Einar Broch Johnsen
Manuel Wimmer (Eds.)

LNCS 13241 ARCoSS

Fundamental Approaches to Software Engineering

25th International Conference, FASE 2022
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2022
Munich, Germany, April 2–7, 2022
Proceedings

Springer

OPEN ACCESS

More information about this series at https://link.springer.com/bookseries/558

Einar Broch Johnsen · Manuel Wimmer (Eds.)

Fundamental Approaches to Software Engineering

25th International Conference, FASE 2022
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2022
Munich, Germany, April 2–7, 2022
Proceedings

Springer

Editors
Einar Broch Johnsen
University of Oslo
Oslo, Norway

Manuel Wimmer
Johannes Kepler University Linz
Linz, Austria

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-99428-0 ISBN 978-3-030-99429-7 (eBook)
https://doi.org/10.1007/978-3-030-99429-7

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 25th ETAPS! ETAPS 2022 took place in Munich, the beautiful capital of Bavaria, in Germany.

ETAPS 2022 is the 25th instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference established in 1998, and consists of four conferences: ESOP, FASE, FoSSaCS, and TACAS. 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 of programming languages, analysis tools, and formal approaches to software engineering. Organizing these conferences in a coherent, highly synchronized conference program enables researchers to participate in an exciting event, having the possibility to meet many colleagues working in different directions in the field, and to easily attend talks of different conferences. On the weekend before the main conference, numerous satellite workshops took place that attract many researchers from all over the globe.

ETAPS 2022 received 362 submissions in total, 111 of which were accepted, yielding an overall acceptance rate of 30.7%. 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 2022 featured the unifying invited speakers Alexandra Silva (University College London, UK, and Cornell University, USA) and Tomáš Vojnar (Brno University of Technology, Czech Republic) and the conference-specific invited speakers Nathalie Bertrand (Inria Rennes, France) for FoSSaCS and Lenore Zuck (University of Illinois at Chicago, USA) for TACAS. Invited tutorials were provided by Stacey Jeffery (CWI and QuSoft, The Netherlands) on quantum computing and Nicholas Lane (University of Cambridge and Samsung AI Lab, UK) on federated learning.

As this event was the 25th edition of ETAPS, part of the program was a special celebration where we looked back on the achievements of ETAPS and its constituting conferences in the past, but we also looked into the future, and discussed the challenges ahead for research in software science. This edition also reinstated the ETAPS mentoring workshop for PhD students.

ETAPS 2022 took place in Munich, Germany, and was organized jointly by the Technical University of Munich (TUM) and the LMU Munich. The former was founded in 1868, and the latter in 1472 as the 6th oldest German university still running today. Together, they have 100,000 enrolled students, regularly rank among the top 100 universities worldwide (with TUM's computer-science department ranked #1 in the European Union), and their researchers and alumni include 60 Nobel laureates.

The local organization team consisted of Jan Křetínský (general chair), Dirk Beyer (general, financial, and workshop chair), Julia Eisentraut (organization chair), and Alexandros Evangelidis (local proceedings chair).

ETAPS 2022 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 ETAPS Steering Committee 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 Holger Hermanns (Saarbrücken), Marieke Huisman (Twente, chair), Jan Kofroň (Prague), Barbara König (Duisburg), Thomas Noll (Aachen), Caterina Urban (Paris), Tarmo Uustalu (Reykjavik and Tallinn), and Lenore Zuck (Chicago).

Other members of the Steering Committee are Patricia Bouyer (Paris), Einar Broch Johnsen (Oslo), Dana Fisman (Be'er Sheva), Reiko Heckel (Leicester), Joost-Pieter Katoen (Aachen and Twente), Fabrice Kordon (Paris), Jan Křetínský (Munich), Orna Kupferman (Jerusalem), Leen Lambers (Cottbus), Tiziana Margaria (Limerick), Andrew M. Pitts (Cambridge), Elizabeth Polgreen (Edinburgh), Grigore Roşu (Illinois), Peter Ryan (Luxembourg), Sriram Sankaranarayanan (Boulder), Don Sannella (Edinburgh), Lutz Schröder (Erlangen), Ilya Sergey (Singapore), Natasha Sharygina (Lugano), Pawel Sobocinski (Tallinn), Peter Thiemann (Freiburg), Sebastián Uchitel (London and Buenos Aires), Jan Vitek (Prague), Andrzej Wasowski (Copenhagen), Thomas Wies (New York), Anton Wijs (Eindhoven), and Manuel Wimmer (Linz).

I'd like to take this opportunity to thank all authors, attendees, organizers of the satellite workshops, and Springer-Verlag GmbH for their support. I hope you all enjoyed ETAPS 2022.

Finally, a big thanks to Jan, Julia, Dirk, and their local organization team for all their enormous efforts to make ETAPS a fantastic event.

February 2022

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

This volume contains the papers presented at FASE 2022, the 25th International Conference on Fundamental Approaches to Software Engineering. FASE 2022 was organized as part of the annual European Joint Conferences on Theory and Practice of Software (ETAPS 2022).

FASE is concerned with the foundations on which software engineering is built, including topics like software engineering as an engineering discipline, requirements engineering, software architectures, software quality, model-driven development, software processes, software evolution, AI-based software engineering, and the specification, design, and implementation of particular classes of systems, such as (self-) adaptive, collaborative, AI, embedded, distributed, mobile, pervasive, cyber-physical, or service-oriented applications.

FASE 2022 received 61 submissions and used a double-anonymous reviewing process. Each submission was reviewed by three Program Committee members. After an online discussion period, the Program Committee accepted 17 papers as part of the conference program (28% acceptance rate).

FASE 2022 hosted the 4th International Competition on Software Testing (Test-Comp 2022). Test-Comp is an annual comparative evaluation of testing tools. This edition contained 12 participating tools, from academia and industry. These proceedings contain the competition report and two system descriptions of participating tools. The system-description papers were reviewed and selected by a separate Program Committee: the Test-Comp jury. Each paper was assessed by at least three reviewers. Two sessions in the FASE program were reserved for the presentation of the results: the summary by the Test-Comp chair and of the participating tools by the developer teams in the first session, and the community meeting in the second session.

Many people contributed to the success of FASE 2022. We are grateful to the Program Committee members and reviewers for their thorough reviews and constructive discussions. We thank the ETAPS 2022 organizers, in particular, Jan Křetínský and Dirk Beyer (General Chairs), Julia Eisentraut (Organization Chair), Maximilian Weininger (Web Chair), and Alexandros Evangelidis (Proceedings Chair). We thank Marieke Huisman (ETAPS Steering Committee Chair) and Tarmo Uustalu (ETAPS Publicity Chair) for managing the process, and Andrzej Wasowski (FASE Steering Committee Chair) for his feedback and support. We are especially grateful to our Artefact Evaluation Committee Chairs Marie-Christine Jakobs and Eduard Kamburjan. Last but not least, we would like to thank the authors for their excellent work.

February 2022

Einar Broch Johnsen
Manuel Wimmer

Organization

Program Committee Chairs

Einar Broch Johnsen	University of Oslo, Norway
Manuel Wimmer	Johannes Kepler University Linz, Austria

Steering Committee

Wil van der Aalst	RWTH Aachen, Germany
Jordi Cabot	Universitat Oberta de Catalunya, Spain
Marsha Chechik	University of Toronto, Canada
Esther Guerra	Universidad Autónoma de Madrid, Spain
Reiner Hähnle	Technische Universität Darmstadt, Germany
Reiko Heckel	University of Leicester, UK
Tiziana Margaria	University of Limerick, Ireland
Fernando Orejas	Universitat Politècnica de Catalunya, Spain
Julia Rubin	University of British Columbia, Canada
Alessandra Russo	Imperial College London, UK
Andy Schürr	Technische Universität Darmstadt, Germany
Perdita Stevens	University of Edinburgh, UK
Mariëlle Stoelinga	Universiteit Twente, The Netherlands
Gabriele Taentzer	Philipps-Universität Marburg, Germany
Andrzej Wasowski	IT University of Copenhagen, Denmark
Heike Wehrheim	Universtät Paderborn, Germany

Program Committee

Erika Ábrahám	RWTH Aachen, Germany
Shaukat Ali	Simula Research Lab, Norway
Étienne André	Université de Lorraine, France
Thorsten Berger	Ruhr-Universität Bochum, Germany
Tomáš Bureš	Charles University, Czech Republic
Jane Cleland-Huang	University of Notre Dame, USA
Carlo Furia	Università della Svizzera italiana, Switzerland
Stijn de Gouw	Open Universiteit, The Netherlands
Esther Guerra	Universidad Autónoma de Madrid, Spain
Ichiro Hasuo	National Institute of Informatics, Japan
Ralf Huuck	Logilica, Australia
Marie-Christine Jacobs	Technische Universität Darmstadt, Germany
Gerti Kappel	TU Wien, Austria
Leen Lambers	BTU Cottbus-Senftenberg, Germany
Martin Leucker	Universität zu Lübeck, Germany

Yi Li	Nanyang Technological University, Singapore
Augusto Sampaio	Universidade Federal de Pernambuco, Brazil
Ina Schaefer	Technische Universität Braunschweig, Germany
Mariëlle Stoelinga	Universiteit Twente, The Netherlands
Daniel Strüber	Radboud Universiteit Nijmegen, The Netherlands
Gabriele Taentzer	Philipps-Universität Marburg, Germany
S. Lizeth Tapia Tarifa	University of Oslo, Norway
Sebastian Uchitel	Universidad de Buenos Aires, Argentina, and Imperial College London, UK
Dániel Varró	McGill University, Canada
Andrzej Wasowski	IT University Copenhagen, Denmark
Heike Wehrheim	Carl von Ossietzky Universität Oldenburg, Germany
Gianluigi Zavattaro	Università di Bologna, Italy
Steffen Zschaler	King's College London, UK

Artefact Evaluation Committee Chairs

Marie-Christine Jakobs	Technische Universität Darmstadt, Germany
Eduard Kamburjan	University of Oslo, Norway

Artifact Evaluation Committee

Chinmayi Baramashetru	University of Oslo, Norway
Saverio Giallorenzo	Università di Bologna, Italy, and Inria, France
Pablo Gómez-Abajo	Universidad Autónoma de Madrid, Spain
Elena Gómez-Martínez	Universidad Autónoma de Madrid, Spain
Jan Haltermann	Carl von Ossietzky Universität Oldenburg, Germany
Hannes Kallwies	Universität zu Lübeck, Germany
Dylan Marinho	Loria, Université de Lorraine, France
Felix Pauck	Universität Paderborn, Germany
Sven Peldszus	Universität Koblenz-Landau, Germany
Mohammad Rezaalipour	Università della Svizzera italiana, Switzerland
Cedric Richter	Carl von Ossietzky Universität Oldenburg, Germany
Maya Retno Ayu Setyautami	Universitas Indonesia, Indonesia
Asmae Heydari Tabar	Technische Universität Darmstadt, Germany
Matthias Volk	Universiteit Twente, The Netherlands
Xiuheng Wu	Nanyang Technological University, Singapore
Liu Ye	Nanyang Technological University, Singapore

Additional Reviewers

Milad Abdullah	Luca Berardinelli
Aren Babikian	Benedikt Bollig
Lara Bargmann	Tabea Bordis

Ricardo Caldas
Diego Damasceno
Istvan David
Francesca Del Bonifro
Swaib Dragule
Leo Freitas
Antonio Garmendia
Marcus Gerhold
Bineet Ghosh
Razan Ghzouli
Hans-Dieter Hiep
Tobias John
Danylo Khalyeyev
Karam Kharraz
Alexander Knüppel
Paul Kobialka
Jens Kosiol
Wardah Mahmood
Mukelabai Mukelabai
Argentina Ortega
Juliane Päßler
Tobias Pett
Ricardo Prudencio
Qusai Ramadan
Mathias Ramparison
Henrique Rebêlo
Tobias Runge
Lucas Sakizloglou
Arnaud Sangnier
Sota Sato
Alceste Scalas
Malte Schmitz
Martina Seidl
Oszkár Semeráth
Arnab Sharma
Thiago D. Simão
Sabine Sint
Martin Steffen
Daniel Thoma
Palina Tolmach
Michal Töpfer
Gianluca Turin
Freek Verbeek
Chenguang Zhu

Program Committee and Jury — Test-Comp

Kaled Alshmrany	University of Manchester, UK, and Institute of Public Administration, Saudi Arabia
Marek Chalupa	Masaryk University, Czech Republic
Gidon Ernst	LMU Munich, Germany
Joxan Jaffar	National University of Singapore, Singapore
Marie-Christine Jakobs	TU Darmstadt, Germany
Raveendra Kumar Medicherla	Tata Consultancy Services, India
Hoang M. Le	University of Bremen, Germany
Thomas Lemberger	LMU Munich, Germany

Contents

FASE Contributions

FASE Contributions

Information-flow Interfaces*

Ezio Bartocci[1], Thomas Ferrère[2], Thomas A. Henzinger[3], Dejan Nickovic[4], and Ana Oliveira da Costa[1](✉)

[1] Technische Universität Wien, Vienna, Austria
{ezio.bartocci, ana.costa}@tuwien.ac.at
[2] Imagination Technologies, Kings Langley, UK
thomas.ferrere@imgtec.com
[3] IST Austria, Klosterneuburg, Austria
tah@ist.ac.at
[4] AIT Austrian Institute of Technology, Vienna, Austria
dejan.nickovic@ait.ac.at

Abstract. Contract-based design is a promising methodology for taming the complexity of developing sophisticated systems. A formal contract distinguishes between *assumptions*, which are constraints that the designer of a component puts on the environments in which the component can be used safely, and *guarantees*, which are promises that the designer asks from the team that implements the component. A theory of formal contracts can be formalized as an *interface theory*, which supports the composition and refinement of both assumptions and guarantees. Although there is a rich landscape of contract-based design methods that address functional and extra-functional properties, we present the first interface theory that is designed for ensuring system-wide security properties. Our framework provides a refinement relation and a composition operation that support both incremental design and independent implementability. We develop our theory for both *stateless* and *stateful* interfaces. We illustrate the applicability of our framework with an example inspired from the automotive domain.

