The template consists of a *DyWA*-based backend, extended by a specific *Domain Layer*

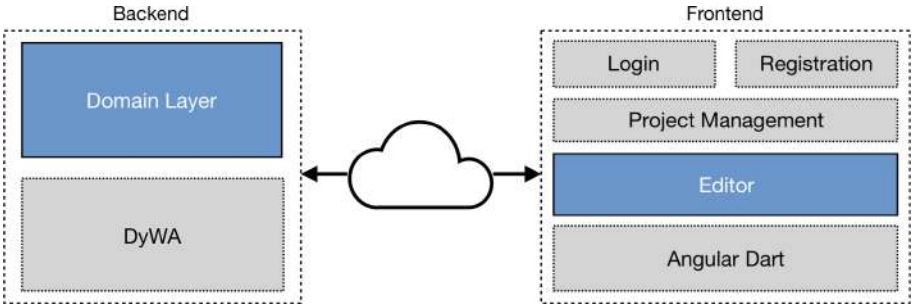


Fig. 3. Overall architecture of the generated web-based modeling environment.

for communication. On the client side, some general parts provide *Registration*, *Login*, and *Project Management*, but the main component is the specific *Editor* generated to handle instances of the graphical modeling language. The underlying single-page web application framework *Angular Dart* [1] is utilized to enable the required features of a rich internet application, like versatile user interaction and asynchronous communication.

Essentially, in the backend, the challenge of providing the metamodel-based model handling (persistence, API, event handling, etc.) is solved, which in the Cinco desktop client world is provided by the EMF framework. The frontend, on the other hand, realizes the rich IDE-like frame application with the graphical editor for the models. In the following, these two parts are explained in more detail to show how the different layers are connected and which parts are generated to establish the entire integrated environment.

3.1 Backend

The backend of a modeling environment generated using Pyro consists of two main layers: One is responsible for the centralized persistence of model instances, the other for receiving and distributing modifications. The lowest level of the web application is the database to store information in a centralized fashion. This layer handles the representation of predefined metamodels for the given domain-specific languages. Pyro modeling environments utilize the *DyWA* as an abstraction layer of a database to store types and objects in a dynamic and loosely coupled fashion [27]. Based on the specified languages' node and edge types, a *Domain Data Plug-in* (see Fig. 4) is generated by Pyro which declares types, associations, attributes, and inheritance. The main reason for using the *DyWA* as model layer is its *Domain Generator*, which generates a specific *DyWA API* providing entities and controllers for the previously given types to handle their instances on a simplified layer above the database. This closely resembles the APIs generated by EMF in the Eclipse world, so that the effort of generating the required *Cinco API* adapters is greatly reduced, which provides functionalities with identical signatures as EMF, so that already

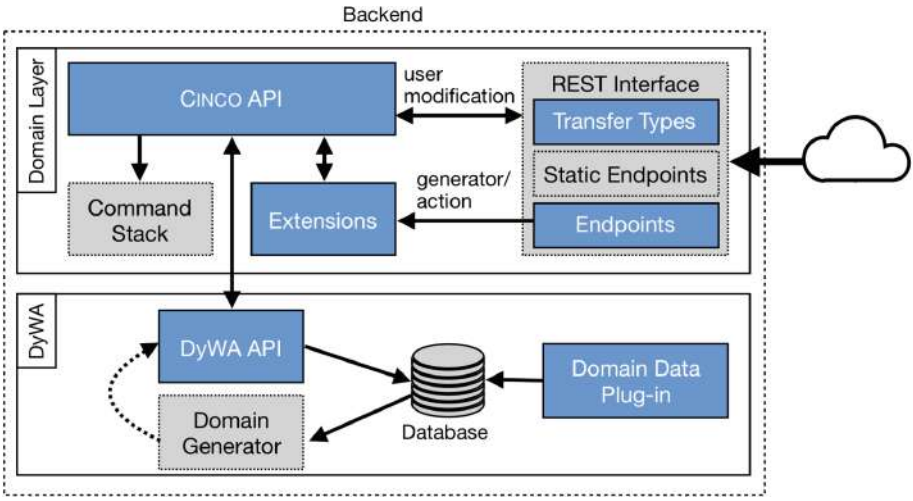


Fig. 4. Backend component architecture and interaction.

existing code can directly be applied (see below). Beyond that, DyWA is prepared for dynamic change of the metamodel, which becomes necessary during modeling language evolution (see [19]).

Since CINCO supports to extend the definition of graphical modeling languages by user-written Java code for hooks, actions, validation checks, and code generators, a holistic reuse mechanism has to be provided in the context of Pyro. To meet this goal, the same CINCO interfaces are rebuilt in the generated web-based modeling environment, providing the same structure and identical signatures. As a result of this, the domain-specific interfaces (see Fig. 4, *CINCO API*) generated by Pyro are compatible to the one CINCO generates for Eclipse and EMF to be used identically by these extensions. In contrast to the desktop-based CINCO Product, a Pyro graph model instance is not persisted in a file on the local system. The Pyro web modeling environment as a distributed system utilizes the DyWA database for storage and centralized access as a server. Thus, the *CINCO API* is internally connected to the corresponding generated *DyWA API* to persist changes in the database, which is hidden from the extensions.

Multi-user collaborative editing with the generated domain-specific modeling languages is one of the main challenges for Pyro. All changes to a centrally held instance of a graph model have to be shared with all participants. For the distribution of the changes performed on a graph model by calling the *CINCO API*) methods, a *Command Stack* is used, to store each individual modification. Since CINCO provides hooks for aspect-oriented extensions, a single action like the movement of a node on the canvas can result in multiple successive commands. As a result, all modifications on a model or any of their elements at runtime are encoded in commands and sequentially stored in the stack. The recorded commands during the *CINCO API* usage are used to synchronize between different

clients looking at the same model as well as the realization of redo and undo functionalities. This synchronization mechanism is described in more detail in Sect. 4.

To use the web modeling environment in a desktop application fashion, an uninterruptible user interaction is necessary. Thus, Pyro utilizes REST-based asynchronous communication for non-blocking data exchange. As a result of this, the outermost component of the generated web application is a *REST Interface*. The interface consists of *Static Endpoints* for project, file, and user management, which are independent from the given modeling languages. These parts are supplemented by generated *Endpoints*, which are based on the CINCO specification and provide methods to create, read, update, and delete (CRUD) a single graph model. In addition to this, the interface contains the central endpoint for commands sent from a client's frontend to the backend. Depending on the used *Extensions*, additional *Endpoints* are generated to fetch and trigger user-written actions or a generator.

3.2 Frontend

To mimic the look and feel of a local desktop modeling environment, the web-based variant generated by Pyro has to provide versatile user interactions. As a result of this, the *Frontend* of the generated web application (see Fig. 5), which realizes the interface for the user, is focused on quick responses and familiar input behavior. To achieve this goal, the frontend part of a web modeling environment is built upon the *Angular Dart* [1] framework, which is used to realize single-page web applications with built-in cross-platform support and comprises an architecture focused on reusable components. In addition to this, it is tailored to asynchronous user interaction and client-side routing, so that it can be used to build rich internet applications, like, for instance, ones resembling integrated development environments (IDEs).

In contrast to a local desktop application, a web application requires additional multi-user focused interfaces. Therefore, the template for the frontend, which is initially created, consists of static user interfaces for *Registration* and *Login* as well as a *Project Management* area to create, edit, and share projects. The specifically generated parts are used by the *Editor*, which comprises domain-specific components. Its user interface is similar to the known Eclipse IDE used by regular CINCO Products (see Fig. 2).

The challenge of preventing delays in the system's response on a user input to enable fluent interaction can be met by avoiding synchronized communication with the backend. The *editor* facilitates this frontend-side computation by two layers used to interact with instances of the graph models. The *Mirror Layer* stores a snapshot of the model present in the database, whereas the *Interaction Layer* is a direct representation of a visible graph which can be modified by the user. This separation enables a delta between the last valid graph, stored in the *Mirror Layer* and the currently visible graph. Thanks to this, generated syntactical validators (e.g., for ensuring lower bounds of given cardinalities) can

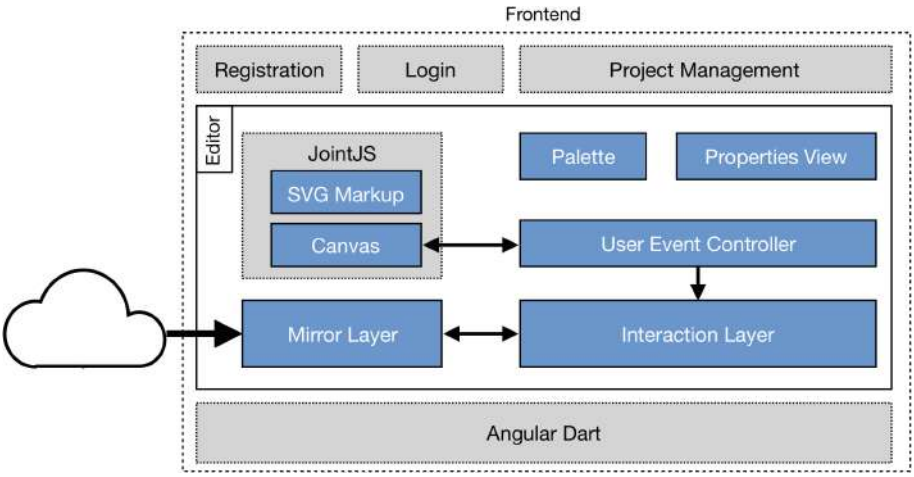


Fig. 5. Front end architecture.

raise errors and the appropriate rollback operation works immediately on the client side without additional communication with the backend.

Pyro specifically aims at supporting users switching from already existing Cinco Products to the web-based modeling environment. Thus, the *Editor*, which is the main part of the frontend, provides multiple components similar to the Eclipse IDE. To not confuse users, functions, behavior and arrangement are recreated. Besides common user interface parts like a project explorer and a menu, specific components for the modeling environment are generated, like the *Canvas*, a *Properties View*, and the *Palette*.

The *Canvas* is based on the *JointJS* framework [9], which in general renders SVGs and adds versatile user interaction for manipulation of nodes and edges via drag&drop functionalities. Using this, it was possible that the web modeling environment running in a browser provides very similar handling to the Eclipse-based desktop application with its Graphiti editor. The exact replication of the node and edge appearance is a central goal of the generated *Canvas*. Ideally, a user cannot distinguish between a Pyro and Cinco visualization of a graph model. This requires the same hierarchical shape structure for the web as in the Graphiti editor, which can be realized by scalable vector graphics (SVGs). The *SVG Markup*, which defines the shapes and styling information of the nodes and edges, is generated based on the concrete syntax specified in the MSL files of Cinco. The *JointJS* framework and *SVG Markup* files are observed by a domain-specific *User Event Controller*, which realizes the listeners and stream handling mechanisms for a single graph model to modify the underlying layers.

Besides the distinct and visible modifications available directly in the *Canvas*, attributes of an edge, node or the graph model (as defined in the MGL metamodel) can be modified using the *Properties View*. It has a generic frame based on a tree view to recursively walk through associated types of the currently

selected element. For every type present in an MGL file, a form for editing the primitive attributes (e.g string, Boolean or integer) is generated. The single fields are tailored to the specified data type of the attribute, to give as much support as possible. Thanks to the two-way data binding of the underlying Angular framework, every change to an attribute is immediately propagated to the underlying layer.

The *Palette* is generated based on the given MGL specifications. It lists all node types available for modeling. In addition to this, the optional annotations of the MGL, e.g. for grouping nodes and dedicated icons for visual support, are considered as well.

4 Collaborative Editing

One of the main features of modeling environments generated by Pyro is the simultaneous editing of graph models by multiple clients at the same time. The continuous synchronization between clients avoids classical revision control repositories for distributed access and instead enables immediate collaboration. To reach the goal of simultaneous synchronization, different aspects have been considered to maintain consistency, scalability and achieve a real-time effect.

In this section, the mechanism used for Pyro web-based modeling environments to communicate is presented and explained. The first part discusses the different challenges of a distributed system with respect to the domain of graphical modeling environments, whereas the second part describes the realization of the command pattern used to exchange modifications on a graph model.

4.1 Simultaneous Synchronization Mechanism

The main communication concept of a generated modeling environment by Pyro as a distributed system is the *optimistic replication strategy* [30]. This concept replicates data and allows the single replicas to diverge, which in the context of Pyro is realized by the separated graph model replicas held in each client. The optimistic replication belongs to the *eventually consistent* consistency model and is furthermore classified as *basically available, soft state and eventually consistent* (BASE) [36]. It benefits from high availability, since it only exchanges updates on given items. In the context of a web-based modeling environment, the updates are based on the modifications a client can do to a node or edge. To enable conflict resolution and maintain consistency regarding commutativity and idempotency, *conflict-free replicated data types* (CRDTs) are represented by commands. CRDT was originally used for text-based synchronization as a simplification of *operational transformation* [34]. It utilizes an additional data structure, based on an identifier of the client, the changed value and the position to create a unique identifier for each changed character of the text. Regarding the graph models handled by Pyro, CRDTs are realized by commands for each type of possible model element modification, which store a unique identifier and the changed properties of the relevant element. In addition to this, the previous

values of the updated properties are stored as well, to enable rollback, undo, and redo functionalities. Thus, Pyro uses operation-based and state-based CRDTs. Thanks to the CRDTs, conflicts of simultaneously editing the same model element at the same time can be detected. In the context of graphical DSLs, conflicts can arise by violating the given static semantics defined in the metamodel. If a conflict is detected, the corresponding command is flagged for rollback and returned to its sender. The client then inverts the modification encoded by the command and applies it to revert the conflicting change.

4.2 Distributed Command Pattern

The distribution of modifications made to a graph model in the Pyro web modeling environment is realized by a *command pattern* [14]. It belongs to the behavioral design pattern, which is used to encapsulate all information needed to perform an update on an object. The commands are sent as HTTP POST requests, combining the graph model and client identifier. An exemplified collaboration of two clients (red and green) modifying the same graph model simultaneously is presented in Fig. 6.

After the initial read from the database, a client only calculates, exchanges and receives commands when a modification is done (see Fig. 6(1)). For every possible change on nodes and edges (e.g., moving a node or bending an edge), a dedicated command encoding the modification is created and sent to the server, extended with a unique identifier of the sender. Thanks to this assignment, all commands can be differentiated (see red commands by client A and green commands by client B in Fig. 6). As an example, the command for the creation of a node consists of the node type, the position and an identifier of the container where it should be instantiated. Other commands, e.g., the move node commands, contain information of the previous as well as the new position, so that they store the delta of the modification.

The *Serializer* (see Fig. 6(2)) is used to parse the received payload and assign the commands to the associated *Command Applier*. Thanks to additional reflective *type* annotations, the received payload can be parsed to recreate the correct command type. The assignment depends on the given graph model type the command belongs to.

The *Command Applier* (see Fig. 6(3)) is the main component of the web server, since it receives, validates and executes the commands. Every modification encoded by a command is initially validated against the syntactical constraints defined by the graph model type. In the case of a constraint violation, the command is inverted based on the given delta, and returned to undo the invalid operation sent from a client. After a successful validation, the modification encoded by the command is applied to the generated domain-specific API, which also triggers the annotated hooks and finally modify the node or edge instances in the central database. Modifications performed on the API itself (e.g., performed by a hook implementation) are again internally encoded as commands for further distribution to other clients. The updates resulting from the hook execution inside the API are combined with the initial command to be

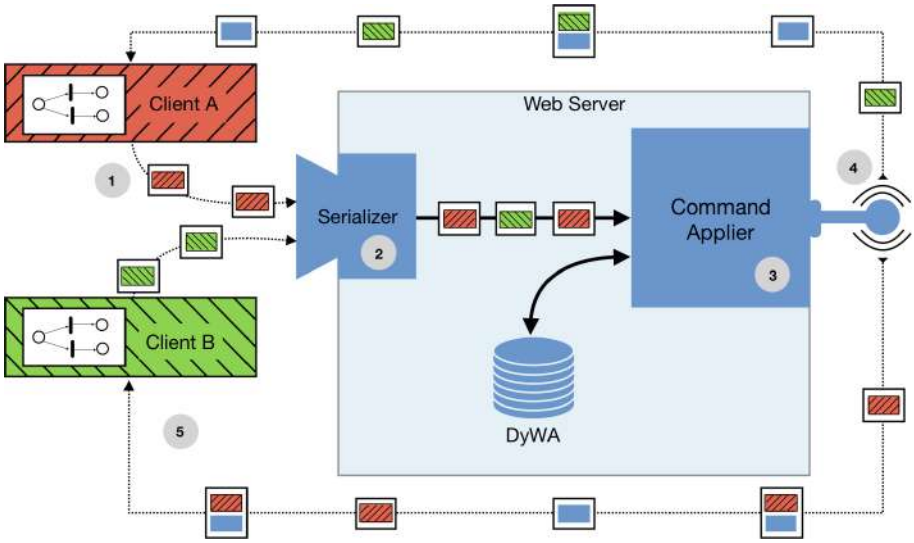


Fig. 6. Concept of the distributed command pattern. (Color figure online)

interpreted as a single transaction shown by the packages of Fig. 6. To ensure the consistency between the sender of a command and the other clients, the initiator is also informed about internally arisen modification based on hook execution. All commands, collected during the execution of the initial modification, are broadcast to other listening clients (see Fig. 6(4)). This mechanism uses bidirectional ongoing connections, so that clients can request to listen on changes made to their currently open graph model.

The commands received by a client (see Fig. 6(5)) are parsed and inspected, to ensure that commands initiated by the client itself are neglected. New changes from other clients are applied to all layers and displayed on the canvas. In addition to this, the client is notified about received changes. Updates caused as a result of self-sent commands (e.g., a modification performed during a hook execution), are only partially applied to guarantee that nodes and edges will not be modified twice.

The command pattern applied to the generated modeling environments is tailored to enable real-time collaborative editing. The main design decisions are focused on scalability and high availability by BASE and CRDT. The operational approach realized with this command pattern is more suitable than a textual language protocol like the *Language Server Protocol* (LSP) [3]. The main difference between the command pattern and the LSP is the way of distributing modifications on the model. In contrast to the presented communication protocol of Pyro, the LSP uses changed regions of a text document for propagation. The intention of the modification has to be evaluated afterwards, whereas in graphical DSLs the commands are used for a direct representation of the occurred change.

5 Conclusion and Perspectives

We have presented Pyro, a framework for enabling domain-specific modeling via the internet. Provided with an adequate metamodel specification, Pyro turns a browser into a collaborative, domain-specific, graphical development environment with features reminiscent of desktop IDEs for programming textual languages. The required metamodeling is supported in a high-level, simplicity-driven fashion: The MGL describes the available node types, edge types, and syntactical constraints, whereas the MSL defines the visual appearance of the modeling artifacts defined in the MGL. Based on these specifications, the entire ready-to-run browser-based domain-specific development environment is generated fully automatically, as has been illustrated along the construction of a graphical development environment for the Architecture Analysis and Design Language (AADL).

The field of web-based development environments is still quite young, so that not many related solutions exist yet. There are the aforementioned collaborative online text editors like Google Docs, Microsoft Office 365 and ShareLaTeX/Overleaf, but in the area of DSLs and modeling, so far we only encountered WebGME [5], an (early stage) online adaption of Vanderbilt University's Generic Modeling Environment [18] and Theia [4], a cross-platform web and desktop IDE for textual DSLs. In addition, itemis (the German company who significantly contributed to the well-known Xtext [6] DSL framework) is currently working on a platform called 'Convecton', which aims at bringing modeling with and execution of domain-specific languages online into the cloud [35]. However, none of these solutions provide a Pyro-like, graphical, collaborative modeling support.

Pyro is still in an early stage of development, and there is a lot of room for improvement, like further enhancing and easing the graphical modeling features, or improving the performance of collaborative modeling by taking advantage of peer-to-peer communication. Pyro is envisioned to enable cross-competence collaboration on a single project in a domain/purpose-specific fashion according to the Language-Driven Engineering (LDE) paradigm [31]. LDE aims at allowing the different stakeholders to formulate their intents in they way they are used to, i.e., in their domain language, and restricted in a fashion that the efforts of the other involved stakeholders are maintained, or as we say, constitute Archimedean points [32] of the considered domain-specific language. Currently, we are starting to explore the impact of the Pyro technology on a larger scale for DIME [7], our framework for developing Web applications.