Keywords: Contract-based design, Interface Theory, Hyperproperties, Information-flow.

1 Introduction

The rise of pervasive information and communication technologies seen in cyber-physical systems, internet of things, and blockchain services has been accompanied by a tremendous growth in the size and complexity of systems [28]. Subtle dependencies involving multiple architectural layers and unforeseen environmental interactions can expose these systems to cyber-attacks. This problem is further exacerbated by the heterogeneous nature of their constituent components,

* This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 956123 and was funded in part by the FWF project W1255-N23 and by the ERC-2020-AdG 101020093.

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 3–22, 2022.
https://doi.org/10.1007/978-3-030-99429-7_1

which are often developed independently by different teams or providers. In such a scenario, defining and enforcing security requirements across components at an early stage of the design process becomes a necessity. This engineering approach is called *security-by-design*. Although in recent years there has been impressive progress in the verification of security properties for individual system components, the science of compositional security design [22,23] is still in its infancy.

Security policies are usually enforced by restricting the flow of information in a system [30]. *Information-flow policies* define which information a user or a software/hardware component is allowed to observe or to interfere with while interacting with another component.

The goal of information-flow control is to ensure that a system as a whole satisfies the desired policies. It is especially challenging to verify that there are no side-channels or implicit flows that violate a given policy. For example, in a modern car, the tight coupling between the cyber and the physical components allows an attacker to infer computational properties, such as secrets used for encryption, from side-channels, such as power consumption and electromagnetic radiation [17]. Moreover, the increasing connectivity of automotive systems with their environment makes it easier for the attacker to gather data about the system behavior. The attacker can use this data to exploit weaknesses of the system implementation and gain control over the system [32,7]. These attacks often rely on analyzing and comparing multiple observations to deduce protected information. From a formal-language perspective, such security vulnerabilities are not characterized by properties of a single system execution, but rather by properties of sets of execution traces, which are called *hyperproperties* [12].

The rigorous design of systems that satisfy information flow requirements is essential from the security perspective. This activity can be supported by the verification of information flow properties, a well-studied problem with a rich landscape of both theory and tools, ranging from language-based [29,18,15,11] to simulation-based [25] approaches. Nevertheless, the existing verification solutions do not address two important aspects. First, components in complex systems are often heterogeneous and cannot be analysed with a single verification tool. Moreover, it is not clear how to combine component verification outcomes to infer system-level information-flow properties. Second, existing methods do not provide guidelines on which information flow properties need to be verified against individual components to provide system-level guarantees regarding leakage of information.

In this paper, we present a *contract-based design* [8] approach for the structural aspect of information-flow policies. Contract-based design provides a formal framework for building complex systems from individual components, mixing both top-down and bottom-up steps. A top-down step decomposes and refines system-wide requirements; a bottom-up step assembles a system by combining available components. A formal contract distinguishes between *assumptions*, which are constraints that the designer of a component puts on the environments in which the component can be used safely, and *guarantees*, which are promises that the designer asks from the team that implements the component. A theory

of formal contracts can be formalized as an *interface theory*, which supports the composition and refinement of both assumptions and guarantees [2,3,31]. While there is a rich landscape of interface theories for functional and extra-functional properties [10,4,13,20], we present the first interface theory that is designed for ensuring system-wide security properties, thus paving the way for a science of safety and security co-engineering.

The focus on the structural aspects of information flow and abstraction from concrete semantics enables compositional reasoning in presence of heterogeneous components and is complementary to the existing body of work on information flow verification. A different component implementation verified under different semantics could result in different flows being detected. However, after deriving the component flows from the implementation under some concrete semantics, the theory can be agnostic about the underlying semantic interpretation. Hence it enables the design of secure systems from trusted components by abstracting away how information flows and by focusing on whether it can flow at all. In essence, our approach enables to decompose system-level information flow requirements and derive component properties that need to hold, thus providing a divide-and-conquer procedure for organizing verification tasks.

Our theory is based on information-flow assumptions as well as information-flow guarantees. As an interface theory, our theory supports both *incremental design* and *independent implementability* [3]. Incremental design allows the composition of different system parts, each coming with their own assumptions and guarantees, without requiring additional knowledge of the overall design context. Independent implementability enables the separate refinement of different system parts by different teams that, without gaining additional information about each other's design choices, can still be certain that their designs, once combined, preserve the specified system-wide requirements. While in previous interface theories, the environment of a component is held responsible for meeting assumptions, and the implementation of the component for the guarantees, there are cases of information-flow violations for which blame cannot be assigned uniquely to the implementation or the environment. In information-flow interfaces we therefore introduce, besides assumptions and guarantees, a new, third type of constraint—called *properties*—whose enforcement is the shared responsibility of the implementation and the environment.

We develop our framework for both *stateless* and *stateful* interfaces. Stateless information-flow interfaces are built from primitive information-flow constraints—assumptions, guarantees, and properties—of the form "the value of a variable x is always independent of the value of another variable y." Stateful information-flow interfaces add a temporal dimension, e.g., "the value of y is independent of x *until* the value of z is independent of x." The temporal dimension is introduced through a natural notion of state and state transition for interfaces, not through logical operators. We prove that our calculus of information-flow interfaces satisfies the principles of incremental design and independent implementability.

2 Application Example

We showcase the applicability of our theory with an example from the automotive industry: a stepwise design of a shared communication infrastructure (a bus) from distance warners and a wheel sensor to the braking system and the odometer. We adapted this use-case from the industrial case study presented by Marcus Mikulcak et al. [25]. The main goal of this system design is to ensure the integrity of a communication channel used to perform a safety-critical functionality. We consider two integrity levels, high and low, to characterize functionalities in our system. Then, we want to guarantee that data exchanged by high-integrity functionalities is not compromised by low-integrity functions.

Distance warners sense the car's proximity to other objects and send their analysis to other components. In our example, we have two distance warners, at the front and the back of the car, that use the shared bus to communicate with the braking system. The wheel sensor senses the wheel rotations and sends this information through the shared bus to the odometer. The braking system is a high-integrity system since it performs safety-critical functions. Hence the communication channel between the distance warners and the braking system is classified with high-integrity, while the communication between the wheel sensor and the odometer is low-integrity. Thus, data sent by the wheel sensor should not interfere with the high-integrity channel to prevent distance warnings sent to the braking system from being delayed or lost. The main goal of our design process is to guarantee that the closed system requirement that *information from the wheel sensor does not flow to the braking system* is propagated accordingly to subsystems through successive decomposing and refinement steps.

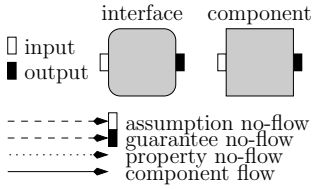


Fig. 1: Representation of the objects in our theory.

Figure 1 shows the graphical representation we adopt throughout the paper for the objects in our theory. We represent the open system no-flows requirements with dashed arrows. Then, arrows to input ports are *assumptions* while arrows to output parts are *guarantees*. The closed system no-flows, *properties*, are represented as dotted arrows. To improve the readability of the drawings, it is implicit that for each drawn property, we have the same guarantee over the open system. When it is clear from the context we may omit port(s) names.

We present, in Fig. 2, the stepwise design of the security requirement that data from the wheel sensor, *wheel_tick*, should not flow to the target of the distance warners, *distw_f_t* and *distw_b_t*. The first interface in Fig. 2 includes two properties that specify this security requirement. The system is then decomposed into the *sending subsystems* (warners and wheel sensor), the *shared bus*, and the *receiving subsystems*.

Naturally, we keep the two properties from the first interface as properties in the Bus interface. However, this natural decomposition *does not define a well-formed interface* according to our theory because the properties in the Bus interface cannot be satisfied given the interface's current assumption and

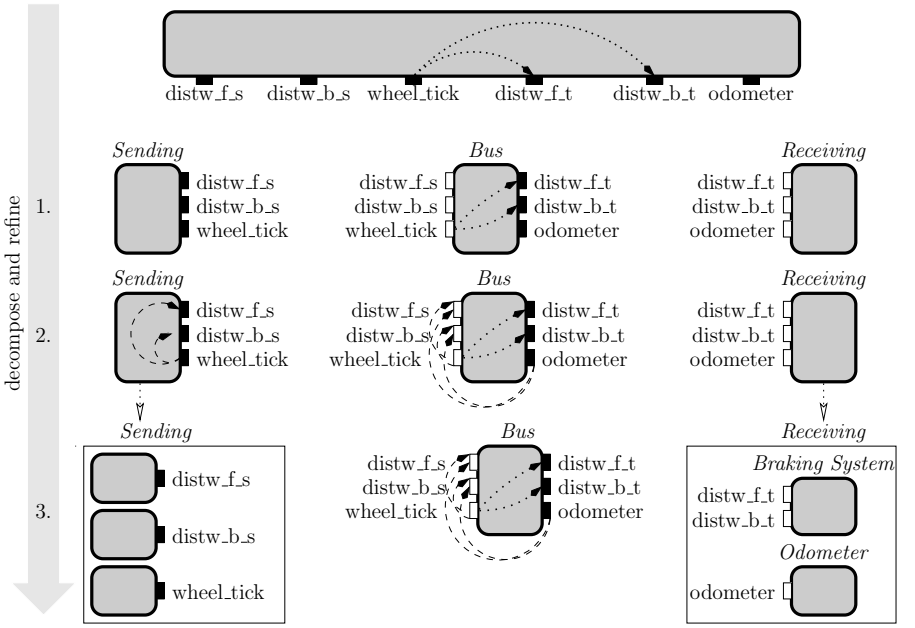


Fig. 2: Top-down design of a shared communication infrastructure used by two distance warners, *distw_f_s* and *distw_b_s*, and a wheel sensor, *wheel_tick*, to communicate with the braking system, *distw_f_t* and *distw_b_t*, and the odometer, *odometer*, respectively.

guarantee. As the environment allows a flow from *wheel_tick* to the source of the front distance warner, *wheel_tick* $\rightsquigarrow$ *distw_f_s* then, with the flow allowed by the guarantee from a distance warner source to its target, we have the flow *wheel_tick* $\rightsquigarrow$ *distw_f_s* $\rightsquigarrow$ *distw_f_t*. This flow is forbidden by the interface's properties. If we specified the no-flow properties in the Bus interface as guarantees, then the interface would be well-formed. However, the composition of the three subsystems would not satisfy the initial specification because guarantees only apply to implementations of their interface, and the flow described above would still be allowed in the composition of the three subsystems. This illustrates two applications of the information-flow interface theory: *to detect inconsistent no-flow specifications* and *faulty decompositions*. Moreover, when an interface is not well-formed we can provide a witness for the property violation. We can use this witness to *guide the refinement of an ill-formed interface* into a well-formed one.

In the second step of our refinement, in Figure 2, we add the missing assumptions to the Bus interface. Our notion of *composition compatibility between interfaces* requires that the Sending interface includes guarantees that implies the Bus assumptions, as the Sending interface will be part of the Bus environment. At this point, with a certified decomposition of the original specification, our theory guarantees that each subsystem can now be further *refined independently* (possibly by different teams). The last step illustrates an independent refinement of the Sending and the Receiving interfaces.

In Fig. 3, we present the stateful view of the system, which requires that the system satisfies the composition of the Sending, the Bus, and the Receiving interfaces derived in Fig. 2 at all times. We present, as well, a refinement of that specification, which requires that in each time point only one of the sending components can use the bus. The interfaces that define each state are named after the sending component that can use the bus (e.g. in the state S_{wheel} only the *wheel_tick* can use the bus). If the access to the bus is mutually exclusive, then we can simplify the assumptions on the environment in the Bus interface. With more guarantees on the implementations we need fewer assumptions to satisfy the properties.

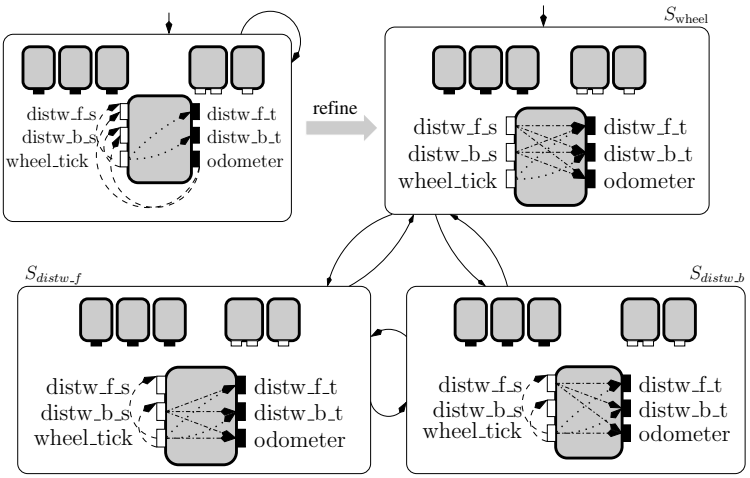


Fig. 3: Design of mutually exclusive shared communication infrastructure for distance warners and the wheel odometer. Each state is defined by the composition of the interfaces inside.

Finally, the components of our system can be, for instance, the Simulink and Stateflow models provided to the authors [25] by their industrial partners. We can then use the tool introduced in their work to verify whether these components implement the stateful interfaces we derived.

In summary, our framework defines relations on both stateless and stateful interfaces specifying information-flow policies that allow to check if: (i) a given interface refines (or abstracts) the current specification; (ii) two interfaces are compatible for composition; (iii) a specification is consistent; (iv) information-flows in a component define an implementation of a given interface; and (v) a system decomposition refines the system specification.

3 Stateless Information-flow Interfaces

In this section, we introduce a stateless interface and component algebra for secure information flow. Information flows between two variables when the value of one influences the other.

We are interested in the *structural* properties of information flow within a system and define relations abstracting flows, *flow relations*, as being both reflexive and transitively closed. An *information-flow component* abstracts the implementation of a system by a *flow relation*. An *information-flow interface* specifies forbidden flows in an open system by defining three kinds of constraints: assumptions, guarantees, and properties. The *assumption* characterizes flows that we assume are not part of the environment while the *guarantee* describes all flows the system forbids and that are local to it. The *property* qualifies the forbidden flows at the interaction between the system and its environment. Hence, it represents a requirement on the closed system that needs to be enforced by guarantees on the open system and assumptions on its environment.

Definition 1. *Let X and Y be disjoint sets of* input *and* output *variables, respectively, with $Z = X \cup Y$ the set of all variables. A* stateless information-flow component *is a tuple $(X, Y, \mathcal{M})$, where $\mathcal{M} \subseteq Z \times Y$ is a (reflexive and transitive) flow relation, called* flows*. A* stateless information-flow interface *is a tuple $(X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$, where: $\mathcal{A} \subseteq Z \times X$ is a relation, called* assumption*; $\mathcal{G} \subseteq Z \times Y$ is a relation, called* guarantee*; and $\mathcal{P} \subseteq Z \times Y$ is a relation, called* property*.*

Given an interface F we are interested in components that do not implement flows forbidden by either the interface guarantees (called *implementations of F*) or the interface assumptions (called *environments of F*).

Definition 2. *A component $f_{\mathcal{E}} = (Y, X, \mathcal{E})$ is called an* environment *of $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$. An environment is* admissible for F*, denoted by $f_{\mathcal{E}} \models F$, iff $\mathcal{E} \subseteq \overline{\mathcal{A}}$. A component $f = (X, Y, \mathcal{M})$* implements *the interface F, denoted by $f \models F$, iff $\mathcal{M} \subseteq \overline{\mathcal{G}}$.*

Example 1.

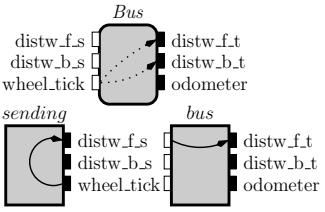


Fig. 4: Interface *Bus* with an implementation, *bus*, and an admissible environment, *sending*.

In Figure 4, we have the first refinement of the interface *Bus* from our application example. The *Bus* interface specifies the requirement on the closed system (using properties) that there are no-flows from *wheel_tick* to both *distw_f_s* and *distw_b_s*. The *Bus* interface specifies this requirement as a guarantee on the open system, too. Then, the *bus* component is an implementation of *Bus* because it has only a flow from *distw_f_s* to *distw_f_t*, which is not in the guarantees of the *Bus* interface. *Bus* does not have any assumptions, then the *sending* component is an environment for *Bus*.

When we compose the components *sending* and *bus*, there is a flow from *wheel_tick* to *distw_f_t*, which is in the properties of the *Bus*. Hence the assumption and guarantee specified over the open system are not enough to ensure the property over the closed system. The composition of these two components witness that the *Bus* interface is not well-formed.

An information-flow interface is *well-formed* when it has at least one implementation and one admissible environment. Therefore, all of its relations must be irreflexive. We refer to irreflexive relations as *no-flow* relations. A well-formed interface ensures, additionally, that an interface property is *consistent* with its assumptions and guarantees. An interface property is not consistent when the flow relation defined by the composition of one of the interface's admissible environments with one of its implementations includes a pair specified in the interface property. To check whether the property is consistent, we compute the *flow relation of the closed system defined by an interface* F, which includes all flows that are in the composition of any of the interface's admissible environments with one of its implementations. The main challenge is that, in general, the complement of an interface's guarantee (assumption) may not define the flow relation of any of its implementations (environments). Hence there may be no maximal implementation or admissible environment for a given interface.

Example 2.

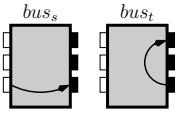


Fig. 5: *Bus* implementations.

In Figure 5, we have two components, bus_s and bus_t, that implement the interface *Bus* from the previous example. A maximal implementation of *Bus* must include the flows in both bus_s and bus_t. As flows are transitively closed, the maximal implementation would include a flow from *wheel_tick* to $distw\_f\_t$, which violates the *Bus* guarantees.

Given that we do not have maximal implementations and maximal admissible environments, then we cannot characterize all flows of the closed system defined by an interface F by computing the transitive closure of all pairs in the complement of F's assumption and guarantee – $(\overline{\mathcal{A}} \cup \overline{\mathcal{G}})^*$. This approach would yield more flows than the flows of the closed system defined by F. Instead, we consider all pairs (z, z') such that there exists a path from z to z' that alternates between flows in the complement of the assumption, $\overline{\mathcal{A}}$, and the guarantee, $\overline{\mathcal{G}}$. We define this notion below as the *composition between no-flow relations*. In Proposition 1 we prove that this definition captures our intended relation between an interface property and its environments and implementations.

Definition 3. *A* no-flow relation $\mathcal{N} \subseteq (A \cup B) \times B$ *is an irreflexive relation. and its* complement *is* $\overline{\mathcal{N}} = ((A \cup B) \times B) \setminus \mathcal{N}$. *Let* $\mathcal{N} \subseteq (A \cup B) \times B$ *and* $\mathcal{N}' \subseteq (A' \cup B') \times B'$ *be two no-flow relations. The* set of flows defined by their composition *is* $\mathcal{N} \bullet \mathcal{N}' = (Id_{A' \cup B'} \cup \overline{\mathcal{N}}) \circ (\overline{\mathcal{N}'} \circ \overline{\mathcal{N}})^* \circ (Id_B \cup \overline{\mathcal{N}'})$, *where* $Id_Z = \{(z, z) \mid z \in Z\}$ *and* $R \circ R' = \{(z, z'') \mid (z, z') \in R \text{ and } (z', z'') \in R'\}$ *is the usual composition between relations.*

We have now all the ingredients to define well-formed interfaces.

Definition 4. *An interface* $(X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ *is* well-formed *iff* $\mathcal{A}$, $\mathcal{G}$ *and* $\mathcal{P}$ *are* no-flow relations*; and the property is consistent, i.e.* $(\mathcal{A} \bullet \mathcal{G}) \cap \mathcal{P} = \emptyset$.

Proposition 1. *For all well-formed interfaces* $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$, *and for all components* $f = (X, Y, \mathcal{M})$ *and* $f_{\mathcal{E}} = (Y, X, \mathcal{E})$*: if* f *implements* F, $f \models F$, *and*

$f_{\mathcal{E}}$ is an admissible environment of F, $f_{\mathcal{E}} \models F$, then their combined flows are consistent with the property of F, $(\mathcal{M} \cup \mathcal{E})^* \cap \mathcal{P} = \emptyset$.

3.1 Composition and Incremental Design

We now present how to *compose* components and interfaces. We introduce a *compatibility* predicate that checks whether the composition of two interfaces is a well-formed interface. We prove that these two notions support the *incremental design* of systems.

The different types of variables between interfaces F and F' are defined as $Y_{F,F'} = Y \cup Y'$, $X_{F,F'} = (X \cup X') \setminus Y_{F,F'}$, and $Z_{F,F'} = Y_{F,F'} \cup X_{F,F'}$. The same definition applies to components f and f'. The *composition of components* f and f' is the reflexive and transitive closure of the union of the individual component flows, i.e. $f \otimes f' = (X_{f,f'}, Y_{f,f'}, (\mathcal{M} \cup \mathcal{M}')^*)$. We present interface composition by defining separately $\mathcal{A}$, $\mathcal{G}$ and $\mathcal{P}$ of the composed interface.

We compose interfaces through their shared variables. *Shared variables between two interfaces* are all variables that are an input variable in one of the interfaces and an output variable in the other one. The *composite flows* between two interfaces is the set with all flows that are in the composition of any of their implementations. As for the definition of flows in the closed system defined by an interface, the composite flows are the composition of the guarantees of the interfaces being composed (as defined in Definition 3). The composition of two interfaces should not restrict their sets of implementations, thus the *composite guarantees* are the complement of the composite flows.

Definition 5. *Let $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ and $F' = (X', Y', \mathcal{A}', \mathcal{G}', \mathcal{P}')$ be two interfaces. Their* composite flows *are $\overline{\mathcal{G}}_{F,F'} = \mathcal{G} \bullet \mathcal{G}'$. The* composite guarantees *of F and F' are defined as $\mathcal{G}_{F,F'} = (Z_{F,F'} \times Y_{F,F'}) \setminus \overline{\mathcal{G}}_{F,F'}$, also denoted by $\mathcal{G}_{F \otimes F'}$.*

The assumption of an interface resulting from the composition of multiple interfaces is the weakest condition on the environment that allows the interfaces being composed to work together. Additionally, it must support incremental design, i.e. the admissibility of an environment must be independent of the order in which the interfaces are composed.

Naturally, all assumptions of each interface must be considered during composition. However, not all of them can be kept as assumptions of the composite interface, because shared variables will be output variables of the composition. If the environment can still influence the information flow to a shared variable, then we may need to add assumptions to prevent such a flow. *Propagated assumptions* between two interfaces are derived by looking in their respective assumptions for no-flow pairs pointing to a shared variable.

Example 3.

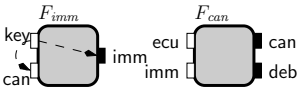


Fig. 6: Propagating assumptions.

In Figure 6, we depict an interface specifying information-flow policies for a car immobilizer, F_{imm}, along with an interface for a Controller Area Network (CAN bus), F_{can}. Interface F_{imm} has only one assumption that key does not flow to can. In this design, the immobilizer uses the CAN to communicate with the car electronic control unit (ECU). Our goal is to compose both interfaces. These interfaces share the port can. Thus, can will be an output port of their composition. The interface F_{can} cannot guarantee that the only assumption in F_{imm} is satisfied after composition because it does not have a port key. As we are working with open systems and assume that the environment is helpful, we can add further assumptions to ensure the correctness of this composition. For example, we can add assumptions that prevent key from flowing to an input port in F_{can} that can flow to can. Such flows could be part of a flow from key to can, which would violate the assumption we want to enforce. In this case, we note that in F_{can} information in ecu can flow to can. So, the composite interface needs to include the assumption that key does not flow to ecu. This is a *propagated assumption*.

Definition 6. *The set of* assumptions propagated from $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ to $F' = (X', Y', \mathcal{A}', \mathcal{G}', \mathcal{P}')$ *is* $\hat{\mathcal{A}}_{F \to F'} = \{(z, z') \mid \exists s \in X \cap Y' \text{ s.t. } (z, s) \in \mathcal{A} \text{ and } (z', s) \in \overline{\mathcal{G}}_{F,F'}\}$. *The* set with all propagated assumptions *of* F *and* F' *is* $\hat{\mathcal{A}}_{F,F'} = \hat{\mathcal{A}}_{F \to F'} \cup \hat{\mathcal{A}}_{F' \to F}$. *The* composite assumptions *of* F *and* F' *are defined as* $\mathcal{A}_{F,F'} = (\mathcal{A} \cup \mathcal{A}' \cup \hat{\mathcal{A}}_{F,F'}) \cap (Z_{F,F'} \times X_{F,F'})$, *also denoted by* $\mathcal{A}_{F \otimes F'}$.

Example 4. From the example before, information from the ports ecu, imm and deb can all flow to can. So, they are flows in the composite interface and, by Definition 5, $\{(\mathsf{ecu}, \mathsf{can}), (\mathsf{imm}, \mathsf{can}), (\mathsf{deb}, \mathsf{can})\} \subseteq \overline{\mathcal{G}}_{F_{imm}, F_{can}}$. Then, $\hat{\mathcal{A}}_{F_{imm} \to F_{can}} = \{(\mathsf{key}, \mathsf{ecu}), (\mathsf{key}, \mathsf{imm}), (\mathsf{key}, \mathsf{deb})\}$. From those assumptions only $(\mathsf{key}, \mathsf{ecu})$ points to a variable in $X_{F,F'}$, so $\mathcal{A}_{F_{imm}, F_{can}} = \{(\mathsf{key}, \mathsf{ecu})\}$.

The properties of the composition contains all properties of each interface being composed. They include, additionally, all derived properties from the assumptions and guarantees of the composite. *Derived properties* are guarantees that hold under any admissible environment. They are defined by all pairs (z, y) in an interface guarantee s.t. there is no combination of flows allowed by its assumptions and guarantees that creates a flow from z to y. Then, the *derived properties* of an assumption $\mathcal{A}$ and guarantee $\mathcal{G}$ is defined as $\mathcal{P}^{\mathcal{A},\mathcal{G}} = \mathcal{G} \setminus (\mathcal{A} \bullet \mathcal{G})$. The *composite properties* of F and F' are $\mathcal{P}_{F,F'} = \mathcal{P} \cup \mathcal{P}' \cup \mathcal{P}^{\mathcal{A}_{F,F'}, \mathcal{G}_{F,F'}}$.