References

1. About AngularDart. https://webdev.dartlang.org/angular. Accessed 13 Feb 2019
2. Graphiti - A Graphical Tooling Infrastructure. http://www.eclipse.org/graphiti/. Accessed 13 Feb 2019
3. Official page for Language Server Protocol. https://microsoft.github.io/language-server-protocol/. Accessed 12 Feb 2019
4. Theia - Cloud and Desktop IDE. https://www.theia-ide.org. Accessed 12 Feb 2019

5. WebGME. https://webgme.org/. Accessed 13 Feb 2019
6. Xtext - Language Engineering Made Easy! http://www.eclipse.org/Xtext/. Accessed 13 Feb 2019
7. Boßelmann, S., et al.: DIME: a programming-less modeling environment for web applications. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 809–832. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_60
8. Carnegie Mellon University: Welcome to OSATE. http://osate.org/. Accessed 13 Feb 2019
9. client IO: Joint API. http://www.jointjs.com/api. Accessed 13 Feb 2019
10. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York (2000)
11. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? June 2005. http://martinfowler.com/articles/languageWorkbench.html. Accessed 13 Feb 2019
12. Fowler, M., Parsons, R.: Domain-Specific Languages. Addison-Wesley/ACM Press (2011). http://books.google.de/books?id=ri1muolw_YwC
13. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Boston (2008)
14. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002). ACM SIGPLAN Notices, vol. 37, pp. 161–173. ACM (2002)
15. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley/IEEE Computer Society Press, Hoboken (2008)
16. Kiczales, G., et al.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
17. Larson, B.R., Chalin, P., Hatcliff, J.: BLESS: formal specification and verification of behaviors for embedded systems with software. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 276–290. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_19
18. Ledeczi, A., et al.: The generic modeling environment. In: Workshop on Intelligent Signal Processing (WISP 2001) (2001)
19. Lybecait, M., Kopetzki, D., Naujokat, S., Steffen, B.: Towards Language-to-Language Transformation (2019, to appear)
20. Lybecait, M., Kopetzki, D., Steffen, B.: Design for 'X' through model transformation. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 381–398. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_23
21. Lybecait, M., Kopetzki, D., Zweihoff, P., Fuhge, A., Naujokat, S., Steffen, B.: A tutorial introduction to graphical modeling and metamodeling with CINCO. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 519–538. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_31
22. Margaria, T., Steffen, B.: Simplicity as a driver for agile innovation. Computer **43**(6), 90–92 (2010)
23. Margaria, T., Steffen, B.: Service-orientation: conquering complexity with XMDD. In: Hinchey, M., Coyle, L. (eds.) Conquering Complexity, pp. 217–236. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2297-5_10
24. McAffer, J., Lemieux, J.M., Aniszczyk, C.: Eclipse Rich Client Platform, 2nd edn. Addison-Wesley Professional (2010)

25. Naujokat, S.: Heavy Meta. Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools. Dissertation, TU Dortmund, Dortmund, Germany, August 2017. http://hdl.handle.net/2003/36060
26. Naujokat, S., Lybecait, M., Kopetzki, D., Steffen, B.: CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. Softw. Tools Technol. Transf. **20**(3), 327–354 (2017)
27. Neubauer, J., Frohme, M., Steffen, B., Margaria, T.: Prototype-driven development of web applications with DyWA. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8802, pp. 56–72. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45234-9_5
28. Robby, Hatcliff, J., Belt, J.: Model-based development for high-assurance embedded systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 539–545. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_32
29. SAE International: Architecture Analysis & Design Language (AADL), January 2017. https://www.sae.org/standards/content/as5506c/. SAE Standard AS5506C
30. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. (CSUR) **37**(1), 42–81 (2005)
31. Steffen, B., Gossen, F., Naujokat, S., Margaria, T.: Language-driven engineering: from general-purpose to purpose-specific languages. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science: State of the Art and Perspectives. LNCS, vol. 10000. Springer, Heidelberg (2019, to appear)
32. Steffen, B., Naujokat, S.: Archimedean points: the essence for mastering change. LNCS Trans. Found. Mastering Change (FoMaC) **1**(1), 22–46 (2016)
33. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
34. Sun, C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms, and achievements. In: Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work (CSCW 1998), pp. 59–68. ACM (1998)
35. Voelter, M.: Convecton Presentation at LangDev Meetup at CWI 8–9 March 2018. https://github.com/cwi-swat/langdev/blob/gh-pages/slides/Convecton@LangDev.pdf. Accessed 13 Feb 2019
36. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)

Efficient Model Synchronization by Automatically Constructed Repair Processes

Lars Fritsche[1(✉)], Jens Kosiol[2], Andy Schürr[1], and Gabriele Taentzer[2]

[1] TU Darmstadt, Darmstadt, Germany
{lars.fritsche,andy.schuerr}@es.tu-darmstadt.de
[2] Philipps-Universität Marburg, Marburg, Germany
{kosiolje,taentzer}@mathematik.uni-marburg.de

Abstract. Model synchronization, i.e., the task of restoring consistency between two interrelated models after a model change, is a challenging task. Triple Graph Grammars (TGGs) specify model consistency by means of rules. They can be used to automatically derive specifications of edit operations for single models and repair rules that propagate model changes to related models. model (re-)synchronization activities more effectively, a construction mechanism for *short-cut* rules has been recently developed. They describe consistency-preserving complex edit operations across model boundaries. We show that edit and repair rules can be derived from *short-cut* rules. As proof of concept, we implemented the construction and application of *short-cut* edit and repair rules in eMoflon. Our evaluation shows that *short-cut*-rule-based repair processes have considerably decreased data loss and improved runtime compared to former model synchronization processes in eMoflon.

Keywords: Model synchronization · Triple Graph Grammars · Short-cut rule

1 Introduction

Model-driven engineering has become an important technique to cope with the increasing complexity of modern software systems. In the field of Concurrent Engineering [7], for example, products are no longer realized in series but allow parallel tasks. Each of these tasks has its view onto the product and, as a view evolves, it may become inconsistent with the other ones. Keeping views synchronized by checking and preserving their consistency can be a challenging problem which is not only subject to ongoing research but also of practical interest for industrial applications such as stated above.

Triple Graph Grammars (TGGs) [24] are a declarative, rule-based bidirectional transformation approach that aims to synchronize models stemming from different views (usually called *domains* in the TGG literature). Their purpose

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 116–133, 2019.
https://doi.org/10.1007/978-3-030-16722-6_7

is to define a consistency relationship between pairs of models in a rule-based manner by defining traces between their elements. Given a finite set of rules that define how both models co-evolve, a TGG can be automatically *operationalized* into *source* and *forward rules*. The source rules of an operationalized TGG can be used to build up models of one domain while forward rules translate them to models of the other domain, thereby establishing traces between their elements. From a synchronization point of view, source rules specify edit operations to change one model while forward rules specify repair operations to synchronize model changes with one another [16,19,24]. Even though both, the translation and the synchronization process, are formally defined and sound, there are in fact several practical issues that arise for model synchronization from (potentially transitive) dependencies between rule applications: To synchronize changed models, popular TGG approaches do not always fix inconsistencies locally but revert all dependent rule applications and start a retranslation process. However, this kind of synchronization often deletes and recreates a lot of model elements to reestablish model consistency, potentially losing information that is local to just one model and wasting processing time. Existing solutions for this problem are rather ad hoc and come without any guarantee to reestablish the consistency of modified models [12,14].

As a new solution to this synchronization problem, we derive *repair rules* from *short-cut* rules [8] that we recently introduced to handle complex consistency-preserving model updates more effectively and efficiently. The construction of *short-cut* rules is a kind of sequential rule composition that allows to replace a rule application with another one while preserving involved model elements (instead of deleting and re-creating them). We used *short-cut* rules to describe model changes exchanging one edit step by another one. Since in this paper we want to use *short-cut* rules for model synchronization as well, they have to be operationalized into *source* and *forward* rules.

Our formal contributions (in Sect. 4) are two-fold: As *short-cut* rules may be non-monotonic, i.e., may be deleting, we formalize the operationalization of non-monotonic TGG rules which decomposes short-cut rules into (semantically equivalent sequences of) source (edit) and forward (repair) rules. Moreover, we obtain sufficient conditions under which an application of a *short-cut* rule preserves the consistency of related pairs of models. This was left to future work in [8]. Together, this constitutes the correctness of our approach using operationalized *short-cut* rules for model synchronization.

Practically, we implement our synchronization approach in eMoflon [21], a state-of-the-art bidirectional graph transformation tool, and evaluate it (Sect. 5). The results show that the construction of *short-cut* repair rules enables us to react to model changes in a less invasive way by preserving information and increasing the performance. We thus contribute to a more comprehensive research trend in the bx-community towards *Least Change* synchronization [5]. Before presenting these results in detail, we illustrate our approach using an example in (Sect. 2) and recall some preliminaries in (Sect. 3). Finally, we discuss related work in (Sect. 6) and conclude with pointers to future work in (Sect. 7). A technical

report that includes additional preliminaries, all proofs, and the rule set used for our evaluation (including more complex examples) is available online [9].

2 Introductory Example

We motivate the use of *short-cut* repair processes by synchronizing a Java AST (abstract syntax tree) model and a custom documentation model. For model synchronization, we consider a Java AST model as *source* model and its documentation model as *target* model, i.e., changes in a Java AST model have to be transferred to its documentation model. There are correspondence links in between such that both models become correlated.

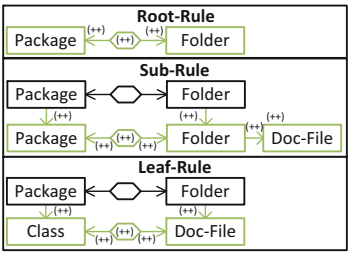


Fig. 1. Example: TGG rules (Color figure online)

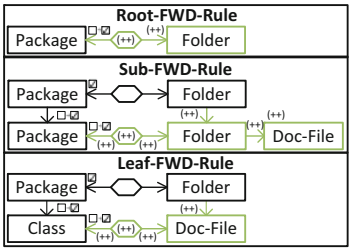


Fig. 2. Example: TGG forward rules

TGG rules. Figure 1 shows the rule set of our running example consisting of three TGG rules: *Root-Rule* creates a root *Package* together with a root *Folder* and a correspondence link in between. This rule has an empty precondition and only creates elements which are depicted in green and with the annotation (++). *Sub-Rule* creates a *Package* and *Folder* hierarchy given that an already correlated *Package* and *Folder* pair exists. Finally, *Leaf-Rule* creates a *Class* and a *Doc-File* under the same precondition as *Sub-Rule*.

These rules can be used to generate consistent triple graphs in a synchronized way consisting of source, correspondence, and target graph. A more general scenario of model synchronization is, however, to restore the consistency of a triple graph that has been altered on just one side. For this purpose, each TGG rule has to be operationalized to two kinds of rules: *source* rules enable changes of source models which is followed by translating this model to the target domain with *forward* rules. As *source* rules for single models are just projections of TGG rules to one domain, we do not show them explicitly.

Forward translation rules. Figure 2 depicts the *forward* rules. Using these rules, we can translate the Java AST model depicted on the source side of the triple graph in Fig. 3(a) to a documentation model such that the result is the complete graph in Fig. 3(a). To obtain this result we apply *Root-FWD-Rule* at the root

Package, *Sub-FWD-Rule* at *Packages* `p` and `subP`, and finally *Leaf-FWD-Rule* at *Class* `c`. To guide the translation process, context elements that have already been translated are annotated with ☑ in *forward* rules. A formerly created source element gets the marking □ → ☑ to indicate that applying the rule will mark this element as translated; a formalization of this marking is given in [20]. Note that *Root-FWD-Rule* can always be applied when *Sub-FWD-Rule* is applicable which can lead to untranslated edges. For simplicity, we assume that the correct rule is applied which in praxis can be achieved through negative application conditions [15].

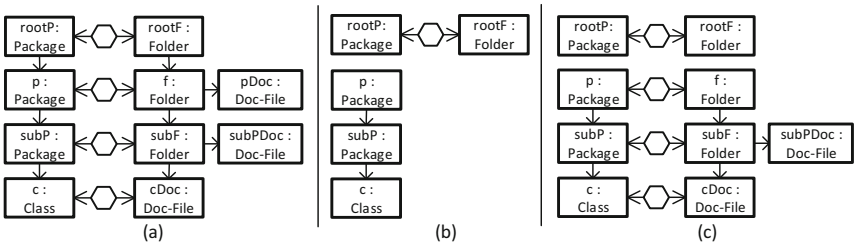


Fig. 3. Exemplary synchronization scenario

Model synchronization. Given the triple graph in Fig. 3(a), a user might want to change a sub *Package* such as `p` to be a root *Package*, e.g., as could be the case when the project is split up into multiple projects. Since `p` was created and translated as a sub *Package* rather than a root element, this change introduces an inconsistency. To resolve this issue, one approach is to revert the translation of `p` into `f` and re-translate `p` with an appropriate translation rule such as *Root-FWD-Rule*. Reverting the former translation step may lead to further inconsistencies as we remove elements that were needed as context elements by other rule applications. The result is a reversion of all translation steps except for the first one which translated the original root element. The result is shown in Fig. 3(b). Now, we can re-translate the unmarked elements yielding the result graph in (c). This example shows that this synchronization approach may delete and re-create a lot of similar structures which appears to be inefficient. Second, it may lose information that exists on the target side only, e.g., a use case may be assigned to a document which does not have a representation in the corresponding Java project.

Model synchronization with short-cut repair. In [8] we introduced short-cut rules as a kind of rule composition mechanism that allows to replace a rule application by another one while preserving elements (instead of deleting and re-creating them). In our example, *Root-Rule* and *Sub-Rule* overlap in elements as the first rule can be completely embedded into the latter one. Figure 4 depicts two possible short-cut rules based on *Root-Rule* and *Sub-Rule*. While the upper short-cut

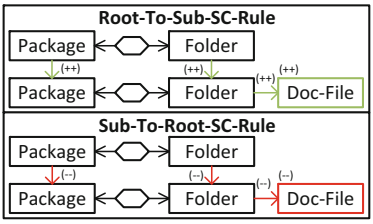


Fig. 4. Short-cut rules (Color figure online)

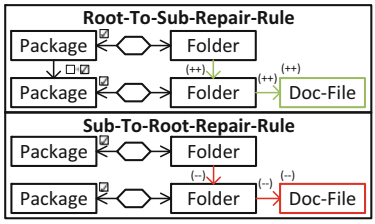


Fig. 5. Repair rules

rule replaces *Root-Rule* with *Sub-Rule*, the lower short-cut rule replaces *Sub-Rule* with *Root-Rule*. Both short-cut rules preserve the model elements on both sides and solely create elements that do not yet exist (++), or delete those depicted in red and annotated with (−−). They are constructed by overlapping both original rules such that each created element that can be mapped to the other rule becomes context and as such, is not touched. When a created element cannot be mapped because it only appears in the replacing rule, it is created. Consequently, an element is deleted if the created element only appears in the replaced rule. Finally, context elements occurring in both rules appear also in the short-cut rule while overlapped context elements appear only once. Using *Sub-To-Root-SC-Rule* enables the user to transform the triple graph in Fig. 3(a) directly to the one in (c).

Yet, these rules can still not cope with the change of a single model since short-cut rules transform both models at once as TGG rules usually do. Hence, in order to be able to handle the deleted edge between `rootP` and `p`, we have to forward operationalize short-cut rules, thereby obtaining *short-cut repair* rules. Figure 5 depicts the resulting *short-cut repair* rules derived from *short-cut* rules in Fig. 4. A non-monotonic TGG-rule is forward operationalized by removing deleted elements from the rule's source graphs as they should not be present after a source rule application. *Short-cut repair* rules allow to propagate source graph changes directly to target graphs to restore consistency. In our example, after having transformed Package `p` into a root element, the rule of choice is *Sub-To-Root-Repair-Rule* which transforms Folder `f` in Fig. 3(a) into a root element and deletes the superfluous *Doc-File*. The result is again the consistent triple graph depicted in Fig. 3(c). This repair allows to skip the costly reversion process with the intermediate result in Fig. 3(b). Note that applying *Sub-To-Root-Repair-Rule* at arbitrary matches may have undesired consequences: One could, e.g., delete the edge between two *Folders* even if the matched *Packages* are still connected. Our Theorem 8 characterizes matches where such violations of the language of the grammar cannot happen. In our implementation, we exploit an incremental pattern matcher to identify valid matches. Using suitable *negative application conditions* [6] would be an alternative approach.

3 Preliminaries

To understand our formal contributions, we assume familiarity with the basics of double-pushout rewriting in graph transformation and, more generally in adhesive categories [6,18] as well as the definition of TGGs and in particular, their operationalizations [24]. Here, we recall non-basic preliminaries for our work which are the construction of short-cut rules, the notion of sequential independence, and a (simple) categorical definition of partial maps.

In [8], we introduced short-cut rules as a new way of sequential composition for monotonic rules. Given an inverse rule of a monotonic rule (i.e., a rule that only deletes) and a monotonic rule, a short-cut rule combines their respective actions into a single rule. Its construction allows to identify elements that are deleted by the first rule as re-created by the second one. These elements are preserved in the resulting short-cut rule. A *common kernel*, i.e., a common subrule of both, serves to identify how the two rules overlap and which elements are preserved instead of being deleted and re-created. We recall their construction since our construction of repair rules is based on it. Examples are depicted in Fig. 4.

Definition 1 (Short-cut rule). *In an adhesive category* $\boldsymbol{C}$*, given two monotonic rules* $r_i : L_i \hookrightarrow R_i$*,* $i = 1, 2$*, and a common kernel rule* $k : L_\cap \hookrightarrow R_\cap$ *for them, the* Short-cut rule $r_1^{-1} \ltimes_k r_2 := (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ *is computed by executing the following steps depicted in Figs. 6 and 7:*

1. *The union* $L_\cup$ *of* L_1 *and* L_2 *along* $L_\cap$ *is computed as pushout* (2)*.*
2. *The LHS* L *of the short-cut rule* $r_1^{-1} \ltimes_k r_2$ *is computed as pushout* (3*a*)*.*
3. *The RHS* R *of the short-cut rule* $r_1^{-1} \ltimes_k r_2$ *is computed as pushout* (3*b*)*.*
4. *The interface* K *of the short-cut rule* $r_1^{-1} \ltimes_k r_2$ *is computed as pushout* (4)*.*
5. *Morphisms* $l : K \to L$ *and* $r : K \to R$ *are obtained by the universal property of* K*.*

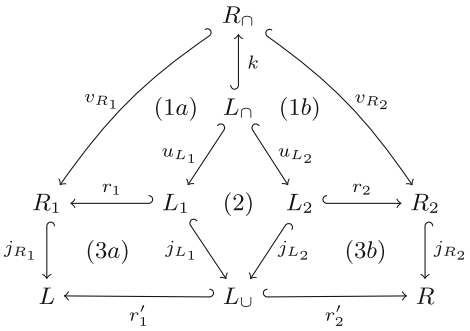


Fig. 6. Construction of LHS and RHS of short-cut rule $r_1^{-1} \ltimes_k r_2$

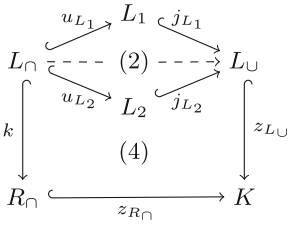
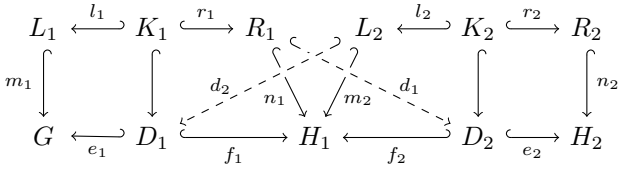


Fig. 7. Construction of interface K of $r_1^{-1} \ltimes_k r_2$

Sequential independence of two rule applications intuitively means that none of these applications enables the other one. This implies that the order of their application may be switched. The definition of sequential independence can be extended to a sequence of rule applications longer than 2. In Theorem 8, we will use this to identify language-preserving applications of short-cut rules.

Definition 2 (Sequential independence). *Given two rules* $p_i = (L_i \overset{l_i}{\hookleftarrow} K_i \overset{r_i}{\hookrightarrow} R_i)$ *with* $i = 1, 2$*, two direct transformations* $G \Rightarrow_{p_1,m_1} H_1$ *and* $H_1 \Rightarrow_{p_2,m_2} H_2$ *via the rules* r_1 *and* r_2 *are* sequentially independent *if there exist two morphisms* $d_1 : R_1 \to D_2$ *and* $d_2 : L_2 \to D_1$ *as depicted below such that* $n_1 = f_2 \circ d_1$ *and* $m_2 = f_1 \circ d_2$*.*



Given rules $p = (L \hookleftarrow K \hookrightarrow R)$ *and* $p_i = (L_i \hookleftarrow K_i \hookrightarrow R_i)$ *with* $1 \leq i \leq t$*, a transformation* $G_t \Rightarrow_{p,m} H$ *is* sequentially independent from a sequence of transformations $G_0 \Rightarrow_{p_1,m_1} G_1 \Rightarrow_{p_2,m_2} \cdots \Rightarrow_{p_t,m_t} G_t, t \geq 2$ *if first,* $G_t \Rightarrow_{p,m} H$ *and* $G_{t-1} \Rightarrow_{p_t,m_t} G_t$ *are sequentially independent and then, the arising transformations* $G_{t-1} \Rightarrow_{p,e_t \circ d_2^t} G'_t$ *and* $G_{t-2} \Rightarrow_{p_{t-1},m_{t-1}} G_{t-1}$ *are sequentially independent and so forth back to the transformations* $G_0 \Rightarrow_{p_1,m_1} G_1$ *and* $G_1 \Rightarrow_{p,e_2 \circ d_2^2} G'_2$ *(where* $e_i : D_i \hookrightarrow G_{i-1}$ *is given by the transformation and* $d_2^i : L \hookrightarrow D_i$ *exists by sequential independence as in the figure above).*

To formalize the application of non-monotonic TGG rules, we need to consider triple graphs with partial morphisms from correspondence to source (or target) graphs. For expressing such triple graphs categorically, we recall a simple definition of partial morphisms [23] to be used in Sect. 4.1. An elaborated theory of triple graphs with partial morphisms is out of scope of this paper.

Definition 3 (Partial morphism. Commuting square with partial morphisms). *A* partial morphism *a from an object A to an object B is a(n equivalence class of) span(s)* $A \overset{\iota_A}{\hookleftarrow} A' \overset{a}{\to} B$ *where* ι_A *is a monomorphism (denoted by* $\hookrightarrow$*). A partial morphism is denoted as* $a : A \dashrightarrow B$*;* A' *is called the* domain *of a. A diagram with two partial morphisms a and c as depicted as square* (1) *in Fig. 8 is said to be* commuting *if there exists a (necessarily unique) morphism* $x : A' \to C'$ *such that both arising squares* (2) *and* (3) *in Fig. 9 commute.*

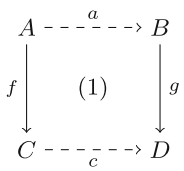


Fig. 8. Square of partial morphisms

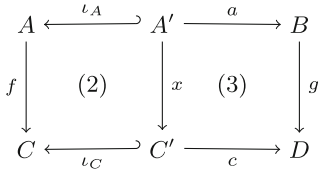


Fig. 9. Commuting square of partial morphisms

4 Constructing Language-Preserving Repair Rules

The general idea of this paper is to use *short-cut repair* rules allowing an optimized model synchronization process based on TGGs. To this end, we operationalize short-cut rules being constructed from the rules of a given TGG. Since those rules are not necessarily monotonic, we generalize the well-known operationalization of TGG rules to the non-monotonic case and show that the basic property is still valid: An application of a source rule followed by an application of the corresponding forward rule is equivalent to applying the original rule instead. This is the content of Sect. 4.1. Constructing *shortspscut* rules in [8], we identified the following problem: Applying a short-cut rule derived from rules of a given grammar might lead to an instance that is not part of the language defined by that grammar. Therefore, in Sect. 4.2, we provide sufficient conditions for applications of short-cut rules leading to instances of the grammar-defined language only. Combining both results ensures the correctness of our approach, i.e., a *shortspscut* repair rule actually propagates a model change from the source to the target model if it is correctly matched.