Definition 7. *The* composition *of two interfaces* F *and* F *is the interface:* $F \otimes F = (X_{F,F'}, Y_{F,F'}, \mathcal{A}_{F,F'}, \mathcal{G}_{F,F'}, \mathcal{P}_{F,F'})$, *where* $\mathcal{A}_{F,F'}$ *is defined in Definition 6,* $\mathcal{G}_{F,F'}$ *defined in Definition 5 and* $\mathcal{P}_{F,F'}$ *in the previous paragraph.*

We allow composition for any two arbitrary interfaces. However, not all compositions result in a well-formed interface. We define next the notions of two

interfaces being *composable* and *compatible*. Composability imposes the syntactic restriction that both interface's output variables are disjoint. Compatibility captures the semantic requirement that whenever an interface F provides inputs to another interface F', then F' needs to include guarantees that imply the assumptions of F.

Definition 8. *Two interfaces* $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ *and* $F' = (X', Y', \mathcal{A}', \mathcal{G}', \mathcal{P}')$ *are* composable *iff* $Y \cap Y' = \emptyset$. *The interfaces* F *and* F' *are* compatible, *denoted* $F \sim F'$ *iff they are composable and* $((\mathcal{A} \cup \mathcal{A}') \cap (Z_{F,F'} \times Y_{F,F'})) \subseteq \mathcal{G}_{F,F'}$.

Clearly, both the composition operator and the compatibility relation are commutative. Additionally, we prove that composition preserves well-formedness and that it supports incremental design of systems. The full proofs are in the appendix.

Theorem 1. *Let* F *and* F' *be well-formed interfaces. If the interfaces are compatible,* $F \sim F'$, *then their composition,* $F \otimes F'$, *defines a well-formed interface.*

Theorem 2 (Incremental design). *Let* F, F' *and* F'' *be interfaces. If* $F \sim F'$ *and* $(F \otimes F') \sim F''$, *then* $F' \sim F''$ *and* $F \sim (F' \otimes F'')$.

Proof. We proved first that composite assumptions are associative. We assume that $F \sim F'$ and $(F \otimes F') \sim F''$. The most interesting case is when (z, s) is an assumption of F and s is a shared variable between F and $F' \otimes F''$. Then, we need to prove that $(z, s) \in \mathcal{G}_{F,F' \otimes F''}$. We prove this by assuming towards a contradiction that $(z, s) \in \overline{\mathcal{G}}_{F,F' \otimes F''}$. We illustrate it in Figure 7.

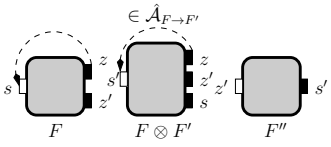


Fig. 7: Incremental design.

By composite flows being associative, $(z, s) \in \overline{\mathcal{G}}_{F \otimes F', F''}$. By (z, s) being an assumption of F and $(s', s) \in \overline{\mathcal{G}}_{F \otimes F'}$, then we have the derived assumption $(z, s') \in \hat{\mathcal{A}}_{F \to F'}$ and, so $(z, s') \in \mathcal{A}_{F \otimes F'}$. Moreover, $(z, s') \in \overline{\mathcal{G}}_{F \otimes F', F''}$, because z can flow to s' when $F \otimes F'$ is composed with F''. This contradicts our initial assumption that $(F \otimes F') \sim F''$. □

We prove additionally that composition is associative for compatible interfaces.

Theorem 3. *If* $F \sim F'$ *and* $F \otimes F' \sim F''$, *then* $(F \otimes F') \otimes F'' = F \otimes (F' \otimes F'')$.

Finally, we show that flows resulting from the composition of any components that implement two given interfaces are allowed by the composition of these interfaces.

Proposition 2. *For all two interfaces* F *and* F', *and all two components* $f = (X, Y, \mathcal{M})$ *and* $f' = (X', Y', \mathcal{M}')$ *that implement them,* $f \models F$ *and* $f' \models F'$, *then the composition of the components implements the composition of the interfaces,* $f \otimes f' \models F \otimes F'$.

3.2 Refinement and Independent Implementability

We now define a refinement relation between interfaces. Intuitively, an interface F' refines F iff F' admits more environments than F, while possibly constraining its implementations.

Definition 9. *Interface* $F' = (X', Y', \mathcal{A}', \mathcal{G}', \mathcal{P}')$ refines $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$, *written* $F' \preceq F$, *when* $\mathcal{A}' \subseteq \mathcal{A}$, $\mathcal{G} \subseteq \mathcal{G}'$ *and* $\mathcal{P} \subseteq \mathcal{P}'$.

Let F and F' be interfaces s.t. $F' \preceq F$. Let $f = (X, Y, \mathcal{M})$ and $f_{\mathcal{E}} = (Y, X, \mathcal{E})$ be components. Then, (a) If $f \models F'$, then $f \models F$; and (b) if $f_{\mathcal{E}} \models F$, then $f_{\mathcal{E}} \models F'$.

Additionally, we show below that refinement and composition supports independent implementability.

Theorem 4 (Independent implementability). *For all well-formed interfaces* F_1', F_1 *and* F_2, *if* $F_1' \preceq F_1$ *and* $F_1 \sim F_2$, *then* $F_1' \sim F_2$ *and* $F_1' \otimes F_2 \preceq F_1 \otimes F_2$.

Proof. The challenging part is to prove that the refined composite contains all properties of the abstracted one, i.e. $\mathcal{P}_{F_1 \otimes F_2} \subseteq \mathcal{P}_{F_1' \otimes F_2}$. We prove by induction on $n \in \mathbb{N}$ that if a pair of variables (z, y) cannot be defined by assume-guarantee paths of size at most n of the abstract composition, then it cannot be defined by assume-guarantee paths of size at most n of the refined composition. We can see easily for the base case. If for all $(z, s) \in \overline{\mathcal{A}}_{F_1, F_2}$ s.t. there exists $(s, y) \in \mathcal{G}_{F_1, F_2}$, then, by $F_1' \preceq F_1$, it follows that for all $(z, s) \in \overline{\mathcal{A}}_{F_1', F_2}$ there exists $(s, y) \in \mathcal{G}_{F_1', F_2}$. Hence if $(z, y) \notin \overline{\mathcal{A}}_{F_1, F_2} \circ \overline{\mathcal{G}}_{F_1, F_2}$, then $(z, y) \notin \overline{\mathcal{A}}_{F_1', F_2} \circ \overline{\mathcal{G}}_{F_1', F_2}$ as well. □

3.3 Discussion

Properties. In this work, we consider transitively closed flows. In this setting, in an open system, information can flow from z to z' by flowing from z to s through the environment, and then from s to z' through one of its implementations. As our algebra focuses on the design of structural requirements of no-flows in open systems, it needs to support the specification of global no-flow requirements. We made them explicit by introducing properties. If we did not include properties in our interfaces, then either assumptions or guarantees would need to take over the role of specifying global no-flows. Let's assume that, alternatively, guarantees would be interpreted as global no-flows. Then, to support incremental design, the compatibility criteria between interfaces would turn out to be overly restrictive, with intuitive and correct designs being considered incompatible. This led us to the distinction between guarantees and properties, where properties may be supported by assumptions on the environment that can restrict the set of compatible interfaces. In other words, the main advantage of having properties is that the designer can choose how to split the responsibilities between the environment and the implementations to satisfy a global no-flow.

Semantics. The structural approach that abstracts away semantic considerations is an important feature of our theory. The practicability of our approach lies in the support for the design of such requirements by decoupling the design process from (its orthogonal) semantic considerations. Hence, our approach does not deny semantics, but rather separates the design of specifications from component implementation concerns. The presented approach even allows using tailored semantics and tools for different parts of the design. For example, at the bottom (component) level, no-flows and flows relations can be instantiated with different semantic interpretations. After deriving the component no-flows from the implementation under a concrete semantics, the theory can be agnostic about the underlying semantic interpretation and can focus on whether there exists a flow at all.

4 Stateful Information-Flow Interfaces

We extend our theory with stateful components and interfaces. These are transition systems in which each state is a stateless component or interface, respectively.

Definition 10. *Let X and Y be disjoint sets of* input *and* output *variables, respectively, with $Z = X \cup Y$ the set of all variables. Let Q be a set of states with $\hat{q} \in Q$ being the initial state and $\delta : Q \to 2^Q$ be a transition relation. A* stateful information-flow component $\mathbf{f}$ *is a tuple $(X, Y, Q, \hat{q}, \delta, \mathbb{M})$, where $\mathbb{M} : Q \to 2^{Z \times Y}$ is a state labeling such that for all states $q \in Q$, $\mathbb{M}(q)$ defines a flow relation. We denote by $\mathbf{f}(q) = (X, Y, \mathbb{M}(q))$ the stateless component implied by the labeling of q. A* stateful information-flow interface $\mathbb{F}$ *is a tuple $(X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$, where $\mathbb{A} : Q \to 2^{Z \times X}$ is called* assumption*; $\mathbb{G} : Q \to 2^{Z \times Y}$ is called* guarantee*; and $\mathbb{P} : Q \to 2^{Z \times Y}$ is called* property*. For each state $q \in Q$ we denote by $\mathbb{F}(q) = (X, Y, \mathbb{A}(q), \mathbb{G}(q), \mathbb{P}(q))$ the stateless interface implied by the assumption, guarantee and property of q.*

A *stateful interface* $\mathbb{F}$ is well-formed iff $\mathbb{F}(\hat{q})$ is a well-formed stateless interface, and for all $q \in Q$ reachable from $\hat{q}$ the stateless interface $\mathbb{F}(q)$ is well-formed. In what follows, $\mathbb{F} = (X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$ and $\mathbb{F}' = (X', Y', Q', \hat{q}', \delta', \mathbb{A}', \mathbb{G}', \mathbb{P}')$ are stateful interfaces, and $\mathbf{f} = (X, Y, Q_\mathbf{f}, \hat{q}_\mathbf{f}, \delta_\mathbf{f}, \mathbb{M})$ and $\mathbf{f}_\mathcal{E} = (Y, X, Q_\mathcal{E}, \hat{q}_\mathcal{E}, \delta_\mathcal{E}, \mathbb{E})$ are stateful components.

A stateful component $\mathbf{f}$ implements a stateful interface $\mathbb{F}$ if there exists a simulation relation from $\mathbf{f}$ to $\mathbb{F}$ such that the stateless components in the relation implement the stateless interfaces they are related to. Admissible environments require a simulation relation from them to the interface they are admissible on.

Definition 11. *A component* $\mathbf{f}$ implements *the interface $\mathbb{F}$, denoted by $\mathbf{f} \models \mathbb{F}$, iff there exists $H \subseteq Q_\mathbf{f} \times Q$ s.t. $(\hat{q}_\mathbf{f}, \hat{q}) \in H$ and for all $(q_\mathbf{f}, q) \in H$: (i) $\mathbf{f}(q_\mathbf{f}) \models \mathbb{F}(q)$; and (ii) if $q'_\mathbf{f} \in \delta_\mathbf{f}(q_\mathbf{f})$, then there exists a state $q' \in \delta(q)$ s.t. $(q'_\mathbf{f}, q') \in H$. A component $\mathbf{f}_\mathcal{E}$ is an* admissible environment *for the interface $\mathbb{F}$, denoted by $\mathbf{f}_\mathcal{E} \models \mathbb{F}$, iff there exists a relation $H \subseteq Q \times Q_\mathcal{E}$ s.t. $(\hat{q}, \hat{q}_\mathcal{E}) \in H$ and for all*

$(q, q_{\mathcal{E}}) \in H$: *(i)* $\mathbf{f}(q_{\mathcal{E}}) \models \mathbb{F}(q)$*; and (ii) if* $q' \in \delta_{\mathbb{F}}(q)$*, then there exists a state* $q'_{\mathcal{E}} \in \delta_{\mathcal{E}}(q_{\mathcal{E}})$ *s.t.* $(q', q'_{\mathcal{E}}) \in H$.

As for stateless interfaces, we have that interface's properties are satisfied after we compose any of its implementations $\mathbf{f}$ with any of its admissible environments $\mathbf{f}_{\mathcal{E}}$.

Proposition 3. *For all well-formed interfaces* $\mathbb{F}$*, and all relations* $H \subseteq Q_{\mathbf{f}} \times Q$ *and* $H_{\mathcal{E}} \subseteq Q \times Q_{\mathcal{E}}$ *that witness* $\mathbf{f} \models \mathbb{F}$ *and* $\mathbf{f}_{\mathcal{E}} \models \mathbb{F}$*, respectively, it holds: (i)* $(\mathbb{M}(\hat{q}_{\mathbf{f}}) \cup \mathbb{E}(\hat{q}_{\mathcal{E}}))^* \cap \mathbb{P}(\hat{q}) = \emptyset$*; and (ii) for all* $q \in Q$ *that are reachable from* $\hat{q}$*, if* $(q_{\mathbf{f}}, q) \in H$ *and* $(q, q_{\mathcal{E}}) \in H_{\mathcal{E}}$*, then* $(\mathbb{M}(q_{\mathbf{f}}) \cup \mathbb{E}(q_{\mathcal{E}}))^* \cap \mathbb{P}(q) = \emptyset$.

Composition of two components is defined as their synchronous product. The composition of two interfaces is defined as their synchronous product, as well. However, we only keep the states that are defined by the composition of two compatible stateless interfaces.

Definition 12. *Let* $\mathbb{F}$ *and* $\mathbb{F}'$ *be two interfaces. Their* composition *is defined as the tuple:* $\mathbb{F} \otimes \mathbb{F}' = (X_{\mathbb{F},\mathbb{F}'}, Y_{\mathbb{F},\mathbb{F}'}, Q_{\mathbb{F},\mathbb{F}'}, \hat{q}_{\mathbb{F},\mathbb{F}'}, \delta_{\mathbb{F},\mathbb{F}'}, \mathbb{A}_{\mathbb{F},\mathbb{F}'}, \mathbb{G}_{\mathbb{F},\mathbb{F}'}, \mathbb{P}_{\mathbb{F},\mathbb{F}'})$*, where:* $\hat{q}_{\mathbb{F},\mathbb{F}'} = (\hat{q}, \hat{q}')$ *and* $Q_{\mathbb{F},\mathbb{F}'} = \{\hat{q}_{\mathbb{F},\mathbb{F}'}\} \cup \{(q, q') \mid \mathbb{F}(q) \sim \mathbb{F}'(q')\}$*;* $(q_2, q'_2) \in \delta_{\mathbb{F},\mathbb{F}'}(q_1, q'_1)$ *iff* $q_2 \in \delta(q_1)$ *and* $q'_2 \in \delta'(q'_1)$*; for all* $(q, q') \in Q_{\mathbb{F},\mathbb{F}'}$: $\mathbb{F}_{\mathbb{F},\mathbb{F}'}(q, q') = \mathbb{F}(q) \otimes \mathbb{F}'(q')$.

Two stateful interfaces are compatible if the stateless interfaces defined by their initial states are compatible, i.e. $\mathbb{F}(\hat{q}) \sim \mathbb{F}'(\hat{q}')$. It follows from the results proved for the stateless interfaces that compatibility is commutative, composition preserves well-formedness and stateful interfaces support incremental design.

Proposition 4. *If* $\mathbf{f} \models \mathbb{F}$ *and* $\mathbf{g} \models \mathbb{G}$*, then* $\mathbf{f} \otimes \mathbf{g} \models \mathbb{F} \otimes \mathbb{G}$.

Given an interface, we define transitions parameterized by no-flows on its input variables (i.e. with fixed assumptions) or on its output variables (i.e. with fixed guarantees and properties).

Definition 13. *Let* $\mathbb{F}$ *be an interface.* Input transitions *from a given state* $q \in Q$ *are defined as* $\delta^X(q) = \{\delta^X(q, \mathcal{A}) \mid \mathcal{A} \subseteq Z \times X\}$ *with* $\delta^X(q, \mathcal{A}) = \{q' \in \delta(q) \mid \mathbb{A}(q') = \mathcal{A}\}$. Output transitions *from a given state* $q \in Q$ *are defined as* $\delta^Y(q) = \{\delta^Y(q, \mathcal{G}, \mathcal{P}) \mid \mathcal{G} \subseteq Z \times Y$ *and* $\mathcal{P} \subseteq Z \times Y\}$ *with* $\delta^Y(q, \mathcal{G}, \mathcal{P}) = \{q' \in \delta(q) \mid \mathbb{G}(q') = \mathcal{G}$ *and* $\mathbb{P}(q') = \mathcal{P}\}$.

Interface $\mathbb{F}_R$ refines $\mathbb{F}_A$, if all output steps of $\mathbb{F}_R$ can be simulated by $\mathbb{F}_A$, while all input steps of $\mathbb{F}_A$ can be simulated by $\mathbb{F}_R$. This corresponds to alternating refinement [5].

Definition 14. *Interface* $\mathbb{F}_R$ refines $\mathbb{F}_A$*, written* $\mathbb{F}_R \preceq \mathbb{F}_A$*, iff there exists a relation* $H \subseteq Q_R \times Q_A$ *s.t.* $(\hat{q}_R, \hat{q}_A) \in H$ *and for all* $(q_R, q_A) \in H$*: (i)* $\mathbb{F}_R(q_R) \preceq \mathbb{F}_A(q_A)$*; (ii) for all set of states* $O \in \delta^Y_R(q_R)$*, there exists* $O' \in \delta^Y_A(q_A)$ *s.t. for all set of states* $I' \in \delta^X_A(q_A)$*, there exists* $I \in \delta^X_R(q_R)$ *s.t.* $(O \cap I) \times (O' \cap I') \subseteq H$.

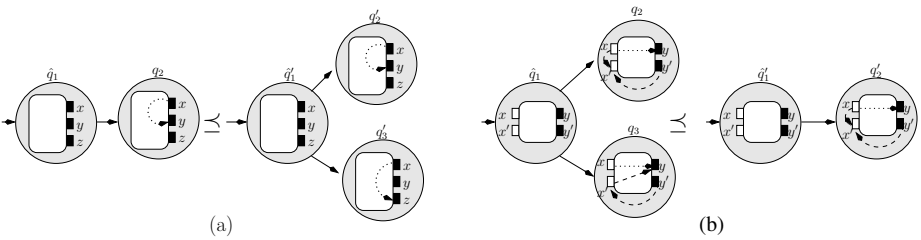


Fig. 8: Refined interfaces with witness: (a) relation $\{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2)\}$; and (b) relation $\{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2), (q_3, q'_2)\}$.

Example 5. In Figure 8 we depict two examples of refined stateful interfaces.

In Figure 8(a) the stateless interface in each state only uses output ports and it only specifies properties. The initial state of both stateful interfaces is the same, so they clearly refine each other. As there are no assumptions and guarantees, then, by Definition 14, we need to check that for all successors of the initial state in the refined interface q_s, there exists a successor of the initial state in the abstract interface q'_s such that $\mathbb{P}_A(q'_s) \subseteq \mathbb{P}_R(q_s)$. This holds for the states (q_2, q'_2). Hence the relation $\{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2)\}$ witnesses the refinement. Note that the refined interface is obtained by removing a nondeterministic choice on the transition function.

The witness relation for the refinement depicted in Figure 8(b) is $\{(\hat{q}_1, \hat{q}'_1)$, $(q_2, q'_2), (q_3, q'_2)\}$. The initial states are the same, so the condition (i) in Definition 14 is trivially satisfied. The refined interface has two distinct *output transitions* from the initial state $\hat{q}_1$. It can either go to state q_2 by choosing the set of guarantees and proposition with only one element (x, y) or it can transition to state q_3 by committing to the set of no-flows $\{(x, y), (x', y)\}$ for the guarantees and $\{(x, y)\}$ as property. From the initial state of the abstract interface, there exists only one *input transition* possible, to assume that x does not flow to x' and y' does not flow to x. The following holds for both states accessible from the initial state in the refined interface: $\mathbb{A}_R(q_2) \subseteq \mathbb{A}_A(q'_2)$ and $\mathbb{A}_R(q_3) \subseteq \mathbb{A}_A(q'_2)$. The refined interface specifies an alternative transition from the initial state (represented by state q_3) that allows more environments while restricting the implementation and preserving the property.

Theorem 5. *Let* $\mathbb{F}' \preceq \mathbb{F}$. *(a) If* $\mathbf{f} \models \mathbb{F}'$, *then* $\mathbf{f} \models \mathbb{F}$. *(b) If* $\mathbf{f}_{\mathcal{E}} \models \mathbb{F}$, *then* $\mathbf{f}_{\mathcal{E}} \models \mathbb{F}'$.

Theorem 6 (Independent implementability). *For all well-formed interfaces* $\mathbb{F}'_1$, $\mathbb{F}_1$ *and* $\mathbb{F}_2$, *if* $\mathbb{F}'_1 \preceq \mathbb{F}_1$ *and* $\mathbb{F}_1 \sim \mathbb{F}_2$, *then* $\mathbb{F}'_1 \sim \mathbb{F}_2$ *and* $\mathbb{F}'_1 \otimes \mathbb{F}_2 \preceq \mathbb{F}_1 \otimes \mathbb{F}_2$.

The composition operation on stateful information-flow interfaces can be generalized to distinguish between compatible and incompatible transitions of interfaces when they are composed. Usually this is done by labeling transitions with letters from an alphabet, so that only transitions with the same letter can be synchronized. While necessary for practical modeling, we omit this technical generalization to allow the reader to focus on the novelty of our formalism, which

is the ability to specify information-flow constraints (environment assumptions, implementation guarantees, and global properties) at each state of an interface.

5 Related Work

To the best of our knowledge, we are the first to provide a theory for top-down and bottom-up design of information-flow system requirements that supports both incremental design and independent implementability of systems. The literature closest to our work about information-flow focus on the semantic aspects of it. The novelty of our work lies on explicit separation of the structural concerns from the semantic aspects of information-flow.

Language-based techniques have been proved useful to verify and enforce information flow policies [29]. Examples range from type systems [15] to program analysis using program-dependency graphs (PDGs) [18,16]. In our approach we aim at composition and refinement notions that are independent of the language adopted for the implementations.

Information-flow properties can be specified with respect to the observed behavior of a system, in which each of its execution runs is abstracted as a trace. In this approach, properties often compare multiple executions of a system to certify that no forbidden flow can be deduced by an observer. Such properties over multiple execution traces are called hyperproperties [12]. Temporal logics [26], like LTL or CTL* are used to specify trace properties of reactive systems. HyperLTL and HyperCTL* [11] extend temporal logics by introducing quantifiers over path variables. They allow relating multiple executions and expressing information-flow security properties [12,11]. Epistemic temporal logics (ETL) [9] provide the knowledge connective with an implicit quantification over traces. With ETL we can reason about the knowledge gain of agents over time. Then, we can specify which information can be learned by the agents while interacting with the system [6]. All these LTL extensions reason about closed systems while our approach allows compositional reasoning about open systems. Moreover, we focus here on the structural aspect of information-flow, and not yet on its semantic interpretation. Thus, all information-flow trace-based semantics are orthogonal to our approach.

Interface theories belong to the broader area of contract-based design [8], originally popularized by Meyer [24], following earlier ideas introduced by Floyd and Hoare [14,19]. Our theory follows closely the philosophy for formal frameworks for systems design introduced for Interface automata (IA) [1] and Assume/Guarantee (A/G) [2] interfaces. Interface theories were later extended with extra-functional requirements such as resource [10], timing [4,13] and security [21] requirements. Unlike in previous interface formalisms, we had to introduce the notion of *properties* which capture the intent of the designer and can be used to steer the refinement of interfaces.

Interface for structure and security (ISS) [21] is a variant of IA that enables specification of two types of actions on (1) low and (2) high confidential information. ISS uses a bisimulation-based notion of non-interference that checks

whether the system behaves in the same way when high actions are performed or when they are considered hidden actions. Our approach is orthogonal to IA and their extensions: we do not characterise the type of actions of each component, but only their input/output ports, defining explicitly the information-flow relations between variables.

Our approach took inspiration from *relational interfaces* (RIs) [31]. RIs specify the legal inputs that the environment is allowed to provide to the component along with the legal outputs that the component can generate when provided with these input. RIs do not have assumptions and guarantees defined separately. Instead, they have a contract that specifies the desired input-output behavior. A contract in RIs is expressed over individual traces. Then, an RI contract can only relate input and output values in a trace, and not across multiple traces. This restricts considerably RIs expressivity concerning information-flow properties. Besides, RIs are trace-based interfaces, while in our approach we focus on the structural aspect of information-flow, which may change from state to state (in the stateful case). Our approach can be seen as a limited way to introduce relational properties into A/G interfaces, namely solely for guiding refinement. This limited way avoids many of the technical complexities of general relational interfaces [31].

6 Conclusion

We propose a novel interface theory to specify information-flow properties. Our framework includes both *stateless* and *stateful* interfaces and supports both incremental design and independent implementability. To achieve this, unlike in previous interface formalisms, we introduce the notion of *properties* which captures the intent of the designer for the interaction between assumptions and guarantees. Moreover, properties can be used to steer the refinement of interfaces. It will be interesting to study the introduction of such design-guiding properties in the context of other interface languages.

As future work, we will explore how to extend our theory with sets of *must-flows*, i.e. support for modal specifications [27]. This will enable, for example, to specify flows that a state q must implement so that the system can transition to a different state, which is useful to specify declassification of information. Another direction is to explore trace semantics for our interfaces.

References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: European Software Engineering Conference/Foundations on Software Engineering (ESEC/FSE). p. 109120. ACM (2001). https://doi.org/10.1145/503209.503226
2. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Embedded Software. LNCS, vol. 2211, pp. 148–165. Springer (2001). https://doi.org/10.1007/3-540-45449-7_11

3. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Engineering Theories of Software Intensive Systems. NATO Science Series (Series II: Mathematics, Physics and Chemistry), vol. 195, pp. 83–104. Springer Netherlands (2005). https://doi.org/10.1007/1-4020-3532-2_3
4. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In: Embedded Software. LNCS, vol. 2491, pp. 108–122. Springer (2002). https://doi.org/10.1007/3-540-45828-X_9
5. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: CONCUR'98 Concurrency Theory. LNCS, vol. 1466, pp. 163–178. Springer (1998). https://doi.org/10.1007/BFb0055622
6. Balliu, M., Dam, M., Le Guernic, G.: Epistemic temporal logic for information flow security. In: Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS). pp. 1–12. ACM (2011). https://doi.org/10.1145/2166956.2166962
7. Benadjila, R., Renard, M., Lopes-Esteves, J., Kasmi, C.: One car, two frames: attacks on hitag-2 remote keyless entry systems revisited. In: 11th USENIX Workshop on Offensive Technologies (2017)
8. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J., Reinkemeier, P., Sangiovanni-Vincentelli, A.L., Damm, W., Henzinger, T.A., Larsen, K.G.: Contracts for system design. Foundations and Trends in Electronic Design Automation **12**(2-3), 124–400 (2018). https://doi.org/10.1561/1000000053
9. Bozzelli, L., Maubert, B., Pinchinat, S.: Unifying hyper and epistemic temporal logics. In: Foundations of Software Science and Computation Structures (FoSSaCS). LNCS, vol. 9034, pp. 167–182. Springer (2015). https://doi.org/10.1007/978-3-662-46678-0_11
10. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Embedded Software. LNCS, vol. 2855, pp. 117–133. Springer (2003). https://doi.org/10.1007/978-3-540-45212-6_9
11. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Principles of Security and Trust (POST). LNCS, vol. 8414, pp. 265–284. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15
12. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393
13. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC). pp. 91–100. ACM (2010). https://doi.org/10.1145/1755952.1755967
14. Floyd, R.W.: Assigning meanings to programs. Proceedings of Symposium on Applied Mathematics **19**, 19–32 (1967). https://doi.org/10.1007/978-94-011-1793-7_4
15. Focardi, R., Maffei, M.: Types for security protocols. Formal Models and Techniques for Analyzing Security Protocols **5**, 143–181 (2011). https://doi.org/10.3233/978-1-60750-714-7-143
16. Graf, J., Hecker, M., Mohr, M.: Using JOANA for information flow control in Java programs - a practical guide. In: Software Engineering 2013 - Workshopband. LNI, vol. P-215, pp. 123–138. Gesellschaft für Informatik e.V. (2013), https://dl.gi.de/20.500.12116/17361
17. Hamilton, M.D., Tunstall, M., Popovici, E.M., Marnane, W.P.: Side channel analysis of an automotive microprocessor. In: IET Irish Signals and Systems