4.1 Operationalization of Generalized TGG Rules

Since the operationalization of TGG rules has been introduced for monotonic rules only, we extend the theory to general triple rules and, moreover, allow for partial morphisms from correspondence to source and target graph in triple graphs. We split a rule on triple graphs into a *source rule* that only affects the source part and a *forward rule* that affects correspondence and target part.

Definition 4 (TGG rule). *Let the category of triple graphs and graph morphisms be given. A triple rule p is a span of triple graph morphisms*

$$p = ((L_S \xleftarrow{\sigma_L} L_C \xrightarrow{\tau_L} L_T) \xleftarrow{(l_S,l_C,l_T)} (K_S \xleftarrow{\sigma_K} K_C \xrightarrow{\tau_K} K_T) \xhookrightarrow{(r_S,r_C,r_T)} (R_S \xleftarrow{\sigma_R} R_C \xrightarrow{\tau_R} R_T))$$

which, wherever possible, are abbreviated by

$$p = (L_{SCT} \xleftarrow{(l_S,l_C,l_T)} K_{SCT} \xhookrightarrow{(r_S,r_C,r_T)} R_{SCT}).$$

Rules p_S and p_F are called source rule *and* forward rule *of p.*

$$p_S = ((L_S \leftarrow \emptyset \rightarrow \emptyset) \xleftarrow{(l_S,id_\emptyset,id_\emptyset)} (K_S \leftarrow \emptyset \rightarrow \emptyset) \xhookrightarrow{(r_S,id_\emptyset,id_\emptyset)} (R_S \leftarrow \emptyset \rightarrow \emptyset)),$$

$$p_F = (R_S L_{CT} \xleftarrow{(id_{R_S}, l_C, l_T)} R_S K_{CT} \xrightarrow{(id_{R_S}, r_C, r_T)} R_{SCT})$$

with $\emptyset$ *being the empty graph. In* $R_S L_{CT} = (R_S \dashleftarrow L_C \overset{\tau_L}{\hookrightarrow} L_T)$*, the morphism from* L_C *to* R_S *may be partial and is defined by the span* $(L_C \overset{l_C}{\hookleftarrow} K_C \overset{r_S \circ \sigma_K}{\hookrightarrow} R_S)$ *with* $\sigma_K : K_C \hookrightarrow R_C$*.* Target *and* backward rules p_T *and* p_B *are defined symmetrically in the other direction.*

Given a TGG, a short-cut repair rule *is a forward rule* p_F *of a short-cut rule* $p = r_1^{-1} \ltimes_k r_2$ *where* r_1, r_2 *are (monotonic) rules of the TGG, i.e., a repair rule is an operationalized short-cut rule.*

The above definition is motivated by our application scenario, i.e., the case where a user edits the source (or target) model independently of the other parts. The partial morphism in the forward rule reflects that a model change may introduce a situation where the result is no longer a triple graph. A deleted source element may have a preimage in the correspondence graph that is not deleted as well. In the example *short-cut* rules in Fig. 4, this problem does not occur since edges are deleted only. But in general, this definition of p_S has the disadvantage that often, p_S is not applicable to any triple graph since the result would not be one.

In practical applications, however, the source rule specifies a user edit action that is performed on the source part only, ignoring correspondence and target graphs. The fact that the result is not a triple graph any longer is not a technical problem. A missing source element that should be referenced by a correspondence element gives information about a location that needs some repair. Therefore, we define the application of a source rule such that the resulting triple graph is allowed to be partial. Furthermore, forward rules may be applied to partial triple graphs allowing for dangling correspondence relations.

Definition 5 (Constructing an operationalized rule application). *Let a triple graph rule* $p = (L_{SCT} \xleftarrow{(l_S, l_C, l_T)} K_{SCT} \xrightarrow{(r_S, r_C, r_T)} R_{SCT})$ *with source rule* p_S *and forward rule* p_F *be given. An operationalized rule application* $G \Rightarrow_{p_S, m_S} G' \Rightarrow_{p_F, m_F} H$ *is constructed as follows:*

1. *The rule* $p_S^{\mathrm{pr}} = L_S \overset{l_S}{\hookleftarrow} K_S \overset{r_S}{\hookrightarrow} R_S$ *is the* projection *of* p_S *to its source part.*
2. *Given a match* m_S^{pr} *for* p_S^{pr}*, construct the transformation* $t_S^{\mathrm{pr}} : G_S \Rightarrow_{p_S^{\mathrm{pr}}, m_S^{\mathrm{pr}}} H_S$*, called* source application *and inducing the span* $G_S \overset{f_S}{\hookleftarrow} D_S \overset{g_S}{\hookrightarrow} H_S$*.*
3. *The transformation* t_S^{pr} *can be extended to the transformation* $t_S : G = (G_S \overset{\sigma_G}{\hookleftarrow} G_C \overset{\tau_G}{\hookrightarrow} G_T) \Rightarrow_{p_S, m_S} G' = (H_S \dashleftarrow G_C \overset{\tau_G}{\hookrightarrow} G_T)$ *via* p_S *at match* m_S*. The partial morphism* $G_C \dashrightarrow H_S$ *is given as the span* $G_C \hookleftarrow G'_C \to H_S$ *that arises as pullback of the co-span* $G_C \to G_S \hookleftarrow D_S$ *as depicted in Fig. 10, i.e., as morphism* $g_S \circ p_D : G_C \dashrightarrow H_S$ *with domain* G'_C*.*
4. *Given co-match* $n_S : R_S \hookrightarrow H_S$ *and matches* $m_X : L_X \hookrightarrow G_X$ *with* $X \in \{C, T\}$ *such that both arising squares are commuting, i.e.,* $m_F = (n_S, m_C, m_T)$ *is a morphism of partial triple graphs, construct transformation* $t_F : G' \Rightarrow_{p_F, m_F} H = (H_S \overset{\sigma_H}{\hookleftarrow} H_C \overset{\tau_H}{\hookrightarrow} H_T)$*, called* forward application*, using transformations* $G_X \Rightarrow_{p_X, m_X} H_X$ *for* $X \in \{C, T\}$ *if they exist*

and if there are morphisms $\sigma'_D : D_C \to H_S$ *and* $\tau_D : D_C \to D_T$ *such that* $H_S D_C D_T \hookrightarrow H_S G_C G_T$ *and* $R_S K_C K_T \hookrightarrow H_S D_C D_T$ *are triple morphisms.*

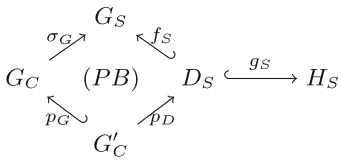


Fig. 10. Retrieval of partial morphism $G_C \dashrightarrow H_S$

In the setting of this paper, it is enough to allow for partial morphisms only in the input graph and not in the output graph of a forward rule application. Intuitively this means that such an application deletes those elements from the correspondence graph that could not be mapped to elements in the source graph any longer and additionally deletes the preimages in the correspondence graph of all deleted elements from the target graph as well (if there are any). The next lemma states that the application of a source rule is well-defined, i.e., that the mentioned partial morphism actually exists.

Lemma 6 (Correctness of application of source rules). *Let a (non-monotonic) triple graph rule*

$$p = (L_{SCT} \xleftarrow{(l_S, l_C, l_T)} K_{SCT} \xrightarrow{(r_S, r_C, r_T)} R_{SCT})$$

with source rule p_S *and projection* p_S^{pr} *to the source part be given. Given a match* m_S *for* p_S *to a triple graph* $G = (G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T)$ *such that* $G_S \Rightarrow_{p_S^{\mathrm{pr}}, m_S} H_S$*, the partial morphism* $D_C \dashrightarrow H_S$ *as described in Definition 5 exists.*

The next theorem states that a sequential application of a source and a forward rule indeed coincides with an application of the original rule as long as the matches are consistent. This means that the forward rule has to match the RHS R_S of the source rule again and the LHS L_C of the correspondence rule needs to be matched in such a way that all elements not belonging to the domain of the partial morphism from correspondence to source part in the input model are deleted. The forward rule application defined in Definition 5 fulfills this condition by construction.

Theorem 7 (Synthesis of rule applications). *Let a triple graph rule* p *with source and forward rules* p_S *and* p_F *be given. If there are applications* $G \Rightarrow_{p_S, m_S} G'$ *with co-match* n_S *and* $G' \Rightarrow_{p_F, m_F} H$ *with* $m_F = (n_S, m_C, m_T)$ *as constructed above, then there is an application* $G \Rightarrow_{p,m} H$ *with* $m = (m_S, m_C, m_T)$*.*

4.2 Language-Preserving Short-Cut Rules

In this section we identify sufficient conditions for an application of a short-cut rule that guarantee the result to be an element of the language of the original grammar. Since our conditions apply to arbitrary adhesive categories and are not specific for TGGs, we present the result in its general form.

Theorem 8 (Characterization of valid applications). *In an adhesive category $\mathcal{C}$, given a sequence of transformations*

$$G \Rightarrow_{r,m} G_0 \Rightarrow_{p_1,m_1} G_1 \Rightarrow_{p_2,m_2} \cdots \Rightarrow_{p_t,m_t} G_t \Rightarrow_{r^{-1} \ltimes_k r', m_{sc}} H$$

with rules $p_1, \ldots, p_t$ and $r^{-1} \ltimes_k r'$ being the short-cut rule of monotonic rules $r : L \hookrightarrow R$ and $r' : L' \hookrightarrow R'$ along a common kernel k, there is a match m' for r' in G and a transformation sequence

$$G \Rightarrow_{r',m'} G'_1 \Rightarrow_{p_1,m'_1} \ldots G'_{t-1} \Rightarrow_{p_t,m'_t} H,$$

provided that

1. *the application of $r^{-1} \ltimes_k r'$ with match m_{sc} is sequentially independent of the sequence of transformations $G_0 \Rightarrow_{p_1,m_1} G_1 \Rightarrow_{p_2,m_2} \cdots \Rightarrow_{p_t,m_t} G_t$ and*
2. *the thereby implied match m'_{sc} for $r^{-1} \ltimes_k r'$ in G_0, restricted to the RHS R of r, equals the co-match $n : R \hookrightarrow G_0$ of the transformation $G \Rightarrow_{r,m} G_0$ (i.e., $m'_{sc} \circ j_R = n$ where j_R embeds R into the LHS of $r^{-1} \ltimes_k r'$ as in Fig. 6).*

In particular, given a grammar $GG = (\mathcal{R}, S)$ such that $r, r', p_1, \ldots, p_t \in \mathcal{R}$ and $G \in \mathcal{L}(GG)$, then $H \in \mathcal{L}(GG)$.

Independence of the short-cut rule application $t_{sc} : G_t \Rightarrow_{r^{-1} \ltimes_k r', m_{sc}} H$ from the preceding transformation sequence $t : G \Rightarrow G_t$ requires the existence of morphisms in two directions: morphisms d_2^i from the LHS of the short-cut rule to the context objects D_i arising in t and morphisms d_1^i from the right-hand sides R_i of the rules p_i to the context object of t_{sc} (shifted further and further to the beginning of the sequence). In the case of (typed triple) graphs, the existence of morphisms d_2^i ensures that none of the rule applications in t enabled the transformation t_{sc}. The existence of morphisms d_1^i ensures that the transformation t_{sc} does not delete structure needed to perform the transformation sequence t.

Application to model synchronization. The results in Theorems 7 and 8 are the formal basis for an automatic construction of repair rules. Theorem 7 ensures that a suitable edit action followed by application of a repair rule at the right match is equivalent to the application of a short-cut rule. Thus, whenever an edit action on the source model (or symmetrically the target model) corresponds to the source-action (target-action) of a short-cut rule, application of the corresponding forward (backward) rule synchronizes the model again. Since the language of a TGG is defined by its rules, every valid model can be reached from every other valid model by inverse application of some of the rules of the grammar followed by normal application of some rules. Often, edit actions are rather small steps

(or at least consist of those). Thus, it is not unreasonable to expect that many typical edit actions can be realized as short-cut rules as these formalize the inverse application of a rule followed by application of a normal one. Theorem 8 characterizes the matches for short-cut rules at which application stays in the language of the TGG. For operational short-cut rules, this can either be used for detecting invalid edit actions or determining valid matches for synchronizing forward rules.

5 Implementation and Evaluation

Implementation. Our implementation[1] of an optimized model synchronizer is based on the existing EMF-based general purpose graph and model transformation tool eMoflon [21]. It offers support for rule-based unidirectional and bidirectional graph transformations where the latter is based on TGGs. To support an effective model synchronizer, we automatically calculate a small but useful subset of all possible short-cut rules. This is done by overlapping as many created elements as possible and only varying in the way that context elements are mapped onto each other. These selected short-cut rules are operationalized to get repair rules that allow us to repair broken links similar to our example in Sect. 2. The model synchronization process is based on an *incremental graph pattern matcher* that tracks all matches that dis-/appear due to model changes. Thus, it offers the ability to react to model changes without the need to recompute matches from scratch. Our implementation uses this technique by processing all those matches marked as broken by the pattern matcher after a model change. A broken match is the starting point to find a repair match as it is defined by the co-match of the performed model change and has to be extended. If the pattern matcher can extend a broken match to a repair match, the corresponding *short-cut* repair rule can be applied. Otherwise, we fall back to the old synchronization strategy of revoking the current step. This completely automatized synchronization process ensures that we are able to restore consistency as long as the edited domain model still resides in the language of our TGG.

Evaluation. Our experimental setup consists of 23 TGG rules (shown in our technical report [9]) that specify consistency between Java AST and custom documentation models and 37 short-cut rules derived from our TGG rule set. A small modified excerpt of this rule set was given in Sect. 2. For this evaluation, however, we define consistency not only between *Package* and *Folder* hierarchies but also between type definitions, e.g., *Classes* and *Interfaces*, and *Methods* with their corresponding documentation entries. We extracted five models from Java projects hosted on Github using the tool MoDisco [4] and translated them into our own documentation structure. Also, we generated five synthetic models consisting of n-level *Package* hierarchies with each non-leaf*Package* containing five sub-*Packages* and each leaf *Package* containing five *Classes*. Given such Java

[1] Both the implementation and evaluation workspace can be accessed via https://github.com/Arikae00/FASE19_eMoflon-evaluation.

models, we refactored each model in three different scenarios such as by moving a *Class* from one *Package* to another or completely relocating a *Package*. Then we used eMoflon to synchronize these changes in order to restore consistency to the documentation model, with and without *repair rules*.

These synchronization steps are subject to our evaluation and we pose the following research questions: **(RQ1)** *For different kinds of changes, how many elements can be preserved that would otherwise be deleted and recreated?* **(RQ2)** *How does our new approach affect the runtime performance?* **(RQ3)** *Are there specific scenarios in which our approach performs especially good or bad?*

Repair rules were developed to avoid unnecessary deletions of elements by reverting too many rule applications in order to restore consistency as shown exemplary in Sect. 2. This means that model changes where our approach should perform especially good, have to target rule applications close to the beginning of a rule sequence as this possibly renders many rule applications invalid. This means that altering a root *Package* by creating a new *Package* as root would imply that many rule applications have to be reverted to synchronize the changes correctly (Scenario 1). In contrast, our approach might perform poorly when a model change does not inflict a large cascade of invalid rule applications. Hence, we move *Classes* between *Packages* to measure if the effort of applying *repair rules* does infer a performance loss when both the new and old algorithm do not have to repair many broken rule applications (Scenario 2). Finally, we simulate a scenario between the first two by relocating leaf *Packages* (Scenario 3).

Table 1. Legacy vs. new synchronizer – Time in sec. and number of created elements

	Both		Legacy Synchronization						Synchro. by Repair Rules					
	Trans.		Scen. 1		Scen. 2		Scen. 3		Scen. 1		Scen. 2		Scen. 3	
Models	Sec	Elts	Sec	Elts	Sec	Elts	Sec	Elts	Sec	Elts	Sec	Elts	Sec	Elts
lang.List	0.3	25	0.2	20	–	–	0.06	5	0.2	0	–	–	0.03	0
tgg.core	6.4	1.6k	39	1.6k	3.8	99	0.64	17	0.8	0	0.11	0	0.05	0
modisco.java	9.9	3.2k	228	3.3k	18.6	192	3.6	33	2.5	0	0.2	0	0.09	0
eclipse.graphiti	20.7	6.5k	704	6.5k	63.9	490	5.65	25	6.1	0	0.21	0	0.09	0
eclipse.compare	10.74	3.8k	83	3.7k	3.1	76	2.36	47	0.7	0	0.08	0	0.04	0
synthetic $n = 1$	0.3	35	0.32	30	0.2	30	0.03	1	0.1	0	0.05	0	0.03	0
synthetic $n = 2$	0.9	160	1.03	155	0.3	30	0.03	1	0.1	0	0.05	0	0.02	0
synthetic $n = 3$	2.8	785	6	780	0.4	30	0.04	1	0.1	0	0.07	0	0.02	0
synthetic $n = 4$	13.5	3.9k	86.3	3.9k	1.2	30	0.08	1	0.4	0	0.14	0	0.04	0
synthetic $n = 5$	91.5	20k	2731	20k	17.4	30	0.14	1	1.5	0	0.37	0	0.09	0

Table 1 depicts the measured times (Sec) and the number of created elements (Elts) in each scenario. Each created element also represents a deleted element, e.g., through revoking and reapplying a rule or applying a repair rule that creates and deletes elements. In more detail, the table shows measurements for the initial translation of the MoDisco model into the documentation structure and

synchronization steps for each scenario using the legacy synchronizer without *repair rules* and the new synchronizer with *repair rules*.

W.r.t. our research questions stated above, we interpret this table as follows: The right columns of the table show clearly that using repair rules preserves all those elements in our scenarios that would otherwise be deleted and recreated by the legacy algorithm[2] **(RQ1)**. The runtime shows a significant performance gain for Scenario 1 including a worst-case model change **(RQ2)**. *Repair rules* do not introduce an overhead compared to the legacy algorithm as can be seen for the synthetic time measurements in Scenario 3 where only one rule application has to be repaired or reapplied. **(RQ2)**. Our new approach excels when the cascade of invalidated rule applications is long. Even if this is not the case, it does not introduce any measurable overhead compared to the legacy algorithm as shown in Scenarios 2 and 3 (**RQ3**).

Threats to validity. Our evaluation is based on five real world and five synthetic models. Of course, there exists a wide range of projects that differ significantly from each other due to their size, purpose, and developer styles. Thus, the results may probably differ for other projects. Nonetheless, we argue that the four larger projects extracted from Github are representative since they are part of established tools from the Eclipse community. In this evaluation, we selected three edit operations that are representative w.r.t. their dependency on other edit operations. They may not be representative w.r.t. other aspects such as size or kind of change, which seems to be of minor importance in this context. Also we limited our evaluation to one TGG rule set due to space issues. However, in our experience the approach shows similar results for a broader range of TGGs which can be accessed through eMoflon.

6 Related Work

Reuse in existing work on TGGs. Several approaches to model synchronization based on TGGs suffer from the fact that the revocation of a certain rule application triggers the revocation of all dependent rule applications as well [12,16,19]. Especially from a practical point of view such cascades of deletions shall be avoided: In [10], Giese and Hildebrandt propose rules that save nodes instead of deleting and then re-creating them. Their examples can be realized by our construction of *repair rules*. But they do not present a general construction or proof of correctness. This is left as future work in [11] again, where other aspects of [10] are formalized and proven to be correct.

In [3], Blouin et al. added a specially designed repair rule to the rules of their case study to avoid information loss. Greenyer et al. [14] also propose to not directly delete elements but to mark them for deletion and allow for reuse of these marked elements in other rule applications. But this approach comes without any formalization or proof of correctness as well. Again, the given example can be realized as short-cut repair. These uncontrolled and informal approaches are

[2] Scenario 1: We expect the new root element to already be translated.

potentially harmful. Re-using elements wrongly may lead to, e.g., containment cycles or unconnected data. Hence, providing precise and sufficient conditions for correct re-use of data is highly desirable as re-use may improve scalability and decrease data-loss. Our short-cut rules formalize when data can be correctly reused. In summary, we do not only offer a unifying principle behind different practically used improvements of TGGs but also give a precise formalization that allows for automatic construction of the rules needed. Thereby, we present conditions under which rule applications lead to valid outputs.

Comparison to other bx approaches. Anjorin et al. [2] compared three state-of-the-art bx tools, namely eMoflon [21] (rule-based), mediniQVT [1] (constraint-based) and BiGUL [17] (bx programming language) w.r.t. model synchronization. They point out that synchronization with eMoflon is faster than with both other tools as the runtime of these tools correlates with the overall model size while the runtime of eMoflon correlates with the size of the changes done by edit operations. Furthermore, eMoflon was the only tool able to solve all but one synchronization scenario. One scenario was not solved because it deleted more model elements than absolutely necessary in that case. Using short-cut repair rules, we can solve the remaining scenario and moreover, can further increase eMoflons model synchronization performance.

Change-preserving model repair. Change-preserving model repair as presented in [22,25] is closely related to our approach. Assuming a set of consistency-preserving rules and a set of edit rules to be given, each edit rule is accompanied by one or more repair rules completing the edit step, if possible. Such a complement rule is considered as repair rule of an edit rule w.r.t. an overarching consistency-preserving rule. Operationalized TGG rules fit into that approach but provide more structure: As graphs and rules are structured in triples, a source rule is also an edit rule being complemented by a forward rule. In contrast to that approach, source and forward rules can be automatically deduced from a given TGG rule. By our use of short-cut rules we introduce a pre-processing step to first enlarge the sets of consistency-preserving rules and edit rules.

Generalization of correspondence relation. Golas et al. provide a formalization of TGGs in [13] which allows to generalize correspondence relations between source and target graphs as well. They use special typings for the source, target, and correspondence parts of a TGG and for edges between a correspondence part and source and target part instead of using graph morphisms. That approach also allows for partial correspondence relations. But it makes the deletion of elements more complex as it becomes important how many incident edges a node has (at least in the double-pushout approach). We therefore opted for introducing triple graphs with partial morphisms. They allow us to just delete a node without caring if it is needed within an existing correspondence relation.