Conference (ISSC). pp. 4–9. Institution of Engineering and Technology (2008). https://doi.org/10.1049/cp:20080630
18. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security **8**(6), 399–422 (2009). https://doi.org/10.1007/s10207-009-0086-1
19. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
20. Larsen, K.G., Nyman, U., Wasowski, A.: Interface input/output automata. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) International Symposium on Formal Methods (FM). LNCS, vol. 4085, pp. 82–97. Springer (2006). https://doi.org/10.1007/11813040_7
21. Lee, M., D'Argenio, P.R.: Describing secure interfaces with interface automata. Electronic Notes in Theoretical Computer Science **264**(1), 107–123 (2010). https://doi.org/10.1016/j.entcs.2010.07.008
22. Mantel, H.: On the composition of secure systems. In: IEEE Symposium on Security and Privacy. pp. 88–101. IEEE Computer Society (2002). https://doi.org/10.1109/SECPRI.2002.1004364
23. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: IEEE Computer Security Foundations Symposium (CSF). pp. 218–232. IEEE (2011). https://doi.org/10.1109/CSF.2011.22
24. Meyer, B.: Applying 'design by contract'. Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
25. Mikulcak, M., Herber, P., Göthel, T., Glesner, S.: Information flow analysis of combined simulink/stateflow models. Information Technology And Control **48**(2), 299–315 (2019). https://doi.org/10.5755/j01.itc.48.2.21759
26. Pnueli, A.: The temporal logic of programs. In: Annual Symposium on Foundations of Computer Science (FOCS). pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
27. Raclet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: A modal interface theory for component-based design. Fundamenta Informaticae **108**(1-2), 119–149 (2011). https://doi.org/10.3233/FI-2011-416
28. Ratasich, D., Khalid, F., Geissler, F., Grosu, R., Shafique, M., Bartocci, E.: A roadmap toward the resilient internet of things for cyber-physical systems. IEEE Access **7**, 13260–13283 (2019). https://doi.org/10.1109/ACCESS.2019.2891969
29. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121
30. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security **3**(1), 30–50 (2000). https://doi.org/10.1145/353323.353382
31. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM Transactions on Programming Languages and Systems (TOPLAS) **33**(4), 14 (2011). https://doi.org/10.1145/1985342.1985345
32. Verdult, R., Garcia, F.D., Balasch, J.: Gone in 360 seconds: Hijacking with hitag2. In: 21st USENIX Security Symposium. pp. 237–252 (2012)

A Survey-driven Feature Model for Software Traceability Approaches

Edouard Romari Batot✉[1], Sebastien Gérard[2], and Jordi Cabot[1,3]

[1] SOM-IN3 - Universitat Oberta de Catalunya, Barcelona, España – {ebatot,jcabot}@uoc.edu
[2] CEA LIST, Paris France – sebastien.gerard@cea.fr
[3] ICREA, Barcelona, España – jordi.cabot@icrea.cat

Abstract. Traceability is the capability to represent, understand and analyze the relationships between software artefacts. Traceability is at the core of many software engineering activities. This is a blessing in disguise as traceability research is scattered among various research subfields, which impairs a global view and integration of the different innovations around the recording, identification, evaluation and management of traces. This also limits the adoption of traceability solutions in industry.
In this sense, the goal of this paper is to present a characterization of the traceability mechanism as a feature model depicting the shared and variable elements in any traceability proposal. The features in the model are derived from a survey of papers related to traceability published in the literature. We believe this feature model is useful to assess and compare different proposals and provide a common terminology and background. Beyond the feature model, the survey we conducted also help us to identify a number of challenges to be solved in order to move traceability forward, especially in a context where, due to the increasing importance of AI techniques in Software Engineering, traces are more important than ever in order to be able to reproduce and explain AI decisions.

1 Introduction

The need for traceability has always been salient in software and systems development. Across the years, there has been a continuous interest in developing techniques to facilitate the representation and analysis of traces and links between related artefacts. It helps explaining their execution and evolution as required in many software engineering activities and disciplines such as code-generation, program understanding, software maintenance, and debugging.

The importance of traceability was first recognized in system engineering, especially related to the development and certification of critical systems where it is a primary concern. As an example, traceability is part of any certification mechanism in all commercial software-based aerospace systems as stated in documents like the RTCA DO-178C (2012) [76,62]. The consideration of various levels of abstraction in software development and the meaning of verification in model-based development paradigm – which figures abstract representations (models) as the core artefact for conceptualization – was later introduced with companion documents (specifically, DO-331). The

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 23–48, 2022.
https://doi.org/10.1007/978-3-030-99429-7_2

automotive industry has followed the same path with the construction of an international standard for functional safety, the ISO-26262 [46].

Despite these important evidences on the need for explicit (and automated) tracing abilities in software development, traceability is not widely adopted, even less automated. There is little feedback from its concrete use in industry beyond the critical domains above [75] and when existing, it ends up being mostly a manual process [55]. Moreover, with no standard definition or representation of traces, it is difficult to bridge the gaps between the different partial traceability solutions existing in research subfields [4,102,101]. Even the software engineering body of knowledge does not seem to properly consider the power of traceability as it only *mentions* traceability once [16].

The foundation for an effective modelling of traceability is disseminated among a profuse literature. Approaches vary greatly in their means and goals. Moreover, most focus on specific pairs of artefacts and therefore remain difficult to integrate in different industrial scenarios. Note also that this happens in a context where artificial intelligence techniques are being integrated in development processes, raising the need for more powerful reproducibility and explainability concerns, both requiring the assistance of traceability mechanisms.

This paper aims to provide a comprehensive perspective on the state of the art of traceability techniques in software development and their limitations. With the short-term goal of facilitating the evaluation and comparison of current solutions. And the mid-term goal of accelerating the development of new traceability solutions that could benefit from the existing ones thanks to our new conceptualization in the form of a feature model describing the potential dimensions and concerns a traceability solution may wish to consider. We do not create the feature model only based on our (partial) knowledge and expertise in the domain. Instead, we ground our classification with a survey of the published literature in this field. According to this survey, we group the traceability features in three main dimensions: trace definition, trace identification and trace management, with the corresponding feature hierarchies for each of them.

The paper is organized as follows. After a brief introduction, we discuss in Section 2 an overview of the scientific work related to traceability. We then remind some basic terminology in Section 3. Section 4 describes how we conducted our literature review and Section 5 presents a detailed feature model derived from the survey of the retrieved works. This analysis also helps us to propose a number of discussion points and open challenges in Section 6 before concluding this work.

2 State of the art of software traceability

Traceability was proposed, from the very beginning of software engineering, to ensure that a system being developed actually reflects its design. Already in the original NATO working conference, quality projects were praised for making "the system that they are designing contain explicit *traces* of the design process" [81]. From that point on, traceability has been studied from a myriad of perspectives, dimensions and applications.

Historically, traceability historically started in requirement engineering. The very idea to follow the impact of changes in the requirements to other artefacts (and backward) was then and remains today the most prominent goal [35]. Precise and rich re-

quirements allow a proper follow up of their later implementations [21]. Through time, the advantages of using traces – *i.e.,* the record of (inter-)dependencies between artefacts, has revealed to be applicable to most if not all sphere of software maintenance. The use of traces spans from software certification and testing, feature location, debugging, code generation, and so on. With the proliferation of traceability purposes, some authors explicitly asked for better sharing of experiences in using traceability [36] and evaluating the solutions existing so far [91]. Surveys and literature reviews trying to group and compare them began to appear as well, though most of them focused on specific subareas such as requirement engineering [35,15], model-driven development [32,101,70,86,63], software product lines [96,3], benchmarking [91], and information retrieval [23,13,39]. To complement these scientific surveys, Konigs *et al.* survey industrial application of traceability approaches, showing its limited penetration [52]. Neumuller *et al.* show that the adoption is worse in small businesses where traceability is even less automated [67]. Finally, Charalampidou *et al.* add to the conclusion of other surveys that "although many studies include some empirical validation", there is still much to be done with respect to validation and reproducibility [20].

This is aggravated by the fact that, as pointed out above, many of the proposals belong to different research subfields, which limits the discovery and awareness of alternative solutions. For instance, authors point out that researchers in requirement engineering and in model-based development do not communicate enough among each others [101,70,85]. This lack of communication and shared understanding is one of the open challenges in the traceability domain [22,4,28]. To solve this issue, several works aim at proposing specific traceability models. Unfortunately, many investigations suffer a lack of generalizability due the specific nature of the problem being solved (*e.g.,* certification conformity [51], model transformation coevolution [38]), or the specific nature of the solution considered (*e.g.,* w.r.t. its language: SysML [65], w.r.t. its engineering field: SPL [3], agile [60]).

As an example, the automatic identification of trace links is one of the most studied features. There are plenty of proposals but as they are evaluated using different datasets and configurations, they cannot be directly compared [89,39,13]. Another example would be model-driven engineering, where the use of traceability specific languages together with automated model transformation appears as an ideal soil to grow end-to-end traceability. This led authors to present classifications and terminologies for a systematic perspective on the tracing of MDE development [70,28,85]. Nevertheless, proposals tend to focus on a specific model-driven engineering problem: the co-evolution of models and transformations [2] instead of aiming for more general solutions. Mustafa *et al.* argue that "the main issues in traceability nowadays are building traceability models that can accommodate the capturing of traceability information and providing common semantics for trace links" [63]. As a result of this confusing situation, authors asked for more standardized practices. Two proposals gather terminology for fundamental and model based terminology [36,45]. We take our general knowledge about traceability from them and add to their definitions an actionable categorization for existing and coming traceability approaches.

We agree with these authors that this lack of *de juro / de facto* standard is hampering the benefits of current solutions and hindering evolution in the field. This paper intends

to cover this gap by proposing a traceability characterization that stems from the analysis of existing proposals. We believe this model can be useful to researchers trying to improve traceability techniques in any subfield and to practitioners looking for a way to compare and choose the traceability solution that best suits their needs.

3 Towards a common traceablility terminology

A clear conclusion from the previous section is the lack of a common agreed upon conceptualization for traceability that helps evaluating, comparing and reusing traceability solutions over a variety of scenarios and application domains. Thus, the *incoherency problem* still arises in traceability research [100]. Even if an individual article makes a claim that withstood rigorous testing and statistical analysis, it might not use the same words as an adjacent article, or it would use the same words but intend different meanings. For instance, the term *traceability* is used to designate both the ability to trace system elements, and the traceability links (the relations) themselves [15,4].

Therefore, before proposing our global traceability feature model to classify traceability solutions, we first recap the different usages of the key traceability concepts and propose a unified definition that we will use in the rest of the paper.

3.1 Traceability components

Traceability research refers mainly to a definition from Gotel *et al.* that defines traceability as the ability to describe and follow the life-cycle of a requirement, from its initial specification to the design and code elements of the system implementing it [35]. This is still the most popular meaning for traceability [15,7] even if modeling approaches try to generalize this notion by seeing traceability as a valuable tool to link all types of linking artefacts at either the same or different levels of abstraction [56,95].

Regardless of the specific interpretation of traceability, we observe a division of knowledge into four main areas:

- **Strategizing traceability**. It involves defining the explicit traceability purpose for the project at hand and how to best reach that goal. Maro *et al.* address the importance of a coherent strategy. The authors propose an introductory methodology to "provide support for establishing a traceability strategy that allows the organization to achieve its goals and measure the impact of [its] traceability strategy" [60].
- **Trace and artefact representation**. It covers the design / adaptation of a language to be used to define the traces and decisions regarding its syntax, expressiveness, variability, integration, etc. For instance, this can be done by means of creating a full traceability domain-specific language.
- **Trace link identification**. It designates the identification of traces in a software system, be it a post-requirement assisted elicitation, a live record during a system execution or an automatic AI-based inference process. This latter approach is the motto right now to help the identification of links between heterogeneous artefacts.
- **Trace management**. It refers to the ways to use and maintain the traces. This includes tool support for the persistence, retrieval, and analysis of traces.

The first area is a high-level concern that influences the requirements of the other three to cover the specific needs of a project. These three will therefore be used to structure our feature model later on. Note that the representation component should be part of any traceability solution as it is the base component to be able to, at the very least, express traceability information.

3.2 Traceability glossary

We propose some general definitions for the most frequently encountered traceability terms while searching for and studying solutions for traceability in any of the above categories. These definitions, mostly borrowed from past literature [36,45], aim to encompass the different uses and dimensions of traceability depicted above. Our set of terms is not exhaustive but provide a common core generic enough to be then adapted to specific scenarios. This is also why we try to be precise with the definitions, while also offering room for slightly different (but compatible) interpretations.

- **Traceability** is the ability to trace different artefacts of a system (of systems). Gotel *et al.* define traceability as "requirements traceability [which] refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction" [35]. Gotel's definition has been extended to MDE software traceability as "any relationship that exists between artifacts involved in the software engineering life cycle" [1].
- A **trace** is a path from one artefact to another. A trace is composed of atomic **trace links** that directly relate artefacts to each others. The representation of traces, their data structure and behaviour, is defined in a traceability grammar or metamodel [25] depending on how the trace language is defined. In any case, the language definition specifies the concepts and relationships available to define traces. As discussed before, no standard language has emerged yet.
- An **artefact** can be any element of a system - *e.g.,* unstructured documentation, source code, design diagrams, test cases and suites... The nature of artefacts follows two main dimensions: the life cycle phase they belong to (*e.g.,* specification, design, implementation, test), and their type (*e.g.,* unstructured natural language, grammar-based code, model-based artefact). The **granularity of artefacts** is the level to which artefacts can be decomposed into sub parts. We call a **fragment**, the resulting product of the decomposition of an artefact. A fragment can be itself broken down into smaller parts (or sub-fragments), and so on.
- A **trace link** is a direct relationship between two artefacts. Links can be typed to better support the heterogeneous nature of traceability applications. The type of the link can help express the rationale behind the relationship - it informs not only *how* artefacts are linked but also *why* [55]. Typing is a primary concern in conceptual modeling in general [68]. This definition of a *link* is consistent with the concept of link in popular modeling languages like UML or SysML.
  Links can be explicit or implicit. An **implicit link** shows artefacts bondage at a syntactic or semantic level without the need for an **explicit link** to be part of the model (*e.g.,* a binary class and its respective source code artefact are implicitly "linked" to each other, yet this bondage is not part of any language or grammar definition) [70].

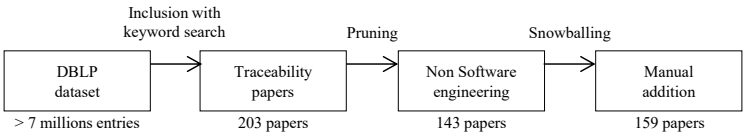


Fig. 1: Survey Process.

- An **agent** is the (human) actor accountable for an artefact, or a link.
- **Trace integrity** is the degree of reliability that bares a trace. It is an indirect measure that includes, for example, both the age of a trace, the volatility of artefacts targeted by the trace, and the automation level of tracing features.

On top of these concepts, a recent work, by Holtmann *et al.*, makes a distinction between a *foundational* and a *specifically model-based* terminology [45]. This latter add a specification for *model* and *language* scope definitions, as well as a distinction between *relational* and *referential* trace links.

- **Intra/Inter model** trace links differentiate between relations that links elements of the same instance of the language and relations linking elements from distinct instances. This distinction was first introduced by Lindval *et al.* [54].
- **Intra/Inter DSL** differentiate between relations that links elements in models based on the same language and relations that links elements in models from different languages.
- The distinction between **Relational and Referential trace links** lies in the instantiation (or not) of the instance link. "A relational trace link is represented by *a dedicated node* with incident directed edges pointing to the trace artifact nodes" whereas "a referential trace link is *a directed edge* from one trace artifact node to another trace artifact node". In the latter case, a trace link is commonly represented as a *property* of the source artefact.

Some of these concepts will explicitly appear in our feature traceability model while others act as requirements and usages that should be supported/facilitated by the features in the model and taken into account when choosing a specific traceability solution depending on how well that solution covers the specific features of interest for the project at hand.

4 Traceability Survey method

In this section we depict the methodology we followed to collect papers proposing traceability solutions, including at the very least the core *representation* component (see previous section). The analysis of these papers will give rise to the feature model we will present next.

The selection process combined the manual selection of a few approaches based on our own experience working in this field and/or covered by other meta-studies [36,4,22,39] together with a systematic literature search mining bibliographic data sources following the literature review process established by Kitchenham and Charters [49]. Fig. 1 depicts the three main steps of the process.

4.1 Data source and search strategy

We used DBLP [10] as our core electronic database to search for primary studies on traceability. To avoid missing possibly relevant approaches, we decided not to put a specific period constraint for the search, but we limited the scope of the search to papers of five pages or more to avoid opinion and vision papers, posters, tool demos and other types of short papers to reduce the number of results while maximizing their quality.

Based on the topic of this survey, we defined the terms of the search query according to the recommendations of Kitchenham and Charters [49]. We apply the query on the title and abstract of potential relevant publications. As using very generic terms like "trace" or "traceability" returned thousands of results, we decided to combine in the search query trace-related keywords with language-related ones since we target traceability proposals that, at the very least, discuss how traces need to be represented / expressed and not only discuss their application to some specific domain without going deep into the details. As many traceability languages are model-based, we included model, modeling, and other core MDE concepts as part of the language variations. This resulted in a total of 203 papers.

Here is the exact query we applied:

```
.*(([Tt]rac(eability|ing))|([Tt]race[rs])).* AND
.*(([Mm]odel[- ])(([Dd]riven)|([Bb]ased))|
MD[DAE]|Model[l]ing|[Tt]ransformation| DSL|[Ll]anguage).*
```

4.2 Pruning

In what follows, we describe our inclusion and exclusion criteria. We further explain how we applied these criteria on the previous set of papers.

Inclusion criteria	Exclusion criteria
1. the paper is a technical contribution	
2. the paper is about tracing in software engineering	1. the paper is not a primary study
3. traceability is the main concern of the paper	

Before we applied these criteria on the potential papers fetched by our query, we removed automatically papers of less than 5 pages long. We also automatically extracted papers whose titles mentioned "biology", "education", "kinetics", "logistics", "physiology", "physics", "neuroscience", "agriculture", and "food" which appeared each in a couple of results. We manually examined the 183 papers left and excluded 40 papers that did not fulfill the criteria or were duplicates.

4.3 Snowballing

At the end of the previous steps, we double-checked that we did not miss any potentially relevant approach due to a number of reasons, *e.g.*, some workshop papers are only indexed by ACM or papers that may be using different synonyms for traceability like "composition" or "extension".

Finally, we added papers we were aware of based on direct knowledge or from other surveys we had read (if not already in the result set) and a few more we found by snowballing on the selected papers references. They amount to a total of 10 more papers. This lead to a final result of 159 papers. Among them, there are 41 journal articles, 82 in conference proceedings, and 36 workshop reports (see Table 1). Fig. 2 shows the chronological distribution of the selected publications.

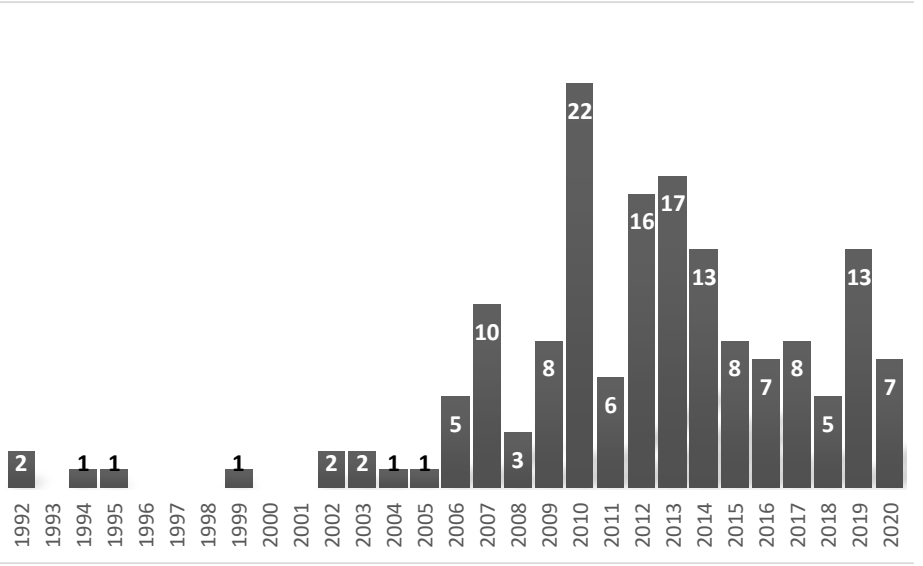


Fig. 2: Papers selected related to traceability and modeling.

Publication type	
Journal	41
Conference	82
Workshop	36

Table 1: Publication types of the selected papers.

4.4 Threats to validity in the selection process

We acknowledge limitations in the execution of our survey method. First, we only used DBLP as a source database. Yet, it is recognized as a representative electronic database for scientific publications on software engineering and already contains more than five million publications from over two million authors. Setting the limit based on the number of pages alone to elude short papers is another threat to validity. Yet, it is a reproducible practice that limits the number of papers to analyse and thus helps concentrate on the topic rather than the engineering of the survey. Then, the vocabulary related to traceability is scattered among various fields of application with their respective nuances. We mitigate the risk of missing papers by manually adding papers that were not using variations of this term but were still referenced by papers that did. Still, focusing

on traceability as a key term was also a conscious decision as we wanted to characterize the works in this field, focusing on those papers that define themselves as part of it.

5 A feature model to characterize software traceability

This section presents our feature model describing the traceability features and dimensions found in the analysis of the literature. Our feature model groups them by similarity and provides additional descriptions on the most important aspects of each one, *e.g.*, different existing alternative implementation of the same feature and/or the most/the least studied ones in each group. Next subsections provide some background on feature modeling and then zoom in to each of the three main dimensions of traceability: trace representation, trace identification, and trace management.

5.1 Introduction to feature modelling

A feature model leverages features as the abstraction mechanism to reason about product variability. It is a hierarchically arranged set of features, where relationships between a parent feature and its child features may be categorized as: *and* – all subfeatures must be selected, *alternative* – only one subfeature can be selected, *inclusive or* – one or more can be selected, *mandatory*, and *optional* [48]. Each feature represents an increment in product functionality.

Feature modeling is a technique that has been intensively used for documenting the points of variability in a software product line, how the points of variability constraint one another, and what constitutes a complete configuration of the system. But beyond product lines, feature models are also more and more used to shed light on complex domains by representing the core concerns and variation points in a complex ecosystems (*e.g.*, [17]), as we do in this paper.

5.2 Trace definition and representation

All approaches must discuss their representation of trace artefacts even if they can differ on the type of traces they consider and the application they target. Representations are so diverse that our survey selected more than 80 papers mentioning their own distinct definition for traceability – with 20 metamodels effectively depicted in those papers. Some researchers present generic graph-based representations [87,37] while others focus on representations much more specific to a concrete application like a metamodel for change impact analysis [34] or multi-model consistency [94]. In both cases, what traceability approaches target and how they represent a trace is differently approached.

Fig. 3 shows the hierarchy of features related to the definition and the representation of trace artefacts. A peculiar focus is put on the typing of traces' relationships. Typing relationships is important to add semantics to the trace so that the engineer can know not only what the linked artefacts are but also why they are linked. As such, it facilitates the application of traceability solutions to specific domains. We also detail the genericity of the language, the nature of the artefacts covered by the traceability proposal, and the possibility to annotate traces with quality properties.

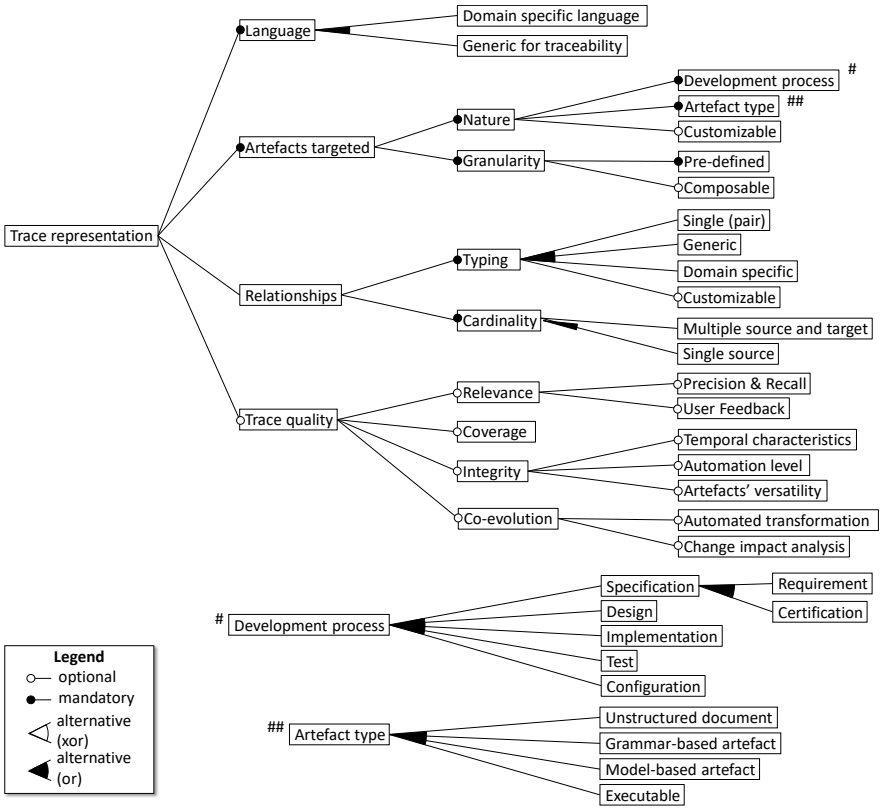


Fig. 3: Features related to the representation of a trace.

We would like to remark the contribution of model-based approaches for traceability in this section. The use of MDE tooling such as ATL [84,47], or the Eclipse Modeling Framework (EMF) allows the automated generation of traceability information as a side effect of executing operations [32,101]. The modeling community has proposed metamodels for end-to-end traceability [43,41], as well as metamodels specific to engineering domains such as model transformation [47,3,97,11] or software product line [47,97]. Paige *et al.* call for more flexible modeling where models of different formats are associated to each others' with annotations that allow automated bond or dependency inference between both application and engineering domains [89,72].