7 Conclusion

Model synchronization, i.e., the task of restoring consistency between two models after a model change, poses challenges to modern bx approaches and tools: We expect them to synchronize changes without losing data in the process, thus, preserving information and furthermore, we expect them to show a reasonable performance. While Triple Graph Grammars (TGGs) provide the means to perform model synchronization tasks in general, both requirements cannot always be fulfilled since basic TGG rules do not define the adequate means to support intermediate model editing. Therefore, we propose additional edit operations being short-cut rules, a special form of generalized TGG rules that allow to take back one edit action and to perform an alternative one. In our evaluation, we show that operationalized short-cut rules allow for a model synchronization with considerably decreased data loss and improved runtime.

To better cope with practical application scenarios, we like to extend our approach by formally incorporating type inheritance, application conditions and attributes in the model synchronization process. Since all of these have been formalized in the setting of ($\mathcal{M}$-)adhesive categories and our present work uses that framework as well, these extensions are prepared but up to future work. Propagating changes from one domain to another is basically done here by operationalizing short-cut rules. A more challenging task is what we call model integration where related pairs of models are edited concurrently and have to be synchronized. These model edits may be in conflict across model boundaries. It is up to future work to allow short-cut rules in model integration. Our hope is to decrease data loss and to improve runtime of model integration tasks as well.

References

1. Ikv++: Medini QVT. http://projects.ikv.de/qvt
2. Anjorin, A., Diskin, Z., Jouault, F., Ko, H., Leblebici, E., Westfechtel, B.: Benchmarx reloaded: a practical benchmark framework for bidirectional transformations. In: 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, pp. 15–30 (2017). http://ceur-ws.org/Vol-1827/paper6.pdf
3. Blouin, D., Plantec, A., Dissaux, P., Singhoff, F., Diguet, J.-P.: Synchronization of models of rich languages with triple graph grammars: an experience report. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 106–121. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_8
4. Bruneliàre, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014). https://doi.org/10.1016/j.infsof.2014.04.007
5. Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: On principles of least change and least surprise for bidirectional transformations. J. Object Technol. **16**(1), 3:1–3:31 (2017). https://doi.org/10.5381/jot.2017.16.1.a3
6. 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

7. Eppinger, S.D.: Model-based approaches to managing concurrent engineering. J. Eng. Des. **2**(4), 283–290 (1991). https://doi.org/10.1080/09544829108901686
8. Fritsche, L., Kosiol, J., Schürr, A., Taentzer, G.: Short-cut rules. Sequential composition of rules avoiding unnecessary deletions. In: Mazzara, M., Ober, I., Salaün, G. (eds.) STAF 2018. LNCS, vol. 11176, pp. 415–430. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_30
9. Fritsche, L., Kosiol, J., Schürr, A., Taentzer, G.: Optimizing TGG-based model synchronization by automatic short-cut repair processes: extended version. Technical report, Philipps-Universität Marburg (2019). https://www.uni-marburg.de/fb12/arbeitsgruppen/swt/forschung/publikationen/2019/FKST19-TR.pdf
10. Giese, H., Hildebrandt, S.: Efficient model synchronization of large-scale models. Technical report 28, Hasso-Plattner-Institut (2009)
11. Giese, H., Hildebrandt, S., Lambers, L.: Bridging the gap between formal semantics and implementation of triple graph grammars. Softw. Syst. Model. **13**(1), 273–299 (2014). https://doi.org/10.1007/s10270-012-0247-y
12. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. **8**(1), 21–43 (2009). https://doi.org/10.1007/s10270-008-0089-9
13. Golas, U., Lambers, L., Ehrig, H., Giese, H.: Toward bridging the gap between formal foundations and current practice for triple graph grammars. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 141–155. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_10
14. Greenyer, J., Pook, S., Rieke, J.: Preventing information loss in incremental model synchronization by reusing elements. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 144–159. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21470-7_11
15. Hermann, F., Ehrig, H., Golas, U., Orejas, F.: Efficient analysis and execution of correct and complete model transformations based on triple graph grammars. In: Proceedings of the First International Workshop on Model-Driven Interoperability. pp. 22–31. MDI 2010. ACM, New York (2010). https://doi.org/10.1145/1866272.1866277
16. Hermann, F., et al.: Model synchronization based on triple graph grammars: correctness, completeness and invertibility. Softw. Syst. Model. **14**(1), 241–269 (2015). https://doi.org/10.1007/s10270-012-0309-1
17. Ko, H., Zan, T., Hu, Z.: BiGUL: a formally verified core language for putback-based bidirectional programming. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 61–72 (2016). https://doi.org/10.1145/2847538.2847544
18. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. Theor. Inform. Appl. **39**(3), 511–545 (2005). https://doi.org/10.1051/ita:2005028
19. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient model synchronization with precedence triple graph grammars. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 401–415. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_27
20. Leblebici, E., Anjorin, A., Fritsche, L., Varró, G., Schürr, A.: Leveraging incremental pattern matching techniques for model synchronisation. In: de Lara, J., Plump, D. (eds.) ICGT 2017. LNCS, vol. 10373, pp. 179–195. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_11

21. Leblebici, E., Anjorin, A., Schürr, A.: Developing eMoflon with eMoflon. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 138–145. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_10
22. Ohrndorf, M., Pietsch, C., Kelter, U., Kehrer, T.: Revision: a tool for history-based model repair recommendations. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 105–108. ACM (2018). https://doi.org/10.1145/3183440.3183498
23. Robinson, E., Rosolini, G.: Categories of partial maps. Inf. Comput. **79**(2), 95–130 (1988). https://doi.org/10.1016/0890-5401(88)90034-X
24. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59071-4_45
25. Taentzer, G., Ohrndorf, M., Lamo, Y., Rutle, A.: Change-preserving model repair. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 283–299. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_16

Offline Delta-Driven Model Transformation with Dependency Injection

Artur Boronat(✉)

Department of Informatics,
University of Leicester, Leicester, UK
aboronat@le.ac.uk

Abstract. When model transformations are used to implement consistency relations between very large models (VLMs), incrementality plays a cornerstone role in the realization of practical consistency maintainers. State-of-the-art model transformation engines with support for incrementality normally rely on a publish-subscribe model for linking model updates – deltas – to the application of model transformation rules, in so called dependencies, at run time. These deltas can then be propagated along an already executed model transformation. A small number of such engines use domain-specific languages (DSLs) for representing model deltas offline in order to enable their use in asynchronous, event-based execution environments.

The principal contribution of this work is the design of a forward delta propagation mechanism for incremental execution of model transformations, which decouples dependency tracking from delta propagation using two innovations. First, the publish-subscribe model is replaced with dependency injection, physically decoupling domain models from consistency maintainers. Second, a standardized representation of model deltas is reused, facilitating interoperability with EMF-compliant tools, both for defining deltas and for processing them asynchronously. This procedure has been implemented in a model transformation engine, whose performance has been evaluated empirically using the VIATRA CPS benchmark. In the experiments performed, the new transformation engine shows gains in the form of several orders of magnitude in the initial phase of the incremental execution of the benchmark model transformation and delta propagation is performed in real time, independently of the size of the models involved, whereas the up-to-now best-performant approach is dependent.

Keywords: Mappings between languages · Traceability · Incremental model transformation · Performance benchmark

1 Introduction

Significant issues in the application of Model-Driven Engineering (MDE) in large-scale industrial problems stem from interoperability and scalability of

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 134–150, 2019.
https://doi.org/10.1007/978-3-030-16722-6_8

current MDE tools [1,16,17]. Model transformation, widely accepted as the *heart and soul* of MDE [23], deals with model manipulation either by translating models or by synchronizing them. Current tool support for model transformation is a key root cause for many of the bottlenecks hampering scalability in MDE [2,8]. This is particularly crucial when transformations are used to implement consistency maintainers between very large models (VLMs), consisting of milions of elements. In this context, incrementality ensures that only those parts of the model that are inconsistent or that have been modified − a model delta − are transformed or, more precisely, propagated along an already executed transformation [11,12].

Current state-of-the-art approaches that support incremental execution of model transformations share common features: the delta propagation mechanism is usually decoupled from the delta detection mechanism in order to facilitate maintainability of the consistency maintainer; and deltas are represented either in memory for synchronous notification or offline, with dedicated domain-specific languages, for asynchronous notification. The most mature tools rely on a publish/subscribe mechanism, where model deltas are notified at run time whenever a model is updated. This notification mechanism is synchronous and loosely couples model updates with the delta propagation mechanism, facilitating maintainability of the underlying transformation engine after fixing the type of notification. However, it usually requires an observer for each object that can be modified, with a consequent impact on performance, and the model transformation must be live, in memory, in order to listen for changes. These problems can be avoided by using offline deltas. The publish/subscribe mechanism can be extended to enable asynchronous delta notification but this is normally achieved by using dedicated domain-specific languages to represent deltas offline, which do not involve standardized formats, hindering the interoperability of those transformation engines in existing modeling tool ecosystems.

In this paper, the design of a forward delta propagation procedure is presented for executing model transformations in incremental mode that can handle documented change scenarios [4], i.e. documents representing a change to a given source model. Such documents are defined with the EMF change model [24], both conceptually and implementation-wise, guaranteeing interoperability with EMF-compliant tools. This design decision replaces a publish/subscribe notification with dependency injection: each notification is directly performed by the implementation of the domain model at run time by injecting the dependency corresponding to the model update that has been performed. Aspect-oriented programming is used to weave code into an already existing implementation of a domain model totally decoupling domain models from the consistency maintainer at design time. The proposed forward delta propagation procedure has been implemented in YAMTL [6], a model transformation engine for VLMs, enabling the execution of model transformations both in batch mode and in incremental mode without additional user specification overhead. This new extension dramatically improves the performance of the batch execution mode when dealing with sparse model deltas, which can be propagated in real time (i.e. in μs.).

This work is structured as follows: Sect. 2 provides a self-contained description of the class of model transformations supported using a class diagram to relational schema model transformation; Sect. 3 presents the forward propagation procedure implemented in the model transformation engine together with the main innovations; Sect. 4 discusses the performance of the transformation engine with an adaptation of the VIATRA CPS benchmark; Sect. 5 discusses related work from reactive and bidirectional model transformation.

2 Model Transformation: A Running Example

The type of model transformations that are considered in this work are classified as unidirectional and out-place. For example, when considering the well-known example that maps class diagrams to relational schemas, a class diagram is used by queries to extract information and a relational schema is built from scratch. If we consider a graph transformation perspective, both models are considered to form part of the same graph in order to enable transformation by rewriting. In that case, we are only considering transformations where the two models are two clearly disjoint subgraphs and where rewriting is performed deterministically.

In this work, model transformations are represented using an implementation-agnostic graphical syntax, quite close to that used in the graph transformation literature. In this representation, metamodels are given as class diagrams, the abstract syntax of models is given as object diagrams and model transformations are represented as a collection of rules, where each rule is defined as a pair of model patterns, called left-hand side (LHS) and right-hand side (RHS). The notion of metamodel, model and model pattern correspond to those of type graph, attributed graph with containments and node inheritance, and graph pattern in the graph transformation literature [5,10]. For example, the rules `A->C` and `R->FK` of Fig. 1 map attributes to columns. The $ before a variable denotes string interpolation.

Graph patterns in rules can be augmented with universally quantified variables (represented by an overlaid box). Moreover, rules are augmented with a `when` clause to express conditions that must be satisfied by the variables in LHS, and with a `where` clause to indicate how variables from LHS and from RHS are related via the application of other rules, expressed as two graph patterns. Formulas in a `when` clause may be expressed in conjunctive form, as all filter conditions must be satisfied in order for the rule to be applied, whereas formulas in a `where` clause may be expressed in disjunctive form (assuming mutually exclusive conditions), as all the side effects expressed in a `where` clause must be evaluated. The variables of RHS of the main rule must appear either in the LHS of the main rule or in the RHS of a `where` transformation step. The rule `C->T` of Fig. 1 illustrates how to map a class to a table with a primary key column `PK_COL` and for each attribute `A` whose type is a `DataType`, the corresponding column is obtained by applying a rule, with the rule `A->C`, and for each attribute `OTHER` whose type is the class `C`, matched in LHS of the main rule, a new foreign key column is added to the table `T`, with the rule `R->FK`.

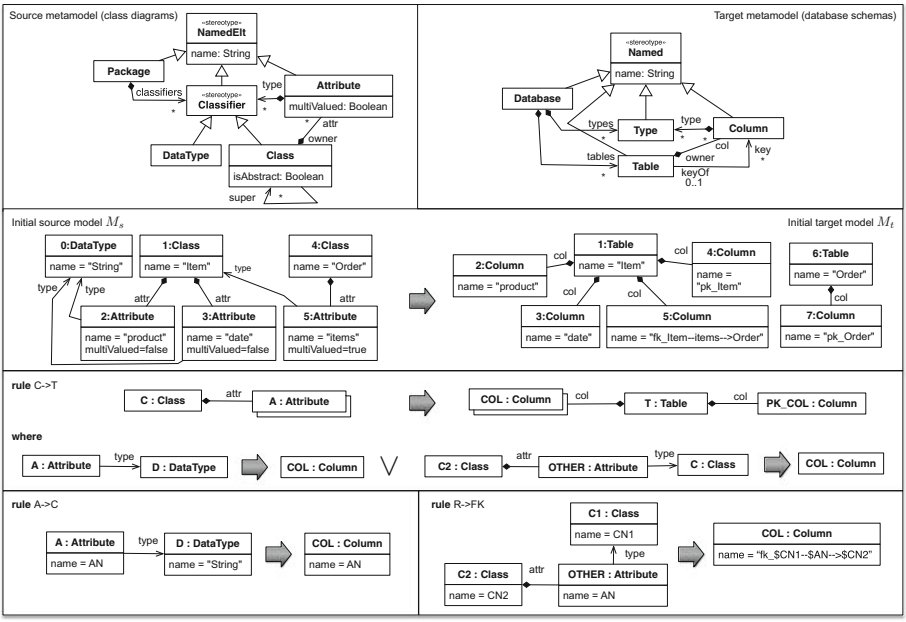


Fig. 1. Metamodels, example and transformation rules.

From an operational point of view, transformation rules are applied unidirectionally from LHS to RHS performing an out-place transformation following two steps. First, during the *matching phase*, matches for the rules in the model transformation are found as long as they are not shared by different rules and these are included in a set *matchPool*. A match is formally defined as a graph morphism from LHS to the source graph, which satisfies the `when` conditions, but it is represented as a map from variables to object identifiers for the sake of presentation in this paper.

Second, during the *execution phase*, each match is processed by triggering the application of a transformation rule, which is represented as a transformation step, denoted by $r : \overrightarrow{in \mapsto \varsigma} \rightarrow \overrightarrow{out \mapsto \varsigma}$, which consists of a labelled pair of two matches, the match for the input pattern of the rule, which enables its application, and the match for the output pattern of the rule, with the objects that result from applying the rule. When a rule is applied, the source model is only used for query purposes but the target model is constructed by adding the pattern of the RHS instantiated with values from the variables both in the LHS and in the RHS of `where` transformation steps. In addition, `where` transformation steps may further expand the structure of the target model. This execution model resembles the application of forward rules used in triple graph grammars (TGGs) [22], where the source graph is annotated as rules are applied and only the target graph is constructed together with a link in a correspondence graph, where each link denotes a transformation step.

3 Delta-Driven Model Transformations

This section presents the mechanism to propagate documented deltas δ_t from a source model M_s to a target model M_t in an incremental way, when the (unidirectional) synchronization correspondence between these two models is represented with a model transformation t as described in the previous section. This has been implemented in the YAMTL transformation engine [6], which has been extended with two modes of execution: *initialization*, the transformation is executed in batch mode but, additionally, tracks those parts of the source model involved in transformation steps as *dependencies*; *propagation*, the transformation is executed incrementally for a given source delta.

In order for a model transformation to be executed in propagation mode, it first needs to be executed in initialization mode in order both to create transformation steps and to inject the dependencies that facilitate the analysis of the impact of changes in the already executed model transformation. Therefore, the transformation t is applied to M_s using the original batch semantics [6] while injecting dependencies in the transformation engine. Once the initialization is done, any number of source forward deltas δ_s can be propagated.

Given a source documented delta δ_s between a source model M_s, already synchronized with a target model M_t via a model transformation $t : M_s \xrightarrow{*} M_t$ (where $\xrightarrow{*}$ denotes a sequence of transformation steps), and an updated source model M'_s, the transformation engine propagates the model update δ_s along t. The effect of this forward propagation is the application of an update δ_t on the target model M_t.

In the following subsections, we explain the different phases of the new execution modes, initialization and propagation, in more detail. As the initialization mode faithfully corresponds to the batch execution of a model transformation, the discussion of this mode focuses on the type of dependencies that are injected in the transformation engine in Sect. 3.1. The discussion on the propagation mode focuses on how deltas are represented in Sect. 3.2. Then, the two main phases of the propagation execution mode, namely impact analysis and delta propagation, are explained in Sects. 3.3 and 3.4, respectively.

3.1 Dependency Injection

When running a model transformation in initialization mode, the engine monitors the source model and whenever an object ς is matched or a feature call, represented as a pair (ς, f) of an EMF object ς and a feature name f, is performed, a dependency is injected into the dependency registry. A dependency thereby links either an object ς or a feature call (ς, f) to transformation steps $r : \overrightarrow{in \mapsto \varsigma} \rightarrow \overrightarrow{out \mapsto \varsigma}$ in which it is used. Such dependencies are detected both during the matching phase and during the execution phase.

In the matching phase, while finding a match for a rule, the engine keeps track of all of the feature calls used in both element and rule `when` conditions. When a match is found to be valid, the collection of dependencies is injected into the dependency registry for the transformation step that uses that match. Otherwise,

Table 1. Analysis of dependencies for the initial MT $t : M_s \xrightarrow{*} M_t$ of Fig. 2.

Rule	Source Match	Target Match	Dependencies from M_s
C->T	c $\mapsto$ 1	t $\mapsto$ 1,	(1,name), (1,att),
		pk_col $\mapsto$ 4	(5, type), (5, multiValued)
C->T	c $\mapsto$ 4	t $\mapsto$ 6, pk_col $\mapsto$ 7	(4, name), (4, attr)
A->C	att $\mapsto$ 2	col $\mapsto$ 2	(2, name)
A->C	att $\mapsto$ 3	col $\mapsto$ 3	(3, name)
R->FK	ref $\mapsto$ 5	fk_col $\mapsto$ 5	(5,name), (5,type),
		fk_col $\mapsto$ 5	(1, name), (4,name)

when the match is not valid, the collected dependencies are discarded. Additionally, when inserting a match in the *matchPool*, the transformation engine also records reverse matches as injected dependencies between matched objects ς and the transformation step in which they are matched.

Dependencies may also be found when executing a transformation step, e.g., while executing initialization expressions associated with attributes in model patterns in RHS and in `where` clauses. In such cases, the transformation engine injects a dependency for the transformation step every time a feature call in the source model is detected. As a result, note that several transformation steps may depend on the same object ς, when rules have more than one single input element, or on the same feature call (ς, f).

Table 1 shows the dependencies that are found when executing the transformation of Fig. 1 in initialization mode from model M_s. Each row in the table represents a transformation step, where: the source match indicates where the rule has been applied, the target match indicates what objects were created, and dependencies refers to the set of feature calls associated with a transformation step. Reverse matches are extracted from source matches, by reading them in the opposite direction.

Dependency injection is configured with an aspect whose pointcut matches feature calls under a user-defined namespace. Hence, the model transformation engine is entirely decoupled from the domain model at design time. They become tightly coupled at compilation time and, hence, at run time.

3.2 Representable Deltas

The EMF change model [24] is used to represent deltas to an instance of any other EMF model. It is built-in in EMF and, therefore, available for any EMF-compliant tool. In this section, we describe how a documented delta is represented with the EMF change model and how it can be automatically defined given any potentially *live* atomic update.

A delta consists of a `ChangeDescription` which contains a map of `objectChanges`, which refer to those objects that are updated and, for each such object, it contains a list of `FeatureChange`s. A `FeatureChange` (`FC`) refers

to the structural feature that needs to be updated and provides the new value. For single-valued attributes, a `FeatureChange` contains the new `dataValue` if the feature is an attribute. For references and multi-valued attributes, a `FeatureChange` includes a containment reference `listChanges` pointing to `ListChange`. `ListChange`s are used to represent addition to, removal from, or movement *within* the given feature values. In particular, movement only captures when an object changes to a different index within the collection. However, it does not capture structural changes, e.g. change of container, which are represented as a removal from and an addition to the corresponding containment references. When a `FeatureChange` refers to a containment reference, objects to be added are pointed by `objectsToAttach` and objects to be removed are pointed by `objectsToDetach`.

`FeatureChange`s capture when a feature value is updated for an object but EMF also permits adding and removing root objects to a resource, representing the model in memory, which need not be contained by any other object. Such changes are considered to be performed on the resource itself and are represented with `ResourceChange`s, one for each changed resource. A `ResourceChange` (`RC`) contains the `ListChange`s for the root objects of the corresponding resource, similarly to multi-valued features. For a more detailed explanation of the EMF change model, we refer the reader to [24].

Table 2 shows a classification of atomic model updates that are representable with the EMF change model as explained above. Note that moving and object structurally, case 12 – *move (inter.)*, – is represented in a composite delta by two opposite actions, removing the object either from the root contents of the resource – if it is a root object (case 2) – or from a containment reference – if it is a contained object (case 10) – and adding it either to the root contents of the resource – if it is to become a root object (case 1) – or to another containment reference in another container object (case 9). This case is not captured by the EMF change model explicitly but the transformation engine is able to infer it, as explained in the following section.

Table 2. Summary of model update types, with their representation in EMF.

Cases	Granularity	Level	Feature	Delta action	Delta representation	*DO*	*DFC*
1,2	atomic	root		add/remove	`RC::listChanges`	✓	
3	atomic	root		move (intra.)	`RC::listChanges`		
4,5	atomic	any	single-valued att	add/remove	`FC`		✓
6,7	atomic	any	multi-valued att	add/remove	`FC::listChanges`	✓	✓
8	atomic	any	multi-valued att	move (intra.)	`FC::listChanges`		✓
9,10	atomic	any	ref	add/remove	`FC::listChanges`		✓
11	atomic	any	ref	move (intra.)	`FC::listChanges`		✓
12	composite	any	containment ref	move (inter.)	opposite remove and add actions in cases $\{2, 10\}/\{1, 9\}$		✓

A delta, which may represent atomic and composite changes, is defined as an instance of the EMF change model and can be serialized. EMF also provides facilities for applying them and reversing them. Furthermore, EMF provides a change recorder, which enables recording *live updates* as a `ChangeDescription` for either a root object, a collection of root objects, a resource or a resource set. The resulting `ChangeDescription` is the representation of a *history scenario* [4], from the updated model to the original one, which is optimized. That is, atomic changes for the same feature of the same object may be discarded or merged, as long as the optimization process preserves reversibility. Hence, reversing the recorded delta may yield less changes than were originally made. Reversed deltas represent *documented scenarios* and can be propagated along a model transformation, as discussed in subsequent sections.

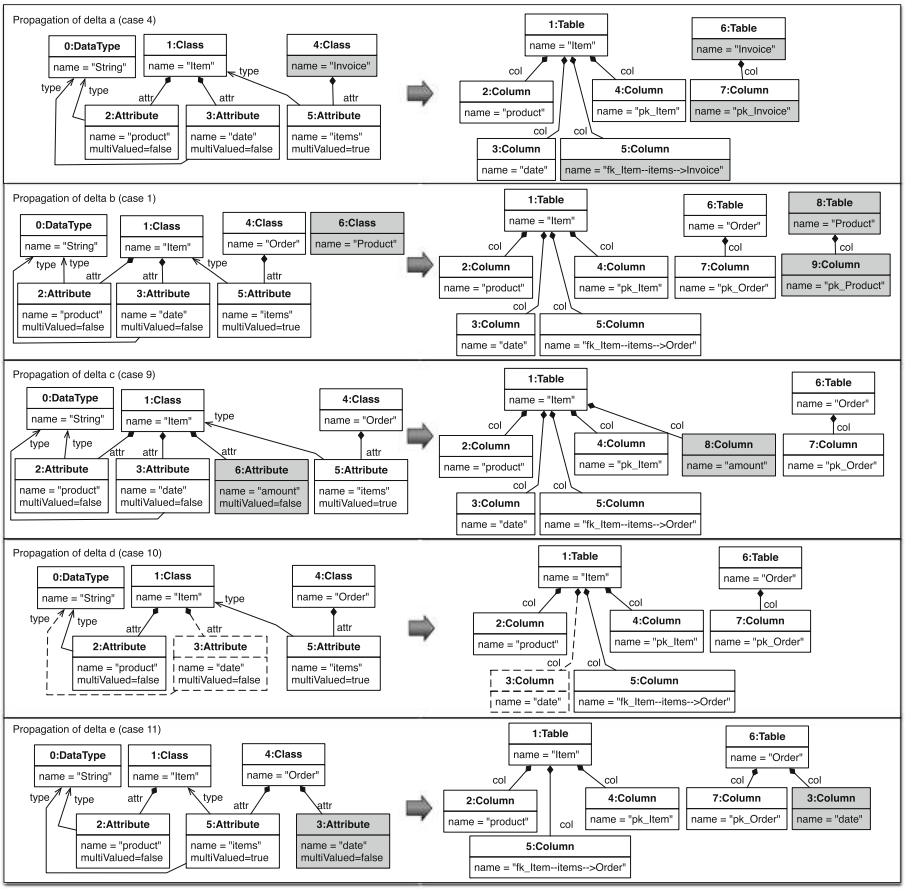


Fig. 2. Source/target metamodels, initial synchronized models and forward delta propagation (a–e).

The EMF change recorder enables the possibility of deferring the observation of updates to the point in which they occur, saving memory resources, and interoperability. Furthermore, recorded (history) deltas can be regarded as a rollback mechanism for implementing transactional model updates, which may be performed live.

Figure 2 shows examples of documented deltas, defined over the source model M_s of the running example. Such deltas are representable as EMF model changes, i.e. operationally, but are graphically depicted using the abstract syntax of M_s, using their state-based representation for the sake of presentation. Additions and updates, including moves, are highlighted in grey colour. Objects that are added, and thus created, have a new identifier. Objects that are updated and/or moved preserve their identifier. Removals are highlighted by using dashed lines for the contour lines of the corresponding shapes. The given deltas are instantiations of case 4 (delta `a`), changing the name of the class `Order` to `Invoice`; case 1 (delta `b`), adding a root class `Product`; case 9 (delta `c`), adding a single-valued attribute `amount` to class `Item`; case 10 (delta `d`), removing the attribute `date` from class `Item`; and case 11 (delta `e`), structurally moving the attribute `date` from class `Item` to class `Order`.

In the following subsections, the different phases of the procedure for forward propagation of source deltas is discussed and the aforementioned examples will be used for illustrating them.

3.3 Impact Analysis

In this subsection, we discuss how source documented deltas are analyzed in order to determine which transformation steps are affected by source changes. This analysis is comprised of three main steps: identification of atomic model updates from a documented delta, initialization of locations for newly enabled rules, and marking of transformation steps impacted by changes.

Identification of atomic model updates. In the first step, the transformation engine infers which objects and which feature calls have been impacted by changes. For objects, it also infers whether an object has been added or removed, ignoring if the object is moved, either within the same collection or structurally.

For affected objects, such information is recorded in the set *DO* of *dirty objects* of the form $(\varsigma, ctype)$, where ς is the affected object and *ctype* is the type of change from the set $\{$ `ADD`, `DEL`$\}$. To obtain a dirty object from the delta, `FeatureChange`s and `ResourceChange`s are traversed considering two cases: when an object ς is added either to a containment feature (for a `FeatureChange`) or to the root contents of the resource (for a `ResourceChange`) and such object is not removed elsewhere in the delta, either from a containment reference or from the root contents of the resource; and, similarly, when an object is deleted and it is not added elsewhere in the delta. *DO* is augmented with $(\varsigma,$ `ADD`$)$ in the first case and with $(\varsigma,$ `DEL`$)$ in the second case.

For affected feature calls, such information is recorded in the set *DFC* of *dirty feature calls* of the form (ς, f), where ς is an object and f is a feature

Table 3. Impact analysis of source deltas a–e.

	Case	*DO*	*DFC*	Rule	Source Match	Target Match	$matchPool_\Delta$	dirty?
a	4	–	(4, `name`)	`C->T`	`c` ↦ 4	`t` ↦ 6, `pk_col` ↦ 7	✓	✓
b	1	(6, *ADD*)	–	`C->T`	`c` ↦ 6		✓	
c	9		(1, `attr`)	`C->T`	`c` ↦ 1	`t` ↦ 1, `pk_col` ↦ 4	✓	✓
		(6, *ADD*)		`A->C`	`att` ↦ 6		✓	
d	10		(1, `attr`)	`C->T`	`c` ↦ 1	`t` ↦ 1, `pk_col` ↦ 4	✓	✓
		(3, *DEL*)		`A->C`	`att` ↦ 3	`col` ↦ 3		✓
e	11	–	(1, `attr`),	`C->T`	`c` ↦ 1	`t` ↦ 1, `pk_col` ↦ 4	✓	✓
			(4, `attr`)	`C->T`	`c` ↦ 4	`t` ↦ 6, `pk_col` ↦ 7	✓	✓

name. For each `FeatureChange` of an `ObjectChange`, the dirty feature call (ς, f) with the object ς referred by the `ObjectChange` and the feature name f referred to by the `FeatureChange` is added to *DFC*.

Table 2 shows how atomic model update types are represented using the EMF change model (column *delta representation*), internally, using the sets *DO* and *DFC*. Table 3 shows the sets *DO* of dirty objects and *DFC* of dirty feature calls for the source deltas of Fig. 2. Note that the sets *DO* and *DFC* decouple the transformation engine from the EMF change model and provide another entry point for defining deltas programmatically, which can be used for capturing atomic *live changes* received via EMF adapters.

Initialization of delta locations. For each dirty object $(\varsigma,$ `ADD`$)$, the object ς is added to the extent associated with $type(o)$ in the location map used for delta propagation. This potentially enables new matches when rules are matched during the delta propagation phase.

Marking of impacted transformation steps. In this step, transformation steps that are affected by the atomic changes in the source delta are marked as dirty. For each dirty object $(\varsigma,$ `ADD`$) \in DO$, the extent of type `type`(ς) is augmented with ς. This will potentially enable new matches for some rule during the change propagation phase. For each dirty object $(\varsigma,$ `DEL`$) \in DO$, we obtain the list of transformation steps that are affected from the map of reverse matches. Such transformation steps will then remain transient and the objects in their target match will not be linked to other objects in the target models. In particular, note that when processing root objects or a containment reference, an object that is removed in the delta is not present in the updated source model and, therefore, it does not trigger the transformation step that had been executed in the initial transformation.

For each dirty feature call $(\varsigma, f) \in DFC$ we obtain the list of transformation steps that are affected from the registry of dependencies. For each such transformation step, the satisfaction of its source match is checked. If such source match is still valid, then it is inserted into $matchPool_\Delta$, the pool of matches that are used to schedule rule applications during the change propagation phase.

For each atomic change in Fig. 2, Table 3 shows the marking of transformation steps that are (re-)scheduled according to the dependencies of Table 1. In particular, if a transformation step is re-scheduled, its current source and target matches are included, it is marked as dirty and included in $matchPool_{\Delta}$. If a transformation step is not to be re-executed, it is simply marked as dirty. New transformation steps, with fresh matches due to new objects, are scheduled in $matchPool_{\Delta}$. This last step is actually achieved by augmenting the corresponding type extent with the new objects and the matches are scheduled during the change propagation phase, explained in the next subsection.

3.4 Change Propagation

After the impact analysis phase, delta propagation proceeds by executing a model transformation using the matching and execution phases, as outlined in Sect. 2. Figure 2 illustrates the propagation of source deltas according to the model transformation of Fig. 1. We highlight how incrementality has been considered in these two phases below.

Matching Phase. During the matching phase (in batch/initialization execution mode), matches for a given rule are found by traversing objects from the extent of the types associated with the elements of the source pattern of the rule, with the constraints specified in the form of graphical patterns and `when` conditions. In propagation mode, the transformation engine employs the same pattern matching algorithm but it fetches objects from the location map used for delta propagation, initialized during the change impact analysis phase. Therefore, new matches may be found for objects that have been created by the source delta. Those matches are inserted both into *matchPool* and $matchPool_{\Delta}$, scheduling new transformation steps. Table 3 shows that two new transformation steps are scheduled, one for rule `C->T` in delta `b`, and one for rule `A->C` in delta `c`.

Execution Phase. During the execution phase, transformation steps determined by the matches in $matchPool_{\Delta}$ are executed. Such matches originate from the impact analysis phase, corresponding to transformation steps that are *dirty* and need to be re-executed, and from the matching phase above, corresponding to new transformation steps.

The re-execution of a transformation step is performed as in the batch/initialization mode but for the creation of transformation steps. Whereas a newly scheduled transformation step needs to get its output objects initialized (instantiated for output elements), a dirty transformation step *reuses* the objects of the target match and unsets their features. This avoids loss of contextual information, which is not affected by changes, when re-executing a transformation step. In particular, those references to output objects that emerge from the external context are preserved. On the other hand, references from those output objects are re-calculated by re-executing the transformation step. It is worth noting that the transformation engine uses `where` clauses to define references to objects that are created by other rules, which in turn uses a cache mechanism

to avoid re-executing the transformation step that produced it. Therefore, when a dirty transformation rule is re-executed, the initialization of output element bindings are performed again. However, those bindings that are initialized in a `where` clause are also initialized incrementally. That is, only those objects that belong to a match of a new scheduled transformation step will be transformed from scratch. References to already initialized objects will be simply fetched. Hence, the granularity of the target delta is as fine grained (at binding level) as the source delta for the underlying graph structure of the model.

4 Performance Analysis

For the empirical analysis of the incremental execution of model transformations in YAMTL using the propagation procedure presented above, we have used the VIATRA CPS benchmark [27]. The transformation *YAMTL-incr* implemented for our model transformation engine passes the sanity checks of the benchmark. The software artifacts used in this section and the results obtained are publicly available in a GitHub repository [7] and YAMTL is available at https://yamtl.github.io/.

This evaluation is an extension of the one performed for the batch component of the VIATRA CPS benchmark in [6]. From the original VIATRA CPS benchmark, two incremental variants of the transformation implemented with *EMF-IncQuery* have been selected: *ExplicitTraceability* (EXPL) [25] and *QueryResultTraceability* (QRT) [26], out of which the first one is the best performing solution up to now. These transformations have been extracted as independent Java projects. Classes implementing them have been kept intact in the new projects, including their namespaces, so that errors are not introduced due to lack of expertise. Although these two transformations produce results that are different from the other transformations, the main differences are due to reordering of multi-valued references and we have considered them valid for this evaluation. On the other hand, a benchmark measurement harness considering the best practices recommended by the VIATRA team [13] was developed in order both to fine-tune measurements and to crosscheck results. This harness removes dependencies to other components of the VIATRA CPS benchmark so that experiments can be run locally.

In the present work, we aimed at answering the following research questions: *(RQ1)* Does *YAMTL-incr* show any performance penalty w.r.t. its execution in batch mode (*YAMTL-batch*)? *(RQ2)* Does *YAMTL-incr* show any improvement in performance w.r.t *EXPL* or *QRT* during initialization phase? *(RQ3)* And during propagation phase?

From the scenarios provided in the original benchmark, the scenarios *client-server* and *statistic based* [29] were considered. The CPS model generator [28] was used to obtain the input models to be used for the analysis so that their size depends on a logarithmic factor. The biggest models considered, in the client server scenario, consist of millions of nodes (10.16M) and edges (27.53M) and are, hence, VLMs.

For each tool and scenario, the experiments are run in isolation, i.e. in a separate Java process. For each of the input models, an initial experiment is performed to warm up the JVM and, then, twelve more experiments to measure performance. Each experiment consists of four phases: model load and engine initialization, initial transformation, delta propagation and model storage. In between each execution phase, the harness sends hints to the JVM to run garbage collection and waits for one second before proceeding on to the next phase. The first phase includes the instantiation of a fresh engine instance, avoiding interference between experiments as caches are not reused. The delta propagation phase includes the application of the delta to the source model and its propagation. Only initial transformation and delta propagation times have been considered in the quantitative analysis. For the results the median obtained for each of these two phases out of ten experiments is used, after removing the minimum and the maximum results.

In both solutions *EXPL* [25] and *QRT* [26], the delta is applied to the source model by directly modifying the resource containing the model. In the solution with YAMTL such delta was recorded and persisted using the EMF change model as described in Sect. 3.2. To analyze whether this feature could become a threat to validity, a separate experiment was run by excluding the query part of the model update (searching for the objects to be updated) in the solution *EXPL* but this change did not affect performance results perceptibly and the original solutions provided by the authors of the VIATRA CPS benchmark were considered. Therefore, the actions performed during the propagation phase are equivalent in all of the evaluated solutions.

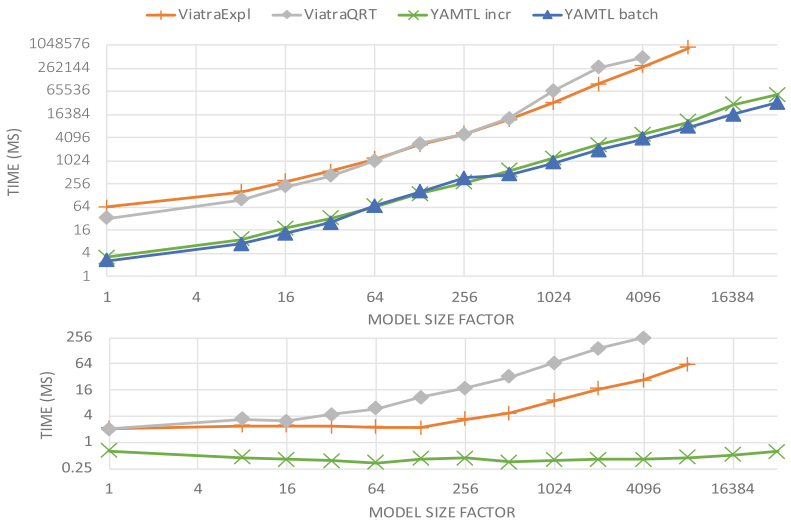


Fig. 3. Performance of initialization (top) and delta propagation (bottom).

Figure 3 shows the performance results obtained both for the initial model transformation and for forward delta propagation for the models generated for the client-server scenario. Scales both for time (ms.) along Y axis and for model size factors along X axis are logarithmic allowing us to compare the scalability of the different approaches. In the initialization phase, we have included the execution of YAMTL in batch mode (*YAMTL-batch*) over the source model, and it can be seen that tracking dependencies incurs a small penalty. However, the other two solutions (*EXPL* and *QRT*) operate several orders of magnitude slower. In the propagation phase, it can be observed that while *YAMTL-incr* exhibits a constant propagation time (in μs.) for the source delta, the cost of the other solutions depends on the size of the input model. Furthermore, for the other incremental approaches, when both initial and propagation time are combined their performance worsens due to their costly initialization phase.

5 Related Work

In this section, we discuss techniques used in related work for achieving incrementality in both reactive and bidirectional model transformation.

Reactive model transformation [3,21] enable the propagation of model updates from source models to target models on demand. State-of-the-art tool support relies on notification mechanisms, enabling live detection of source model updates either for immediate processing, as in VIATRA [3], or for deferred processing, as in ReactiveATL [21]. In these approaches, source model update notifications are usually fine-grained and kept in memory. Such notifications can only be detected when the transformation engine is in memory (live) as well. The use of a notification mechanism means that models are *loosely coupled* to the transformation engine. Working with offline model updates, as in the proposed delta propagation procedure, completely decouples detection of deltas from the transformation engine, freeing model update developers from the overhead of having the transformation infrastructure in memory. The latter is only needed for propagating changes but not for defining them. In reactive approaches, when an observer receives an update notification, information about the intent of the overall model delta, i.e. the contextual information relating different atomic updates, is lost. This problem is avoided using documented deltas, which may be serialized, enabling their processing – e.g. aggregating composite changes like the *move operation* – and optimization – reduction of atomic operations that are cancelled when composed. We refer the reader to [9] for an additional discussion of delta-based model updates against state-based model updates.

Among bidirectional model transformation approaches, Triple Graph Grammars (TGG), introduced in [22], are a declarative approach for specifying bidirectional consistency relations between models. Although our approach is not bidirectional, it is worth comparing how incrementality is supported in operational TGG rules. Incrementality was first introduced in TGG synchronization in [11,12]. Efficient approaches for TGG synchronization [18–20] avoid analyzing the whole model by relying on dependencies which hint at the impact of a model

update directly. Precedence-based approaches [18,20] keep a binary precedence relation over the set of model elements in order to determine when creation or deletion of a model element affects another one. While [18] overestimates the actual dependencies by defining them at the type level, others underestimate them relying on user feedback [20] or on special correspondences [12]. [19] decouples impact analysis of model updates from consistency restoration by delegating the former to VIATRA's incremental pattern matcher, which has a built-in dependency tracker, and by defining operational rules using a reactive model transformation approach. However, these two phases are still tightly coupled using a synchronous communication mechanism between the incremental pattern matcher and the synchronization procedure since the pattern matcher may trigger revocations/applications of forward marking rules after revoking/applying one of them. That is, the model synchronization procedure uses the pattern matcher to know when synchronization terminates. In the delta propagation mechanism proposed in the present work, either the revocation of applied transformation steps or the creation of new transformation steps cannot trigger further applications because rule matches are computed against the source model and they are unique, that is the same match cannot enable two different rules. A new transformation step may be found when new elements are inserted in the source model. On the other hand, when a transformation step is revoked, no other rule can be applied or a conflict would have been detected when the rule was applied the first time.

Some transformation engines with support for bidirectional transformations, like NMF [14,15], support the offline representation of model deltas. However, to the best of our knowledge, none of the aforementioned approaches uses a standardized notation for them, such as the EMF model change, which can be regarded as the de-facto standard for representing model deltas in the EMF modeling tool ecosystem.

6 Concluding Remarks

The main contribution of this work is the design of a delta propagation procedure for executing delta-driven model transformations, which has been implemented in YAMTL. The novelty of the approach consists in the use of a standardized representation of model deltas, which facilitates interoperability with EMF-compliant tools, and in the use of dependency injection mechanism, which allows the transformation engine to be aware of model updates without having to rely on a publish-subscribe infrastructure. The VIATRA CPS benchmark has been used to justify that *(1)* the initialization transformation in YAMTL is several orders of magnitude faster than the up-to-now fastest incremental solutions and that *(2)* propagation of sparse deltas can be performed in real time for VLMs, independently of their size, whereas other solutions show a clear dependence on their size. Hence, YAMTL shows satisfactory scalability in incremental execution of model transformations on VLMs. Additional studies with larger classes of models will be considered in future work.

References

1. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context — Motorola case study. In: Briand, L., Williams, C. (eds.) MODELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005). https://doi.org/10.1007/11557432_36
2. Benelallam, A., Gómez, A., Tisi, M., Cabot, J.: Distributing relational model transformation on mapreduce. J. Syst. Softw. **142**, 1–20 (2018)
3. Bergmann, G., et al.: VIATRA 3: a reactive model transformation platform. In: Kolovos, D., Wimmer, M. (eds.) ICMT 2015. LNCS, vol. 9152, pp. 101–110. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21155-8_8
4. Bergmann, G., Ráth, I., Varró, G., Varró, D.: Change-driven model transformations - change (in) the rule to rule the change. Softw. Syst. Model. **11**(3), 431–461 (2012)
5. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. Softw. Syst. Model. **11**(2), 227–250 (2012)
6. Boronat, A.: Expressive and efficient model transformation with an internal DSL of Xtend. In: MODELS 2018, pp. 78–88. ACM (2018)
7. Boronat, A.: YAMTL evaluation repository with the incremental component of the VIATRA CPS benchmark (2018). https://github.com/yamtl/viatra-cps-incr-benchmark
8. Daniel, G., Jouault, F., Sunyé, G., Cabot, J.: Gremlin-ATL: a scalable model transformation framework. In: ASE, pp. 462–472. IEEE Computer Society (2017)
9. 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
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
11. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MODELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006). https://doi.org/10.1007/11880240_38
12. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. **8**(1), 21–43 (2009)
13. Harmath, D., Ráth, I.: VIATRA/query/FAQ: performance optimization guidelines (2016). https://wiki.eclipse.org/VIATRA/Query/FAQ#Performance_optimization_guidelines
14. Hinkel, G.: Change propagation in an internal model transformation language. In: Kolovos, D., Wimmer, M. (eds.) ICMT 2015. LNCS, vol. 9152, pp. 3–17. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21155-8_1
15. Hinkel, G., Burger, E.: Change propagation and bidirectionality in internal transformation DSLs. Softw. Syst. Model. **18**(1), 249–278 (2017)
16. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: ICSE, pp. 471–480. ACM (2011)
17. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The grand challenge of scalability for model driven engineering. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 48–53. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01648-6_5

18. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient model synchronization with precedence triple graph grammars. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 401–415. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_27
19. Leblebici, E., Anjorin, A., Fritsche, L., Varró, G., Schürr, A.: Leveraging incremental pattern matching techniques for model synchronisation. In: de Lara, J., Plump, D. (eds.) ICGT 2017. LNCS, vol. 10373, pp. 179–195. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_11
20. Orejas, F., Pino, E.: Correctness of incremental model synchronization with triple graph grammars. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 74–90. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_6
21. Perez, S.M., Tisi, M., Douence, R.: Reactive model transformation with ATL. Sci. Comput. Program. **136**, 1–16 (2017)
22. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59071-4_45
23. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
24. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0., 2nd edn. Addison-Wesley Professional (2009)
25. VIATRA Team: Explicit traceability M2M transformation (2016). https://github.com/viatra/viatra-docs/blob/master/cps/Explicit-traceability-M2M-transformation.adoc
26. VIATRA Team: Query result traceability M2M transformation (2016). https://github.com/viatra/viatra-docs/blob/master/cps/Query-result-traceability-M2M-transformation.adoc
27. VIATRA Team: VIATRA CPS benchmark (cps to deployment transformation) (2016). https://github.com/viatra/viatra-docs/blob/master/cps/CPS-to-Deployment-Transformation.adoc
28. VIATRA Team: VIATRA CPS benchmark (model generator) (2016). https://github.com/viatra/viatra-docs/blob/master/cps/Model-Generator.adoc
29. VIATRA Team: VIATRA CPS benchmark (scenario specification) (2016). https://github.com/viatra/viatra-cps-benchmark/wiki/Benchmark-specification#cases

A Logic-Based Incremental Approach to Graph Repair

Sven Schneider[1(✉)], Leen Lambers[1], and Fernando Orejas[2]

[1] Hasso Plattner Institut, University of Potsdam, Potsdam, Germany
Sven.Schneider@HPI.de
[2] Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. Graph repair, restoring consistency of a graph, plays a prominent role in several areas of computer science and beyond: For example, in model-driven engineering, the abstract syntax of models is usually encoded using graphs. Flexible edit operations temporarily create inconsistent graphs not representing a valid model, thus requiring graph repair. Similarly, in graph databases—managing the storage and manipulation of graph data—updates may cause that a given database does not satisfy some integrity constraints, requiring also graph repair.

We present a logic-based incremental approach to graph repair, generating a sound and complete (upon termination) overview of least-changing repairs. In our context, we formalize consistency by so-called graph conditions being equivalent to first-order logic on graphs. We present two kind of repair algorithms: State-based repair restores consistency independent of the graph update history, whereas delta-based (or incremental) repair takes this history explicitly into account. Technically, our algorithms rely on an existing model generation algorithm for graph conditions implemented in AutoGraph. Moreover, the delta-based approach uses the new concept of satisfaction (ST) trees for encoding if and how a graph satisfies a graph condition. We then demonstrate how to manipulate these STs incrementally with respect to a graph update.

1 Introduction

Graph repair, restoring consistency of a graph, plays a prominent role in several areas of computer science and beyond. For example, in model-driven engineering, models are typically represented using graphs and the use of flexible edit operations may temporarily create inconsistent graphs not representing a valid model, thus requiring graph repair. This includes the situation where different views of an artifact are represented by a different model, i.e., the artifact is described by a multi-model, see, e.g. [6], and updates in some models may cause a global inconsistency in the multimodel. Similarly, in graph databases—managing the storage

F. Orejas has been supported by the Salvador de Madariaga grant PRX18/00308 and by funds from the Spanish Research Agency (AEI) and the European Union (FEDER funds) under grant GRAMM (ref. TIN2017-86727-C2-1-R).

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 151–167, 2019.
https://doi.org/10.1007/978-3-030-16722-6_9

and manipulation of graph data—updates may cause that a given database does not satisfy some integrity constraints [1], requiring also graph repair.

Numerous approaches on model inconsistency and repair (see [12] for an excellent recent survey) operate in varying frameworks with diverse assumptions. In our framework, we consider a typed directed graph (cf. [7]) to be inconsistent if it does not satisfy a given finite set of constraints, which are expressed by graph conditions [8], a formalism with the expressive power of first-order logic on graphs. A graph repair is, then, a description of an update that, if applied to the given graph, makes it consistent. Our algorithms do not just provide one repair, but a set of them from which the user must select the right repair to be applied. Moreover, we derive only least changing repairs, which do not include other smaller viable repairs. Our approach uses techniques (and the tool AutoGraph) [17] designed for model generation of graph conditions.

We consider two scenarios: In the first one, the aim is to repair a given graph (state-based repair). In the second one, a consistent graph is given together with an update that may make it inconsistent. In this case, the aim is to repair the graph in an incremental way (delta-based repair).

The main contributions of the paper are the following ones:

- A precise definition of what an update is, together with the definition of some properties, like e.g. least changing, that a repair update may satisfy.
- Two kind of graph repair algorithms: state-based and incremental (for the delta-based case). Moreover, we demonstrate for all algorithms *soundness* (the repair result provided by the algorithms is consistent) and *completeness* (upon termination, our algorithms will find all possible desired repairs)[1].

Summarizing, most repair techniques do not provide guarantees for the functional semantics of the repair and suffer from lack of information for the deployment of the techniques (see conclusion of the survey [12]). With our logic-based graph repair approach we aim at alleviating this weakness by presenting formally its functional semantics and describing the details of the underlying algorithms.

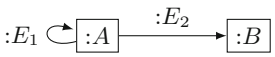
The paper is organized as follows: After introducing preliminaries in Sect. 2, we proceed in Sect. 3 with defining graph updates and repairs. In Sect. 4, we present the state-based scenario. We continue with introducing satisfaction trees in Sect. 5 that are needed for the delta-based scenario in Sect. 6. We close with a comparison with related work in Sect. 7 and conclusion with outlook in Sect. 8. For proofs of theorems and example details we refer to our technical report [18].

2 Preliminaries on Graph Conditions

We recall graph conditions (GCs), defined here over typed directed graphs, used for representing properties on such graphs. In our running example[2], we employ

[1] Note that completeness implies totality (if the given set of constraints is satisfiable by a finite graph, then the algorithms will find a repair for any inconsistent graph).

[2] We refer to Sect. 1 with pointers to related work including diverse use cases in Software Engineering for graph repair with more complex and motivating examples.



$$\neg\exists(a, \neg(\exists(a \xrightarrow{e} b, true) \wedge \neg\exists(a \circlearrowright e, true)))$$

Fig. 1. The type graph TG (left) and the GC ψ (right) for our running example

the type graph TG from Fig. 1 and we use nodes with names a_i and b_i to indicate that they are of type $:A$ and $:B$, respectively.

GCs state facts about the existence of graph patterns in a given graph, called a host graph. For example, in the syntax used in our running example, the GC $\exists(a, true)$ means that the host graph must include a node of type $:A$. Also, $\exists(a \longrightarrow b, true)$ means that the host graph must include a node of type $:A$, another node of type $:B$, and an edge from the $:A$-node to the $:B$-node.

In general, in the syntax that we use in our running example, an atomic GC is of the form $\exists(H, \phi)$ (or $\neg\exists(H, \phi)$) where H is a graph that must be (or must not be) included in the host graph and where ϕ is a condition expressing more restrictions on how this graph is found (or not found) in the host graph. For instance, $\exists(a, \neg\exists(a \xrightarrow{e} b, true))$ states that the host graph must include an $:A$-node such that it has no outgoing edge e to a $:B$-node. Moreover, we use the standard boolean operators to combine atomic GCs to form more complex ones. For instance, $\exists(a, \neg(\exists(a \xrightarrow{e} b, true) \wedge \neg\exists(a \circlearrowright e, true)))$ states that the host graph must include an $:A$-node, such that it does not hold that there is an outgoing edge e to a $:B$-node and node a has no loop. In addition, as an abbreviation for readability, we may use the universal quantifier with the meaning $\forall(H, \phi) = \neg\exists(H, \neg\phi)$. In this sense, the condition ϕ from Fig. 1, used in our running example, states that every node of type $:A$ must have an outgoing edge to a node of type $:B$ and that such an $:A$-node must have no loop.

Formally, the syntax of GCs [8], expressively equivalent to first-order logic on graphs [5], is given subsequently. This logic encodes properties of graph extensions, which must be explicitly mentioned as graph inclusions. For instance, the GC $\exists(a, \neg\exists(a \xrightarrow{e} b, true))$ in simplified notation is formally given in the syntax of GCs as $\exists(\mathrm{i}_H, \neg\exists(a \hookrightarrow (a \xrightarrow{e} b), true))$, where i_H denotes the inclusion $\emptyset \hookrightarrow H$ with H the graph consisting of node a. This is because it expresses a property of the extension i_H. Moreover, therein the GC $\neg\exists(a \hookrightarrow (a \xrightarrow{e} b), true)$ is actually a property of the extension $a \hookrightarrow (a \xrightarrow{e} b)$.

Definition 1 (Graph Conditions (GCs) [8]). *The class of* graph conditions Φ_H^{GC} *for the graph* H *is defined inductively:*

- $\wedge S \in \Phi_H^{\mathrm{GC}}$ *if* $S \subseteq_{\mathrm{fin}} \Phi_H^{\mathrm{GC}}$.
- $\neg\phi \in \Phi_H^{\mathrm{GC}}$ *if* $\phi \in \Phi_H^{\mathrm{GC}}$.
- $\exists(a : H \hookrightarrow H', \phi) \in \Phi_H^{\mathrm{GC}}$ *if* $\phi \in \Phi_{H'}^{\mathrm{GC}}$.

In addition $true$, $false$, $\vee S$, $\phi_1 \Rightarrow \phi_2$, *and* $\forall(a, \phi)$ *can be used as abbreviations, with their obvious replacement.*

A mono $m : H \hookrightarrow G$ satisfies *a* GC $\psi \in \Phi_H^{\mathrm{GC}}$, *written* $m \models_{\mathrm{GC}} \psi$, *if one of the following cases applies.*

- $\psi = \wedge S$ *and* $m \models_{\mathrm{GC}} \phi$ *for each* $\phi \in S$.
- $\psi = \neg\phi$ *and not* $m \models_{\mathrm{GC}} \phi$.
- $\psi = \exists(a : H \hookrightarrow H', \phi)$ *and* $\exists q : H' \hookrightarrow G.\ q \circ a = m \wedge q \models_{\mathrm{GC}} \phi$.

A graph G *satisfies a* GC $\psi \in \Phi_{\emptyset}^{\mathrm{GC}}$*, written* $G \models_{\mathrm{GC}} \psi$ *or* $G \in [\![\psi]\!]$*, if* $\mathrm{i}_G \models_{\mathrm{GC}} \psi$.

3 Graph Updates and Repairs

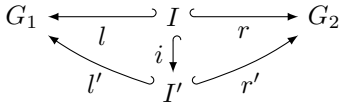
In this section, we define graph updates to formalize arbitrary modifications of graphs, graph repairs as the desired graph updates resulting in repaired graphs, as well as further desireable properties of graph updates.

In particular, it is well known that a modification or update of G_1 resulting in a graph G_2 can be represented by two inclusions or, in general two monos, which we denote by $(l : I \hookrightarrow G_1, r : I \hookrightarrow G_2)$, where I represents the part of G_1 that is preserved by this update. Intuitively, $l : I \hookrightarrow G_1$ describes the deletion of elements from G_1 (i.e., all elements in $G_1 \setminus l(I)$ are deleted) and $r : I \hookrightarrow G_2$ describes the addition of elements to I to obtain G_2 (i.e., all elements in $G_2 \setminus r(I)$ are added).

Definition 2 (Graph Update). *A* (graph) update u *is a pair* $(l : I \hookrightarrow G_1, r : I \hookrightarrow G_2)$ *of monos. The class of all updates is denoted by* $\mathcal{U}$.

Graph updates such as $(\mathrm{i}_G : \emptyset \hookrightarrow G, \mathrm{i}_G : \emptyset \hookrightarrow G)$ where G is not the empty graph delete all the elements in G that are added by r afterwards. To rule out such updates, we define an update $(l : I \hookrightarrow G_1, r : I \hookrightarrow G_2)$ to be *canonical* when the graph I is as large as possible, i.e. intuitively $I = G_1 \cap G_2$. Formally:

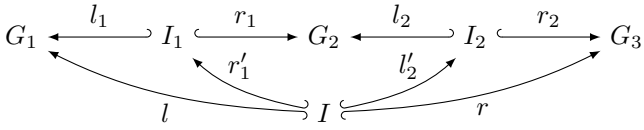
Definition 3 (Canonical Graph Update). *If* $(l : I \hookrightarrow G_1, r : I \hookrightarrow G_2) \in \mathcal{U}$ *and every* $(l' : I' \hookrightarrow G_1, r' : I' \hookrightarrow G_2) \in \mathcal{U}$ *and mono* $i : I \hookrightarrow I'$ *with* $l' \circ i = l$ *and* $r' \circ i = r$ *satisfies that* i *is an isomorphism then* (l, r) *is* canonical*, written* $(l, r) \in \mathcal{U}_{\mathrm{can}}$.



An update u_1 is a sub-update (see [14]) of u whenever the modifications defined by u_1 are fully contained in the modifications defined by u. Intuitively, this is the case when u_1 can be composed with another update u_2 such that (a) the resulting update has the same effect as u and (b) u_2 does not delete any element that was added before by u_1. This is stated, informally speaking, by requiring that I is the intersection (pullback) of I_1 and I_2 and that G_2 is its union (pushout).

Definition 4 (Sub-update [14]). *If* $u = (l : I \hookrightarrow G_1, r : I \hookrightarrow G_2) \in \mathcal{U}$, $u_1 = (l_1 : I_1 \hookrightarrow G_1, r_1 : I_1 \hookrightarrow G_3) \in \mathcal{U}$, $u_2 = (l_2 : I_2 \hookrightarrow G_3, r_2 : I_2 \hookrightarrow G_2) \in \mathcal{U}$,

$(r_1' : I \hookrightarrow I_1, l_2' : I \hookrightarrow I_2)$ is the pullback of (r_1, l_2), and (r_1, l_2) is the pushout of (r_1', l_2') then u_1 is a sub-update *of u, written $u_1 \leq^{u_2} u$ or simply $u_1 \leq u$.*



Moreover, we write $u_1 <^{u_2} u$ or $u_1 < u$ when $u_1 \leq^{u_2} u$ and not $u \leq u_1$.

We now define graph repairs as graph updates where the result graph satisfies the given consistency constraint ψ.

Definition 5 (Graph Repair). *If $u = (l : I \hookrightarrow G_1, r : I \hookrightarrow G_2) \in \mathcal{U}$, $\psi \in \Phi_\emptyset^{\mathrm{GC}}$, and $G_2 \models_{\mathrm{GC}} \psi$ then u is a* graph repair *or simply* repair *of G_1 with respect to ψ, written $u \in \mathcal{U}(G_1, \psi)$.*

To define a finite set of desirable repairs, we introduce the notion of least changing repairs that are repairs for which no sub-updates exist that are also repairs.

Definition 6 (Least Changing Graph Repair). *If $\psi \in \Phi_\emptyset^{\mathrm{GC}}$, $u = (l : I \hookrightarrow G_1, r : I \hookrightarrow G_2) \in \mathcal{U}(G_1, \psi)$, and there is no $u' \in \mathcal{U}(G_1, \psi)$ such that $u' < u$ then u is a* least changing graph repair *of G_1 with respect to ψ, written $u \in \mathcal{U}_{\mathrm{lc}}(G_1, \psi)$.*

Note that every least changing repair is canonical according to this definition. Moreover, the notion of least changing repairs is unrelated to other notions of repairs such as the set of all repairs that require a smallest amount of atomic modifications of the graph at hand to result in a graph satisfying the consistency constraint. For instance, a repair u_1 adding two nodes of type $:A$ may be a least changing repair even if there is a repair u_2 adding only one node of type $:B$.

A graph repair algorithm is *stable* [12], if the repair procedure returns the identity update $(\mathrm{id}_G : G \hookrightarrow G, \mathrm{id}_G : G \hookrightarrow G)$ when graph G is already consistent. Obviously, a graph repair algorithm that only returns least changing repairs is stable, since the identity update is a sub-update of any other repair.

4 State-Based Repair

In this section, we introduce two state-based graph repair algorithms (see [18] for additional technical detail), which compute a set of graph repairs restoring consistency for a given graph.

Definition 7 (State-Based Graph Repair Algorithm). *A state-based graph repair algorithm takes a graph G and a* GC *$\psi \in \Phi_\emptyset^{\mathrm{GC}}$ as inputs and returns a set of graph repairs in $\mathcal{U}(G, \psi)$.*

Note that the tool AutoGraph [17] can be used to verify this condition as follows: It determines the operation $\mathcal{A}$ that constructs a finite set of all minimal graphs satisfying a given GC ψ. Formally, $\mathcal{A}(\psi) = \cap\{S \subseteq [\![\psi]\!] \mid \forall G' \in [\![\psi]\!]. \exists G \in$

$S.\, \exists m : G \hookrightarrow G'.true\}$. While AutoGraph may not terminate when computing this operation due to the inherent expressiveness of GCs, it is known that AutoGraph terminates whenever ψ is not satisfied by any graph.

The state-based algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ uses $\mathcal{A}$ to obtain repairs. $\mathcal{R}\text{epair}_{\text{sb},1}$ computes the set $\mathcal{A}(\psi \wedge \exists(\text{i}_G, true))$ that contains all minimal graphs that (a) satisfy ψ and (b) include a copy of G. All these extensions of G correspond to a graph repair. For our running example, we do not obtain any repair for graph $\mathbf{G'_u}$ from Fig. 2 and GC $\boldsymbol{\psi}$ from Fig. 1 because the loop on node a_2 would invalidate any graph including $\mathbf{G'_u}$. We state that $\mathcal{R}\text{epair}_{\text{sb},1}$ indeed computes the non-deleting least changing graph repairs.

Theorem 1 (Functional Semantics of $\mathcal{R}\text{epair}_{\text{sb},1}$). *$\mathcal{R}\text{epair}_{\text{sb},1}$ is* sound*, i.e., $\mathcal{R}\text{epair}_{\text{sb},1}(G, \psi) \subseteq \mathcal{U}_{\text{lc}}(G, \psi)$, and* complete (upon termination) *with respect to non-deleting repairs in $\mathcal{U}_{\text{lc}}(G, \psi)$.*

The second state-based algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ computes *all* least changing graph repairs. In this algorithm we use the approach of $\mathcal{R}\text{epair}_{\text{sb},1}$ but compute $\mathcal{A}(\psi \wedge \exists(\text{i}_{G_c}, true))$ whenever an inclusion $l : G_c \hookrightarrow G$ describes how G can be restricted to one of its subgraphs G_c. Every graph G' obtained from the application of $\mathcal{A}$ for one of these graphs G_c then results in one graph repair returned by $\mathcal{R}\text{epair}_{\text{sb},2}$ except for those that are not least changing.

To this extent we introduce the notion of a restriction tree (see example in Fig. 2) having all subgraphs G_c of a given graph G as nodes as long as they include the graph G_{min}, which is the empty graph in the state-based algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ but not in the algorithm $\mathcal{R}\text{epair}_{\text{db}}$ in Sect. 6, and where edges are given in this tree by inclusions that add precisely one node or edge.

Definition 8 (Restriction Tree RT). *If G and G_{min} are graphs and $S = \{l : G_c \hookrightarrow G_p \mid G_{min} \subseteq G_c \subset G_p \subseteq G, l$ is an inclusion$\}$, S' is the least subset of S such that the closure of S' under $\circ$ equals S then a* restriction tree $\text{RT}(G, G_{min})$ *is a least subset of S' such that for all two inclusions $l_1 : G \hookrightarrow G_1 \in S'$ and $l_2 : G \hookrightarrow G_2 \in S'$ one of them is in $\text{RT}(G, G_{min})$.*

Considering our running example, the restriction tree in Fig. 2 is traversed entirely except for the four graphs without a border, which are not traversed as they have the supergraph marked 9 satisfying $\boldsymbol{\psi}$ and therefore traversing those would generate repairs that are not least changing. The resulting graph repairs for the condition $\boldsymbol{\psi}$ are given by the graphs marked by 3–6.

Our second state-based graph repair algorithm is indeed sound and complete whenever the calls to AutoGraph using $\mathcal{A}$ terminate.

Theorem 2 (Functional Semantics of $\mathcal{R}\text{epair}_{\text{sb},2}$). *$\mathcal{R}\text{epair}_{\text{sb},2}$ is* sound*, i.e., $\mathcal{R}\text{epair}_{\text{sb},2}(G, \psi) \subseteq \mathcal{U}_{\text{lc}}(G, \psi)$, and* complete*, i.e., $\mathcal{U}_{\text{lc}}(G, \psi) \subseteq \mathcal{R}\text{epair}_{\text{sb},2}(G, \psi)$, upon termination.*

5 Satisfaction Trees

The state-based algorithms introduced in the previous section are inefficient when used in a scenario where a graph needs repair after a sequence of updates

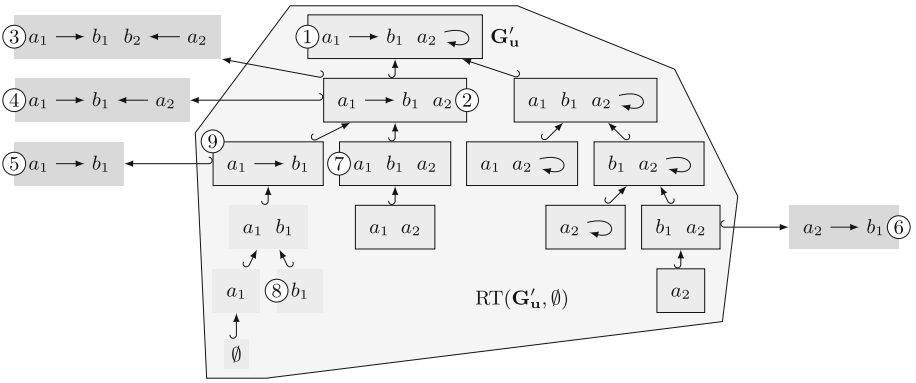


Fig. 2. The restriction tree $\mathrm{RT}(\mathbf{G'_u}, \emptyset)$ (enclosed by the polygon) and four graph repairs (marked 3–6) generated using $\mathcal{R}\mathrm{epair}_{\mathrm{sb},2}$

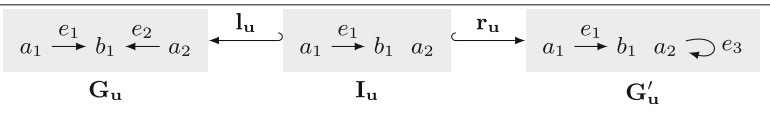
that all need repair. We thus present in Sect. 6 an incremental algorithm reducing the computational cost for a repair when an update is provided. This algorithm uses an additional data structure, called *satisfaction tree* or ST, which stores information on if and how a graph G satisfies a GC ψ (according to Definition 1). In this section, given ψ and G, we define how such an ST γ is constructed and how it is updated once the graph G is updated.

If ψ is a conjunction of conditions, its associated ST γ is a conjunction of STs and if ψ is a negation of a conditions, its associated γ is a negation of an ST. In the case when ψ is a $\exists(a : H \hookrightarrow H', \phi)$, recall that a match $m : H \hookrightarrow G$ satisfies ψ if there exists a $q : H' \hookrightarrow G$ such that $m = q \circ a$ and $q \models_{\mathrm{GC}} \phi$. For this case, we keep in ST each q satisfying these two conditions and also each q that satisfies the first condition, but not the second. More precisely, for the case of existential quantification, the corresponding ST is of the form $\exists(a : H \hookrightarrow H', \phi, m_t, m_f)$, where m_t and m_f are partial mappings (we use $\sup(f)$ to denoted the elements actually mapped by a partial map f) that map matches $q : H' \hookrightarrow G$ that satisfy $m = q \circ a$ (for a previously known $m : H \hookrightarrow G$) to an ST for the subcondition ϕ. The difference between both partial functions is that m_t maps matches q to STs for which $q \models_{\mathrm{GC}} \phi$ while m_f maps matches q to STs for which $q \not\models_{\mathrm{GC}} \phi$. Consider Fig. 3b for an example of an ST $\gamma_{\mathbf{u}}$.

The following definition describes the syntax of STs. The STs are defined over matches into a graph G to allow for the basic well-formedness condition that every mapped match q satisfies $q \circ a = m$.

Definition 9 (Satisfaction Trees (STs)). *The class of all* Satisfaction Trees Γ_m^{ST} *for a mono* $m : H \hookrightarrow G$ *contains* γ *if one of the following cases applies.*

- $\gamma = \wedge S$ *and* $S \subseteq_{\mathrm{fin}} \Gamma_m^{\mathrm{ST}}$.
- $\gamma = \neg\chi$ *and* $\chi \in \Gamma_m^{\mathrm{ST}}$.
- $\gamma = \exists(a, \phi, m_t, m_f)$, $a : H \hookrightarrow H'$, $\phi \in \Phi_{H'}^{\mathrm{GC}}$, $m_t, m_f \subseteq_{\mathrm{fin}} \{(q : H' \hookrightarrow G, \bar{\gamma}) \mid q \circ a = m, \bar{\gamma} \in \Gamma_q^{\mathrm{ST}}\}$, *and* m_t, m_f *are partial maps.*



(a) A graph update $\mathbf{u} = (\mathbf{l_u} : \mathbf{I_u} \hookrightarrow \mathbf{G_u}, \mathbf{r_u} : \mathbf{I_u} \hookrightarrow \mathbf{G'_u})$

$$\gamma_{\mathbf{u}} = \neg\exists(a, \underbrace{\neg(\exists(a \xrightarrow{e} b, true) \wedge \neg\exists(a \circlearrowright e, true))}, \emptyset, \{a_2 \mapsto \gamma_{\mathbf{u},\mathbf{1}}, a_1 \mapsto \gamma_{\mathbf{u},\mathbf{2}}\})$$
$$\gamma_{\mathbf{u},\mathbf{1}} = \neg(\exists(a \xrightarrow{e} b, \underline{true}, \{a_2 \xrightarrow{e_2} b_1 \mapsto true\}, \emptyset) \wedge \neg\exists(a \circlearrowright e, \underline{true}, \emptyset, \emptyset))$$
$$\gamma_{\mathbf{u},\mathbf{2}} = \neg(\exists(a \xrightarrow{e} b, \underline{true}, \{a_1 \xrightarrow{e_1} b_1 \mapsto true\}, \emptyset) \wedge \neg\exists(a \circlearrowright e, \underline{true}, \emptyset, \emptyset))$$

(b) The ST $\gamma_{\mathbf{u}}$ for $\mathbf{G_u}$ (see Fig. 3a) and $\boldsymbol{\psi}$ (see Fig. 1).

$$\gamma^{\mathbf{I}}_{\mathbf{u}} = \neg\exists(a, \underbrace{\neg(\exists(a \xrightarrow{e} b, true) \wedge \neg\exists(a \circlearrowright e, true))}, \{a_2 \mapsto \gamma^{\mathbf{I}}_{\mathbf{u},\mathbf{1}}\}, \{a_1 \mapsto \gamma^{\mathbf{I}}_{\mathbf{u},\mathbf{2}}\})$$
$$\gamma^{\mathbf{I}}_{\mathbf{u},\mathbf{1}} = \neg(\exists(a \xrightarrow{e} b, \underline{true}, \emptyset, \emptyset) \wedge \neg\exists(a \circlearrowright e, \underline{true}, \{a_2 \circlearrowright e_3 \mapsto true\}, \emptyset))$$
$$\gamma^{\mathbf{I}}_{\mathbf{u},\mathbf{2}} = \neg(\exists(a \xrightarrow{e} b, \underline{true}, \{a_1 \xrightarrow{e_1} b_1 \mapsto true\}, \emptyset) \wedge \neg\exists(a \circlearrowright e, \underline{true}, \emptyset, \emptyset))$$

(c) The ST $\gamma^{\mathbf{I}}_{\mathbf{u}}$ for $\mathbf{I_u}$ (see Fig. 3a) and $\boldsymbol{\psi}$ (see Fig. 1) that is obtained as the backward propagation $\mathrm{ppgB}(\gamma_{\mathbf{u}}, \mathbf{l_u})$ from $\gamma_{\mathbf{u}}$ (see Fig. 3b) and $\mathbf{l_u}$ (see Fig. 3a)

$$\gamma'_{\mathbf{u}} = \neg\exists(a, \underbrace{\neg(\exists(a \xrightarrow{e} b, true) \wedge \neg\exists(a \circlearrowright e, true))}, \{a_2 \overset{(\mathsf{R1})}{\mapsto} \gamma'_{\mathbf{u},\mathbf{1}}\}, \{a_1 \mapsto \gamma'_{\mathbf{u},\mathbf{2}}\})$$
$$\gamma'_{\mathbf{u},\mathbf{1}} = \neg(\exists(a \xrightarrow{e} b, \underline{true}, \emptyset_{(\mathsf{R2})}, \emptyset) \wedge \neg\exists(a \circlearrowright e, \underline{true}, \{a_2 \circlearrowright e_3 \overset{(\mathsf{R3})}{\mapsto} true\}, \emptyset))$$
$$\gamma'_{\mathbf{u},\mathbf{2}} = \neg(\exists(a \xrightarrow{e} b, \underline{true}, \{a_1 \xrightarrow{e_1} b_1 \mapsto true\}, \emptyset) \wedge \neg\exists(a \circlearrowright e, \underline{true}, \emptyset, \emptyset))$$

(d) The ST $\gamma'_{\mathbf{u}}$ for $\mathbf{G'_u}$ (see Fig. 3a) and $\boldsymbol{\psi}$ (see Fig. 1) that is obtained as the forward propagation $\mathrm{ppgF}(\gamma^{\mathbf{I}}_{\mathbf{u}}, \mathbf{r_u})$ from $\gamma^{\mathbf{I}}_{\mathbf{u}}$ (see Fig. 3b) and $\mathbf{r_u}$ (see Fig. 3a). Also $\gamma'_{\mathbf{u}}$ is the result of $\mathrm{ppgU}(\gamma_{\mathbf{u}}, \mathbf{u})$ that applies backward and forward propagation. The viable points for the delta-based repair discussed in Sec. 6 are indicated by (R1)–(R3).

Fig. 3. A graph update and an ST with its propagation over the graph update where GCs are underlined in STs for readability

The following satisfaction predicate $\models_{\mathrm{GC}}$ for STs defines when an ST γ for a mono m states that the contained GC ψ is satisfied by the morphism m.

Definition 10 (ST Satisfaction). *An* ST $\gamma \in \Gamma^{\mathrm{ST}}_{m:H \hookrightarrow G}$ *is* satisfied*, written* $\models_{\mathrm{ST}} \gamma$*, if one of the following cases applies.*

- $\gamma = \wedge S$ *and* $\models_{\mathrm{ST}} \chi$ *(for each* $\chi \in S$*)*
- $\gamma = \neg\chi$ *and* $\not\models_{\mathrm{ST}} \chi$*.*
- $\gamma = \exists(a, \phi, m_t, m_f)$ *and* $m_t \neq \emptyset$*.*

The following recursive operation constructs an ST γ for a graph G and a condition ψ so that γ represents how G satisfies (or not satisfies) ψ. Note that the match m in the definition of STs above and the construction of an ST below

corresponds to the match $m : H \hookrightarrow G$ from Definition 1 that we operationalize in the following definition. For conjunction and negation, we construct the STs from the STs for the subconditions. For the case of existential quantification, we consider all morphisms $q : H' \hookrightarrow G$ for which the triangle $q \circ a = m$ commutes and construct the STs for the subcondition ϕ under this extended match q. The resulting STs are inserted into m_t and m_f according to whether they are satisfied.

Definition 11 (Construct ST *(*cst*)).* *Given* $m : H \hookrightarrow G$ *and* $\psi \in \Phi_H^{\mathrm{GC}}$*, we define* $\mathrm{cst}(\psi, m) = \gamma$*, with* $\gamma \in \Gamma_m^{\mathrm{ST}}$ *as follows.*

- *If* $\psi = \wedge S$ *then* $\gamma = \wedge\{\mathrm{cst}(\phi, m) \mid \phi \in S\}$.
- *If* $\psi = \neg\phi$ *then* $\gamma = \neg\,\mathrm{cst}(\phi, m)$.
- *If* $\psi = \exists(a : H \hookrightarrow H', \phi)$*,* $m_{all} = \{(q : H' \hookrightarrow G, \chi) \mid q \circ a = m, \mathrm{cst}(\phi, q) = \chi\}$*,* $m_t = \{(q, \chi) \in m_{all} \mid\models_{\mathrm{ST}} \chi\}$*,* $m_f = m_{all} \setminus m_t$*, then* $\gamma = \exists(a, \phi, m_t, m_f)$.

If G *is a graph and* $\psi \in \Phi_\emptyset^{\mathrm{GC}}$*, then* $\mathrm{cst}(\psi, G) = \mathrm{cst}(\psi, \mathrm{i}_G)$.

This construction of STs then ensures that $\models_{\mathrm{ST}} \gamma$ if and only if $G \models_{\mathrm{GC}} \psi$. Note that $\models_{\mathrm{ST}} \gamma_{\mathbf{u}}$ holds for the ST $\gamma_{\mathbf{u}}$ from Fig. 3b, the GC $\boldsymbol{\psi}$ from Fig. 1, and the graph $\mathbf{G_u}$ from Fig. 3.

Theorem 3 (Sound Construction of STs). *Given* $m : H \hookrightarrow G$*,* $\psi \in \Phi_H^{\mathrm{GC}}$*, and* $\mathrm{cst}(\psi, m) = \gamma$ *then* $\models_{\mathrm{ST}} \gamma$ *iff* $m \models_{\mathrm{GC}} \psi$.

Subsequently, we define a propagation operation ppgU of an ST γ for a graph update $u = (l : I \hookrightarrow G, r : I \hookrightarrow G')$ to obtain an ST γ' such that $\gamma' = \mathrm{cst}(\psi, G')$ whenever $\gamma = \mathrm{cst}(\psi, G)$. This overall propagation is performed by a backward propagation of γ for l using the operation ppgB followed by a forward propagation of the resulting ST for r using the operation ppgF.

For backward propagation, we describe how the deletion of elements in G by $l : I \hookrightarrow G$ affect its associated ST γ. To this end, we preserve those matches $q : H \hookrightarrow G$ for which no matched elements are deleted. This is formalized by requiring a mono $q' : H \hookrightarrow I$ such that $l \circ q' = q$. The matches q with deleted matched elements can not be preserved and are therefore removed.

Definition 12 (Propagate Match *(*ppgMatch*)).* *If* $q : H \hookrightarrow G$ *and* $l : I \hookrightarrow G$ *are monos, then* $\mathrm{ppgMatch}(q, l)$ *is the unique* $q' : H \hookrightarrow I$ *such that* $l \circ q' = q$ *if it exists and* $\perp$ *otherwise.*

The following recursive backward propagation defines how deletions affect the maps m_t and m_f of the given ST. That is, when $\gamma = \exists(a, \phi, m_t, m_f)$, we (a) entirely remove a mapping (m, χ) from m_t or m_f if $\mathrm{ppgMatch}(q, l) = \perp$ and (b) construct for a mapping (m, χ) from m_t or m_f the pair $(\mathrm{ppgMatch}(q, l), \chi')$ where χ' is obtained from recursively applying the backward propagation on χ when $\mathrm{ppgMatch}(q, l) \neq \perp$. The updated pair $(\mathrm{ppgMatch}(q, l), \chi')$ must be rechecked to decide to which partial map this pair must be added to ensure that the resulting ST corresponds to the ST that would be constructed for G' directly.

Definition 13 (Backward Propagation (ppgB)**).** *If* $m : H \hookrightarrow G$, $\gamma \in \Gamma^{\mathrm{ST}}_{m}$, $l : I \hookrightarrow G$, $\mathrm{ppgMatch}(m, l) = m' : H \hookrightarrow I$, *and* $\gamma' \in \Gamma^{\mathrm{ST}}_{m'}$ *then* $\mathrm{ppgB}(\gamma, l) = \gamma'$ *if one of the following cases applies.*

- $\gamma = \wedge S$ *and* $\gamma' = \wedge\{\mathrm{ppgB}(\chi, l) \mid \chi \in S\}$.
- $\gamma = \neg\chi$ *and* $\gamma' = \neg\,\mathrm{ppgB}(\chi, l)$.
- $\gamma = \exists(a, \phi, m_t, m_f)$, $m_{all} = \{(q', \chi') \mid (q, \chi) \in m_t \cup m_f \wedge \mathrm{ppgMatch}(q, l) = q' \neq \bot \wedge \mathrm{ppgB}(\chi, l) = \chi'\}$, $m'_t = \{(q, \chi) \in m_{all} \mid\!\models_{\mathrm{ST}} \chi\}$, $m'_f = m_{all} \setminus m'_t$, *and* $\gamma' = \exists(a, \phi, m'_t, m'_f)$.

Note that $\mathrm{ppgMatch}(\mathrm{i}_G, l) = \mathrm{i}_G$ and, hence, the operation ppgB is applicable for all ST $\gamma \in \Gamma^{\mathrm{ST}}_{\mathrm{i}_G}$, which is sufficient as we define consistency constraints using GCs over the empty graph as well.

In the case of forward propagation where additions are given by $r : I \hookrightarrow G'$ we can preserve all matches using an adaptation. But the addition of further elements may result in additional matches as well that may satisfy the conditions to be included in the corresponding m_t and m_f from the ST at hand.

Definition 14 (Forward Propagation (ppgF)**).** *If* $\gamma \in \Gamma^{\mathrm{ST}}_{m:H \hookrightarrow I}$, $r : I \hookrightarrow G'$, *and* $\gamma' \in \Gamma^{\mathrm{ST}}_{r \circ m}$ *then* $\mathrm{ppgF}(\gamma, r) = \gamma'$ *if one of the following cases applies.*

- $\gamma = \wedge S$ *and* $\gamma' = \wedge\{\mathrm{ppgF}(\chi, r) \mid \chi \in S\}$.
- $\gamma = \neg\chi$ *and* $\gamma' = \neg\,\mathrm{ppgF}(\chi, r)$.
- $\gamma = \exists(a, \phi, m_t, m_f)$, $m_{all} = \{(r \circ q, \gamma') \mid (q, \chi) \in m_t \cup m_f \wedge \mathrm{ppgF}(\chi, r) = \gamma'\} \cup \{(q, \gamma_q) \mid q \circ a = r \circ m, (\nexists q' \in \mathrm{sup}(m_t) \cup \mathrm{sup}(m_f).\ r \circ q' = q), \mathrm{cst}(\phi, q) = \gamma_q\}$, $m'_t = \{(q, \chi) \in m_{all} \mid\!\models_{\mathrm{ST}} \chi\}$, $m'_f = m_{all} \setminus m'_t$, *and* $\gamma' = \exists(a, \phi, m'_t, m'_f)$.

We now define the composition of both propagations to obtain the operation ppgU that updates an ST for an entire graph update.

Definition 15 (Update Propagation (ppgU)**).** *If* $m : H \hookrightarrow G$, $\gamma \in \Gamma^{\mathrm{ST}}_{m}$, $l : I \hookrightarrow G$, $\mathrm{ppgMatch}(m, l) = m' : H \hookrightarrow G'$, *and* $r : I \hookrightarrow G'$ *then* $\mathrm{ppgU}(\gamma, (l, r)) = \mathrm{ppgF}(\mathrm{ppgB}(\gamma, l), r) \in \Gamma^{\mathrm{ST}}_{m'}$.

The overall propagation given by this operation is *incremental*, in the sense that the operation cst is only used in the forward propagation on parts of the graph G', where the addition of graph elements by r from the graph update results in additional matches q according to the satisfaction relation for GCs. Finally, we state that ppgU incrementally computes the ST obtained using cst. The proof of this theorem relies on the fact that this property also holds for ppgB and ppgF.

Theorem 4 (ppgU **is Compatible with** cst**).** *If* G *is a graph,* $\psi \in \Phi^{\mathrm{GC}}_{\emptyset}$, $l : I \hookrightarrow G$, *and* $r : I \hookrightarrow G'$ *then* $\mathrm{ppgU}(\mathrm{cst}(\psi, G), (l, r)) = \mathrm{cst}(\psi, G')$.

6 Delta-Based Repair

The local states of delta-based graph repair algorithms may contain, besides the current graph as in state-based graph repair algorithms, an additional value. In our delta-based graph repair algorithm this will be an ST.

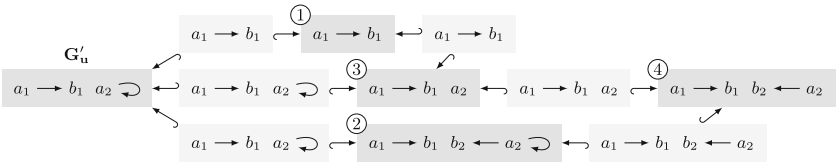


Fig. 4. An example for delta-based graph repair using $\mathcal{R}\text{epair}_{\text{db}}$

Definition 16 (Delta-Based Graph Repair Algorithm). *Delta-based graph repair algorithms take a graph G, a* GC $\psi \in \Phi_{\emptyset}^{\text{GC}}$*, and a value q as inputs and return a set of pairs (u, q') where $u \in \mathcal{U}(G, \psi)$ is a graph repair and q' is a value.*

Our delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ will be based on the single step operation $\mathcal{R}\text{epair}_{\text{db1}}$. Given a graph G, a GC $\psi \in \Phi_{\emptyset}^{\text{GC}}$, the ST γ that equals $\text{cst}(\psi, G)$, and a graph update $u = (l : I \hookrightarrow G, r : I \hookrightarrow G')$, the single step operation $\mathcal{R}\text{epair}_{\text{db}}$ first updates γ using ppgU for the graph update u and then determines using $\mathcal{R}\text{epair}_{\text{db1}}$, if necessary, graph repairs for the resulting ST γ' according to the repair rules described in the following. The algorithm $\mathcal{R}\text{epair}_{\text{db}}$ then uses $\mathcal{R}\text{epair}_{\text{db1}}$ in a breadth first manner to obtain multi-step repairs.

For our example from Fig. 3a, such a multi-step repair of $\mathbf{G'_u}$ is given in Fig. 4 where the graph updates are obtained resulting in the graphs marked 1–3, of which only the graph marked 1 satisfies $\boldsymbol{\psi}$. The algorithm $\mathcal{R}\text{epair}_{\text{db}}$ then computes further graph updates resulting in the graph marked 4 also satisfying $\boldsymbol{\psi}$.

The operation $\mathcal{R}\text{epair}_{\text{db1}}$ for deriving single-step repairs depends on two local modifications. Firstly, a GC $\exists(a : H \hookrightarrow H', \phi)$ occurring as a subcondition in the consistency constraint ψ may be violated because, for the match $m : H \hookrightarrow G$ that locates a copy of H in the graph G under repair, no suitable match $q : H' \hookrightarrow G$ can be found for which $q \circ a = m$ and $q \models_{\text{GC}} \phi$ are satisfied. The operation $\mathcal{R}\text{epair}_{\text{add}}$ resolves this violation by (a) using AutoGraph to construct a suitable graph H_s and by (b) integrating this graph H_s into G resulting in G' such that a suitable match $q : H' \hookrightarrow G'$ can be found.

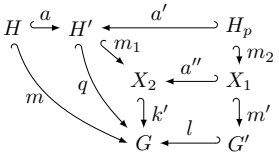
Definition 17 (Local Addition Operation $\mathcal{R}\text{epair}_{\text{add}}$). *If $a : H \hookrightarrow H'$, $\phi \in \Phi_{H'}^{\text{GC}}$, $m : H \hookrightarrow G$, $H_s \in \mathcal{A}(\exists(\text{i}_H, \exists(a, \phi)))$, $k : H \hookrightarrow H_s$, and $(\bar{m} : H_s \hookrightarrow G', r : G \hookrightarrow G')$ is the pushout of (m, k) then $r \in \mathcal{R}\text{epair}_{\text{add}}(a, \phi, m)$.*

$$\begin{array}{ccccc} H' & \xleftarrow{a} & H & \xhookrightarrow{k} & H_s \\ & & \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle \bar{m}} \\ & & G & \xhookrightarrow{r} & G' \end{array}$$

In our running example, $\mathcal{R}\text{epair}_{\text{add}}$ determines a graph repair resulting in the graph marked 2 in Fig. 4. For this repair, we considered the sub-ST marked by (R2) in Fig. 3d, where the morphism m matches the node a from $\boldsymbol{\psi}$ to the node a_2 in $\mathbf{G'_u}$, but where no extension of m can also match a node $:B$ and an edge between these two nodes. The repair performed then uses $a \xrightarrow{e} b$ for the graph H_s, resulting in the addition of the node b_2 and the edge from a_2 to b_2.

Secondly, a GC $\exists(a : H \hookrightarrow H', \phi)$ occurring as a subcondition in the consistency constraint ψ may be satisfied even though it should not when occurring underneath some negation. Such a violation is determined, again for a given match $m : H \hookrightarrow G$, by some match $q : H' \hookrightarrow G$ satisfying $q \circ a = m$ and $q \models_{\mathrm{GC}} \phi$. The local repair operation $\mathcal{R}\mathrm{epair}_{\mathrm{del}}$ repairs such an undesired satisfaction by selecting a graph H_p such that $H \subseteq H_p \subset H'$ using a restriction tree (see Definition 8) and deleting $G_{del} = q(H') \setminus q(H_p)$ from G. Technically, we can not use the pushout complement of a' and q as it does not exists when edges from $G \setminus G_{del}$ are attached to nodes in G_{del}. Hence, we determine the pushout complement of a'' and k', which must be constructed for this purpose suitably.

Definition 18 (Local Deletion Operation $\mathcal{R}\mathrm{epair}_{\mathrm{del}}$). *If $a : H \hookrightarrow H'$, $q : H' \hookrightarrow G$, $a' : H_p \hookrightarrow H' \in \mathrm{RT}(H', H)$, $m_1 : H' \hookrightarrow X_2$ where X_2 is obtained from $q(H')$ by adding all edges (with their nodes) that are connected to nodes in $q(H') \setminus q(a'(H_p))$, $k' : X_2 \hookrightarrow G$ is obtained such that $k' \circ m_1 = q$, $m_2 : H_p \hookrightarrow X_1$ where X_1 is obtained from H_p by adding all nodes in $X_2 \setminus q(H')$, $a'' : X_1 \hookrightarrow X_2$ is obtained such that $a'' \circ m_2 = m_1 \circ a'$, and $(l : G' \hookrightarrow G, m' : X_1 \hookrightarrow G')$ is the pushout complement of (a'', k') then $l \in \mathcal{R}\mathrm{epair}_{\mathrm{del}}(a, q)$.*



In our example, $\mathcal{R}\mathrm{epair}_{\mathrm{del}}$ determines a repair resulting in the graph marked 1 in Fig. 4. For this repair, we considered the sub-ST marked by (R1) in Fig. 3d where the mono m matches the node a from $\boldsymbol{\psi}$ to the node a_2 in $\mathbf{G'_u}$. The repair performed then uses $H_p = \emptyset$ for the removal of the node a_2 along with its adjacent loop (for which the technical handling in $\mathcal{R}\mathrm{epair}_{\mathrm{del}}$ is required).

The recursive operation $\mathcal{R}\mathrm{epair}_{\mathrm{db1}}$ below derives updates from an ST γ that corresponds to the current graph G (for our running example, these are $\gamma'_{\mathbf{u}}$ and $\mathbf{G'_u}$ from Fig. 3d). In the algorithm $\mathcal{R}\mathrm{epair}_{\mathrm{db}}$, we apply $\mathcal{R}\mathrm{epair}_{\mathrm{db1}}$ for the initial match i_G, γ, and *true* where this boolean indicates that we want γ to be satisfied. This boolean is changed in Rule 3 whenever the recursion is applied to an ST $\neg\gamma'$ because we expect that γ' is not to be satisfied iff we expect that $\neg\gamma'$ is to be satisfied. For conjunction, we either attempt to repair a sub-ST for $b = \mathit{true}$ in Rule 1 or we attempt to break one sub-ST for $b = \mathit{false}$. For existential quantification and $b = \mathit{true}$, we use $\mathcal{R}\mathrm{epair}_{\mathrm{add}}$ as discussed before in Rule 4 or we attempt to repair one existing match contained in m_f in Rule 5. Also, for existential quantification and $b = \mathit{false}$, we use $\mathcal{R}\mathrm{epair}_{\mathrm{del}}$ as discussed before in Rule 6 or we attempt to break one existing match contained in m_t in Rule 7.

Definition 19 (Single-Step Delta-Based Repair Algorithm $\mathcal{R}\mathrm{epair}_{\mathrm{db1}}$). *If $m : H \hookrightarrow G$, $\gamma \in \Gamma^{\mathrm{ST}}_m$, and $b \in \mathbf{B}$ then $(l : I \hookrightarrow G, r : I \hookrightarrow G') \in \mathcal{R}\mathrm{epair}_{\mathrm{db1}}(m, \gamma, b)$ if one of the following cases applies.*

- Rule 1 (repair one subcondition of a conjunction):
  $b = \mathit{true}, \gamma = \wedge S,\ \chi \in S,\ \not\models_{\mathrm{ST}} \chi,\ (l, r) \in \mathcal{R}\mathrm{epair}_{\mathrm{db1}}(m, \chi, b)$.
- Rule 2 (break one subcondition of a conjunction):
  $b = \mathit{false}, \gamma = \wedge S,\ \chi \in S,\ \models_{\mathrm{ST}} \chi,\ (l, r) \in \mathcal{R}\mathrm{epair}_{\mathrm{db1}}(m, \chi, b)$.
- Rule 3 (repair/break the subcondition of a negation):
  $\gamma = \neg\chi,\ (l, r) \in \mathcal{R}\mathrm{epair}_{\mathrm{db1}}(m, \chi, \neg b)$.
- Rule 4 (repair an existential quantification by local extension):
  $b = \mathit{true}, \gamma = \exists(a, \phi, m_t, m_f),\ m_t = \emptyset,\ r \in \mathcal{R}\mathrm{epair}_{\mathrm{add}}(a, \phi, m),\ l = \mathrm{id}_G$.
- Rule 5 (repair an existential quantification recursively):
  $b = \mathit{true}, \gamma = \exists(a, \phi, m_t, m_f),\ m_t = \emptyset,\ m_f(k) = \chi,\ (l, r) \in \mathcal{R}\mathrm{epair}_{\mathrm{db1}}(k, \chi, b)$.
- Rule 6 (break an existential quantification by local removal):
  $b = \mathit{false}, \gamma = \exists(a, \phi, m_t, m_f),\ m_t(k) \neq \bot,\ l \in \mathcal{R}\mathrm{epair}_{\mathrm{del}}(a, k),\ r = \mathrm{id}_{G'}$.
- Rule 7 (break an existential quantification recursively):
  $b = \mathit{false}, \gamma = \exists(a, \phi, m_t, m_f),\ m_t(k) = \chi,\ (l, r) \in \mathcal{R}\mathrm{epair}_{\mathrm{db1}}(k, \chi, b)$.

We define the recursive algorithm $\mathcal{R}\mathrm{epair}_{\mathrm{db}}$ to apply $\mathcal{R}\mathrm{epair}_{\mathrm{db1}}$ to obtain repairs as iterated applications of single-step repairs computed by $\mathcal{R}\mathrm{epair}_{\mathrm{db1}}$.

Definition 20 (Delta-Based Repair Algorithm $\mathcal{R}\mathrm{epair}_{\mathrm{db}}$). *If* $u = (l : I \hookrightarrow G, r : I \hookrightarrow G') \in \mathcal{U},\ \gamma \in \Gamma^{\mathrm{ST}}_{\mathrm{i}_G}$, *and* $\gamma' = \mathrm{ppgU}(\gamma, u)$ *then* $\mathcal{R}\mathrm{epair}_{\mathrm{db}}(u, \gamma) = S$ *if one of the following cases applies.*

- $\models_{\mathrm{ST}} \gamma'$ *and* $S = \{((\mathrm{id}_{G'}, \mathrm{id}_{G'}), \gamma')\}$.
- $\not\models_{\mathrm{ST}} \gamma'$, $S' = \{(u', \mathrm{ppgU}(\gamma', u')) \mid u' \in \mathcal{R}\mathrm{epair}_{\mathrm{db1}}(\mathrm{i}_G, \gamma', \mathit{true})\}$, *and* $S = \{(u', \gamma') \in S' \mid \models_{\mathrm{ST}} \gamma'\} \cup \bigcup\{(u'' \circ u', \gamma'') \mid (u', \gamma') \in S', \not\models_{\mathrm{ST}} \gamma', (u'', \gamma'') \in \mathcal{R}\mathrm{epair}_{\mathrm{db}}(u', \gamma'), u'' \circ u' \neq \bot\}$.[3]

This computation does not terminate when repairs trigger each other ad infinitum. However, a breadth-first-computation of $\mathcal{R}\mathrm{epair}_{\mathrm{db}}$ gradually computes a set of sound repairs. Obviously, GCs that trigger such nonterminating computations should be avoided but machinery for detecting such GCs is called for.

Note that the algorithm $\mathcal{R}\mathrm{epair}_{\mathrm{db}}$ computes fewer graph repairs compared to $\mathcal{R}\mathrm{epair}_{\mathrm{sb},2}$ because repairs are applied locally in the scope defined by the GC ψ. For example, no repair would be constructed resulting in the graph marked 4 in Fig. 2. In general, explicitly also using bigger contexts in ψ results in the additional computation of less–local graph repairs. For example, the condition $\boldsymbol{\psi}$ may be rephrased into $\boldsymbol{\psi}' = \boldsymbol{\psi} \wedge \neg\exists(a\ \ b, \neg\exists(a \xrightarrow{e} b, \mathit{true}))$ to also obtain the graph repair marked 4 in Fig. 2. We now define the updates, which we expect to be computed by $\mathcal{R}\mathrm{epair}_{\mathrm{db1}}$, as those that repair a single violation of the GC ψ by defining a local update to be embeddable into the resulting update via a double pushout diagram as in the DPO approach to graph transformation [16].

Definition 21 (Locally Least Changing Graph Update). *If* G_1 *is a graph,* $\psi \in \Phi^{\mathrm{GC}}_{\emptyset}$, $G_1 \not\models_{\mathrm{GC}} \psi$, $(l : I \hookrightarrow G_1, r : I \hookrightarrow G_2) \in \mathcal{U}_{\mathrm{lc}}(G_1, \psi)$, $G_2 \models_{\mathrm{GC}} \psi$, X_1 *is a minimal subgraph of* G_1 *with a violation of* ψ *that is also a violation of* ψ *in*

[3] If u_1 and u_2 are updates then $u_1 \circ u_2 = u$ if $u_1 \leq^{u_2} u$ or $u = \bot$ otherwise (see Definition 4).

G, and the diagram below exists and the right part of it is a DPO diagram then (l, r) *is a* locally least changing graph update.

$$\begin{array}{ccccc} X_1 & \hookleftarrow & I' & \hookrightarrow & X_2 \\ \downarrow & & \downarrow & & \downarrow \\ G_1 & \overset{l}{\hookleftarrow} & I & \overset{r}{\hookrightarrow} & G_2 \end{array}$$

$\mathcal{R}\text{epair}_{\text{db1}}$ indeed generates such locally least changing graph updates because the graph X_1 in this definition corresponds to the H_1 and the H_2 from an ST $\exists(a : H_1 \hookrightarrow H_2, \phi, m_t, m_f)$ that is subject to $\mathcal{R}\text{epair}_{\text{add}}$ and $\mathcal{R}\text{epair}_{\text{del}}$, respectively. For example, for $\mathcal{R}\text{epair}_{\text{add}}$, the graph H_1 in the ST determines a subgraph in G_1 that is a violation of the overall consistency condition given by a GC ψ as its match can not be extended to the graph H_2.

We now define the locally least changing graph repairs (which are to be computed by $\mathcal{R}\text{epair}_{\text{db}}$ such as for example the graphs marked 1 and 4 in Fig. 4) as the composition of a sequence of locally least changing updates where precisely the last graph update results in a graph satisfying the GC ψ.

Definition 22 (Locally Least Changing Graph Repair). *If* G_1 *is a graph,* $\psi \in \Phi_{\emptyset}^{\text{GC}}$, $\pi = (l_1 : I_1 \hookrightarrow G_1, r_1 : I_1 \hookrightarrow G_2) \dots (l_n : I_n \hookrightarrow G_n, r_n : I_n \hookrightarrow G_{n+1})$ *is a sequence of locally least changing graph updates,* $G_1 \in [\![\psi]\!]$ *implies* $n = 0$ *and* $l_1 = r_1 = \text{id}_{G_1}$, $G_i \notin [\![\psi]\!]$ *(for each* $2 \leq i \leq n$*),* $G_{n+1} \in [\![\psi]\!]$, (l, r) *is the iterated composition of the updates in* π*, and* $(l, r) \in \mathcal{U}(G_1, \psi)$ *is a least changing graph repair then* (l, r) *is a* locally least changing graph repair*.*

We now state that our delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ returns all desired locally least changing graph repairs upon termination.

Theorem 5 (Functional Semantics of $\mathcal{R}\text{epair}_{\text{db}}$**).** $\mathcal{R}\text{epair}_{\text{db}}$ *is sound (i.e., it generates only locally least changing graph repairs) and complete (upon termination) with respect to locally least changing graph repairs.*

The state-based algorithms $\mathcal{R}\text{epair}_{\text{sb},1}$ and $\mathcal{R}\text{epair}_{\text{sb},2}$ are inappropriate in environments where numerous updates that may invalidate consistency are applied to a large graph because the procedure of AutoGraph has exponential cost. The incremental delta-based algorithm $\mathcal{R}\text{epair}_{\text{db}}$ is a viable alternative when additional memory requirements for storing the ST are acceptable. The AutoGraph applications for this algorithm have negligible costs because they may be performed a priori and must only be performed for subconditions of the consistency constraint, which can be assumed to feature reasonably small graphs only.

Finally, a classification of locally least changing repairs is useful for user-based repair selection. Delta preserving repairs defined below represent such a basic class, containing only those repairs that preserve the update resulting in a graph not satisfying GC ψ, i.e., it may be desirable to avoid repairs that revert additions or deletions of this update. In our example, the repair related to the graph marked 4 in Fig. 4 is not delta preserving w.r.t. **u** from Fig. 3a.

Definition 23 (Delta Preserving Graph Repair). *If* $\psi \in \Phi_{\emptyset}^{\text{GC}}$, $u_2 = (l_2 : I_2 \hookrightarrow G_2, r_2 : I_2 \hookrightarrow G_3) \in \mathcal{U}(G_2, \psi)$ *is a graph repair,* $u_1 = (l_1 : I_1 \hookrightarrow G_1, r_1 :$

$I_1 \hookrightarrow G_2$) *is a graph update, and there exists a graph update* u *such that* $u_1 <^{u_2} u$ *then* u_2 *is a* delta preserving graph repair *with respect to* u_1.

7 Related Work

According to the recent survey on *model repair* [12], and the corresponding exhaustive classification of primary studies selected in the literature review, published online [11], we can see that the amount and wide variety of existing approaches makes a detailed comparison with all of them infeasible.

We consider our approach to be innovative, not only because of the proposed solutions, but because it addresses the issues of *completeness* and *least changing* for incremental graph repair in a precise and formal way. From the survey [11,12] we can see that only two other approaches [10,19] address completeness and least changing, relying also on constraint-solving technology. The main difference with our approach is that they are not incremental. In particular, the work of Schoenboeck et al. [19] proposes a logic programming approach allowing the exploration of model repair solutions ranked according to some quality criteria, re-establishing conformance of a model with its metamodel. Soundness and completeness of these repair actions is not formally proven. Moreover, the least changing bidirectional model transformation approach of Macedo et al. [10] has only a bounded search for repairs, relying on a bounded constraint solver.

Some *recent work* on rule-based *graph repair* [9] (not covered by the survey) addresses the least-changing principle by developing so-called maximally preserving (items are preserved whenever possible) repair programs. This state-based approach considers a subset of consistency constraints (up to nesting depth 2) handled by our approach, and is not complete, since it produces repairs including only a minimal amount of deletions. Some other recent rule-based graph repair approach [13,20] (also not covered by the survey) proposes so-called change preserving repairs (similar to what we define as delta-preserving). The main difference with our work is that we do not require the user to specify consistency-preserving operations from which repairs are generated, since we derive repairs using constraint solving techniques directly from the consistency constraints.

Finally, there is a variety of work on *incremental evaluation of graph queries* (see e.g. [2,4]), developed with the aim of efficiently re-evaluating a graph query after an update has been performed. Although not employed with the specific aim of complete and least changing graph repair, this work is related to our newly introduced concept of satisfaction trees, also using specific data structures to record with some detail the set of answers to a given query (as described for graph conditions, for example, also in [3]). It is part of ongoing work to evaluate how STs can be employed similarly in this field of incremental query evaluation.

8 Conclusion and Future Work

We presented a logic-based incremental approach to graph repair. It is the first approach to graph repair returning a sound and complete overview of least

changing repairs with respect to graph conditions equivalent to first-order logic on graphs. Technically, it relies on an existing model generation procedure for graph conditions together with the newly introduced concept of satisfaction trees, encoding if and how a graph satisfies a graph condition.

As future work, we aim at supporting partial consistency and gradually improving it. We are confident that we can extend our work to support attributes, since our underlying model generation procedure supports it. Ongoing work is the support of more expressive consistency constraints, allowing path-related properties. Moreover, we are in the process of implementing the algorithms presented here and evaluating them on a variety of case studies. The evaluation also pertains to the overall efficiency (for which we employ techniques for localized pattern matching) and includes a comparison with other approaches for graph repair. Finally, we aim at presenting new and refined properties distinguishing between all possible repairs supporting the implementation of interactive repair selection procedures.

References

1. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 1:1–1:39 (2008). https://doi.org/10.1145/1322432.1322433
2. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the viatra model transformation system. In: GRaMoT, pp. 25–32. ACM (2008). https://doi.org/10.1145/1402947.1402953
3. Beyhl, T., Blouin, D., Giese, H., Lambers, L.: On the operationalization of graph queries with generalized discrimination networks. In: Echahed, R., Minas, M. (eds.) ICGT 2016. LNCS, vol. 9761, pp. 170–186. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40530-8_11
4. Beyhl, T., Giese, H.: Incremental view maintenance for deductive graph databases using generalized discrimination networks. In: GaM@ETAPS, EPTCS, vol. 231, pp. 57–71 (2016). https://doi.org/10.4204/EPTCS.231.5
5. Courcelle, B.: The expression of graph properties and graph transformations in monadic second-order logic. In: Rozenberg [16], pp. 313–400
6. Diskin, Z., König, H., Lawford, M.: Multiple model synchronization with multiary delta lenses. In: Russo, A., Schürr, A. (eds.) FASE 2018. LNCS, vol. 10802, pp. 21–37. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89363-1_2
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
8. Habel, A., Pennemann, K.: Correctness of high-level transformation systems relative to nested conditions. MSCS **19**(2), 245–296 (2009). https://doi.org/10.1017/S0960129508007202
9. Habel, A., Sandmann, C.: Graph repair by graph programs. In: Mazzara, M., Ober, I., Salaün, G. (eds.) STAF 2018. LNCS, vol. 11176, pp. 431–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_31
10. Macedo, N., Cunha, A.: Least-change bidirectional model transformation with QVT-R and ATL. Softw. Syst. Model. **15**(3), 783–810 (2016). https://doi.org/10.1007/s10270-014-0437-x

11. Macedo, N., Tiago, J., Cunha, A.: Systematic literature review of model repair approaches. http://tinyurl.com/hv7eh6h. Accessed 14 Nov 2018
12. Macedo, N., Tiago, J., Cunha, A.: A feature-based classification of model repair approaches. IEEE Trans. Softw. Eng. **43**(7), 615–640 (2017). https://doi.org/10.1109/TSE.2016.2620145
13. Ohrndorf, M., Pietsch, C., Kelter, U., Kehrer, T.: Revision: a tool for history-based model repair recommendations. In: ICSE, pp. 105–108. ACM (2018). https://doi.org/10.1145/3183440.3183498
14. Orejas, F., Boronat, A., Ehrig, H., Hermann, F., Schölzel, H.: On propagation-based concurrent model synchronization. ECEASST **57** (2013). http://journal.ub.tu-berlin.de/eceasst/article/view/871
15. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_23
16. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific (1997)
17. Schneider, S., Lambers, L., Orejas, F.: Automated reasoning for attributed graph properties. STTT **20**(6), 705–737 (2018). https://doi.org/10.1007/s10009-018-0496-3
18. Schneider, S., Lambers, L., Orejas, F.: A logic-based incremental approach to graph repair. Technical report, 126, Hasso Plattner Institute at the University of Potsdam, Potsdam, Germany (2019)
19. Schoenboeck, J., et al.: CARE - A constraint-based approach for re-establishing conformance-relationships. In: APCCM 2014, vol. 154, pp. 19–28. Australian Computer Society (2014). http://crpit.com/abstracts/CRPITV154Schoenboeck.html
20. Taentzer, G., Ohrndorf, M., Lamo, Y., Rutle, A.: Change-preserving model repair. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 283–299. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_16