Language Languages specific to traceability provide the ability to represent trace artefacts with increased relevance and accuracy. Yet, they often suffer the limitation to be built *ad hoc* and lack a significant power of reusability into other domains. Among these domain-specific languages for traceability, some authors attempt a generic definition of traceability [43,6] while others provide a language specific to a single domain, *e.g.*, traceability for software product lines [3].

We found few studies interested in the use of general-purpose software language for traceability - even though this would be appealing to industrial partners interested in instrumenting their legacy systems code with traceability information to facilitate future evolution or migrations [65]. Representing traces in spreadsheets, text files, or databases, shows better learning curves than using a domain specific language, but at the cost of a cognitive gap between software engineers and domain experts. As an unfortunate consequence, "the maintenance costs turns out to grow accordingly [to the usability of generic representations] and team members fail to keep the trace artefacts up-to-date" [21].

A potential sweet spot lies in the making of orthogonal approaches that “plug” traceability concerns on top of other languages to benefit from an existing language structure while keeping most of the benefits of using a DSL.

Artefacts targeted We distinguish between the nature of the artefacts targeted by traceability purposes and their granularity as both dimensions are important. For the nature aspect, on the one hand, investigations differ on the development phase they target. Linking requirement specifications to design and code level predominate in the literature with more than 50% of the papers in the survey addressing requirement traceability. Other phases such as test and verification are targeted as well but in a lesser proportion (10 approaches). On the other hand, the type of the artefacts is important to deduce the level of potential generalization to other phases of the software lifecycle. Papers focus on four different types: unstructured document, structured as grammar-, and model-based artefacts, and binaries.

With regard to the granularity of the artefacts targeted, *i.e.*, their level of decomposition, few approaches go for a customizable granularity to adapt to artefact hierarchies [43,60] while most of the others focus on specific types of artefacts (*e.g.,* to concentrate their work on specific optimizations of trace identification).

Relationship types As many authors have demonstrated, offering to the user the ability to define personalized types of relations between the artefacts of a system fosters the comprehensibility of the traces produced [68]. We distinguish between approaches offering predefined types and approaches allowing custom typing. Often the predefined types relate to the field of software engineering (implements, inherits, uses, executes ...), but not only. For example, Maletic *et al.* mention that a separation between *causal*, *non causal*, and *navigation* relationships can be appropriate [57]. Predefined types allow increased monitoring and user-friendliness to developers. They are found in most contributions relating the optimization of trace identification. On the other hand, allowing users to define the types of relationships specific to their area of expertise helps to fill the gap between the design and the use of tracing functionalities [102].

Obviously a fixed typing facilitates the analysis of the traces as the potential set of semantics and interpretations are fixed while offering domain-specific types increases the usability and comprehensibility of the approach. As an example, SysMLv2 is offering a more powerful mechanism to define links between artefacts compared to the previous SysML version (where we had a sole dependency-like mechanism).

The literature shows also a distinction between approaches considering relationships with multiple sources and targets and relationships allowing only a single source.

Trace quality In most of the papers, quality aspects are barely mentioned. It seems quality of the generated traces is not a major focus, or at least storing and annotating the traces with such information is not. Yet, a few studies mention coverage and integrity. The coverage of a set of execution traces is used in approaches for software testing [33]. Coverage is also used by Rath *et al.* who address the problem of missing links between commits and issues with a classifier they train on textual commit information to identify missing links between issues and commits (*i.e.,* a lack in the coverage indicates such missing links) [82]. Matrix-based visualizations are particularly fit to assist coverage related tasks (See Section 5.4). Integrity of traces is addressed in work on model transformation where co-evolution figures an automatic verification of their coherence with other (versatile) software artefacts [94,92]. In the same manner, Heisig *et al.* tag links which ends artefacts have been modified or deleted to inform the user of such changes [43]. The co-evolution of traces implies measuring distances between artefacts (syntactic, cognitive, geographic, cultural...) [9]. It also refers to the analysis of the changes of the system that impact traceability artefacts [34,98]. In our survey, nine papers address artefacts co-evolution and 17 tackle model transformation limitations. These latter are a valuable tool to automate co-evolution tasks. In the many studies focusing on the optimization of link identification, the quality of the results is mainly evaluated with precision and recall measurements and never rely on inherent trace artefacts characteristics. Few researchers include a user feedback [13].

5.3 Trace identification

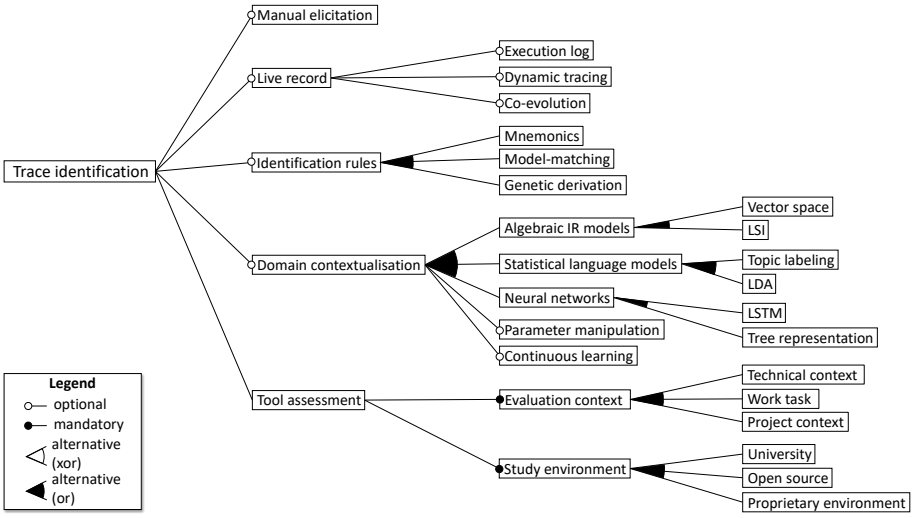


Fig. 4: Features related to the identification of trace links

Fig. 4 shows the hierarchy of features related to the identification of traces with four main possible categories: the manual elicitation of traces, their live record during execu-

tion and evolution, rule-based alternatives to assist the user with automation potential, and AI-augmented identification with domain contextualization.

Manual elicitation Manual elicitation makes possible to create traces in an *ad hoc* manner. As an example, one of our industrial partner chose to hire a developer to elicit trace links necessary for a certification commitment. This was chosen rather than a (semi-)automated approach, as they were not convinced the effort of augmenting an existing tool would pay off for that specific project.

Recording instrumentation Teams can instrument the live record of traces during the execution and the evolution of software artefacts. This way traces recording the system changes are a side-effect of those same changes. There are initiatives to instrument existing languages such as ATL with rich log generation [84,31], while others consider trace record an aspect that can be weaved with current existing languages [78,84]. Ziegenhagen *et al.* mix execution traces with metadatas [103], and use developer interaction records [104] to enrich existing traceability artefact.

Model transformations are considered the hearth and soul of software modeling and, consequently, numerous studies attempt to enrich trace generation during transformation execution [97,83,31]. This ubiquitous integration (see Fig. 5, bottom branch) allows a semantically rich tracing of target and source artefacts [71]. Unfortunately, this option can only be applied when the system is being built, not when the system is already in place.

Identification rules Once a system is in place, teams can identify rules that help retrieve and maintain traceability relations [64,93]. Nentwich *et al.* describe a novel semantics for first-order logic that produces links instead of truth values and give an account of their content management strategy that provides rule-based link generation and consistency check [66]. At the model level, Grammel *et al.* use a graph-based model matching technique to exploit metamodel matching techniques for the generation of trace links for arbitrary source and target models [37], and Saada *et al.* recover execution traces of model transformation using genetic algorithms [83].

Domain contextualization Back in 1992, Borillo *et al.* published an article on the use of information retrieval techniques for linguistics applied to spatial software engineering [14]. This precursor work opened the box for AI-augmented traceability where machine learning algorithms help extract knowledge specific to the application domain (later called domain-contextualized traceability [40]). This is specially useful when the source (or target) of the trace link is an unstructured document or when such document is key to infer traces among other artefacts.

Today, domain contextualization by means of machine learning for topic modeling, word embedding, and more generally knowledge extraction from unorganized text documents, is the most popular traceability feature [39,102]. This collective effort made the identification of bonds between requirement specifications and other artefacts possible with a gradually improving precision [5,23]. Studies on domain contextualization are separated into three subgroups according to the type of tools used (algebraic information retrieval models, statistical language models, and neural networks). For example, Florez *et al.* derive fine-grained requirement to source code links [30], Rath *et al.*

complete missing links between commits and issues [82], Marcus *et al.* identify links between documentation and source code [59]. An interesting publication from Poshyvanyk *et al.* shows that mixing expertise both in information retrieval techniques and engineering domains gives far better results than when taken separately [79]. McMillan *et al.* add that using structural information together with textual information benefits automated link recovery (between requirements and source code) [61]. In total, we found 22 approaches dedicated to this topic alone in our survey. We do not discuss in this paper the techniques related to data collection and training optimization. These are important features for automated learning which are discussed in depth in specialized literature.

Teams are also using genetic algorithms to cope with the variety of algorithms and parameters these approaches use [58,73], and structural information to foster methodologies interweaving [74]. Unfortunately, a common critique rose against these positive results. Too many teams compete with each others to accomplish a better precision and recall when there is no standard to the effective quantification of tracing artefacts into such variables. Too few attempt at qualifying the overall relation between these measurement and the effective impact on software development [22].

In that regard, Shin *et al.* propose a set of guidelines for benchmarking automated traceability techniques. Their evaluation (of 24 approaches) shows that methods of evaluation (when they are used appropriately) sometimes are not suitable to other application domains and that the variation in results across project is not investigated [91]. This corroborate Borg *et al.* who, in a systematic literature mapping on information retrieval approaches to traceability, notice that there are no empirical evidence that any IR model outperforms another consistently [13]. The ability to continuously improve the learning process is mentioned in the literature but we found no evidence of its application.

Tool assessment Very few of the traceability approaches have been empirically assessed on industrial use cases. The actual trend to report solely for precision and recall values indicates an important issue in the automated identification of traces and may justify the weak investment of industry in this sector [13,69].

Borg *et al.* published a taxonomy for information retrieval techniques applied to traceability [12]. They emphasize the importance of the assessment of the tooling used to derive or identify traces. More specifically, the authors draw a differentiation between two orthogonal dimensions: the evaluation context that precises *where* in the context the tool is assessed (*e.g.,* at a technical, work task, or project level); and, the study environment that shows the kind of data used to fulfil the assessment (*e.g.,* proprietary, open source, or academic). These features will affect the measurable attributes used for the assessment as well as their generalizability.

5.4 Trace management

Fig. 5 shows the hierarchy of features related to the management of trace artefacts: their maintenance, integrity, persistence, and integration in running software systems.

Trace Maintenance Trace links may be affected by changes on the artefacts they link (directly or transitively) and therefore can easily become obsolete. This gradual decay must be seriously taken into account to avoid having to re-elicit traces every time they

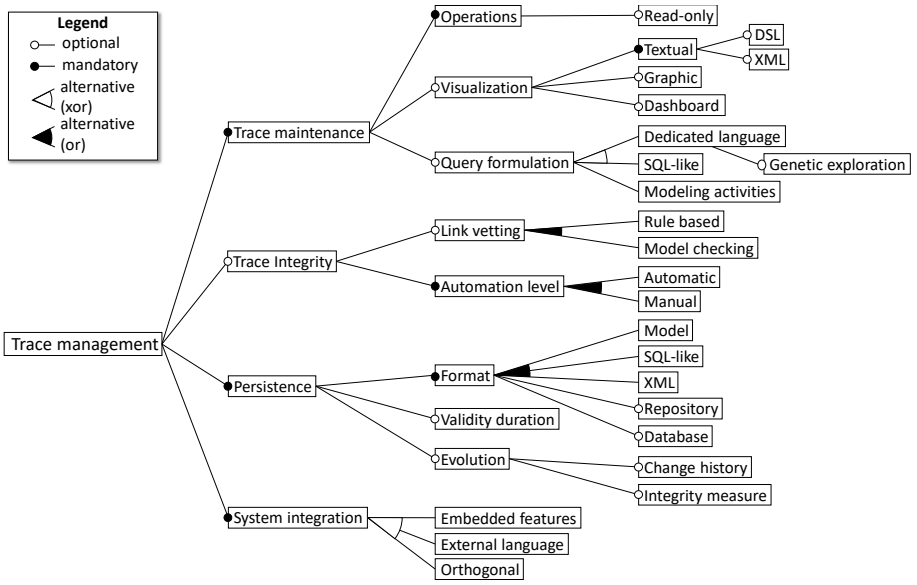


Fig. 5: Tool support for traceability management.

need to be analyzed. A manual maintenance is not always impossible but not typically feasible in practice due to the amount of information such inspections would involve. Co-evolution techniques [64,26,80] attempt to tackle the burden to maintain trace links up-to-date [88,19].

Beyond being able to manipulate traces, we also need to offer proper ways to visualize and inspect them [29]. The use of graphical representations stimulates human perception and the integration of such technique in traceability frameworks is a useful feature to augment user awareness [43]. On the other hand, matrix-based views offer a valuable perspective to understand and analyse traces [53]. They are particularly efficient in assisting the visualization of coverage characteristics of traceability [33,82].

In parallel, allowing a rich formulation of queries to assist the exploration of existing traces will help with reducing the amount of information users need to navigate through [19]. More precisely, structured text, in the form of metamodel instances or XML sheets allows query-based mining of trace datasets [24]. Interaction wise, hypertext links is a *de facto* standard to browse trace links. Indeed, following links through successive clicks has become almost natural. Querying relies on the type of representation of traceability artefacts: SQL-like languages benefit from a long history of information mining while dedicated languages offers better legibility. Genetic programming has also permitted the automation of query formulation [77].

Trace Integrity To cope with the decay and volatility mentioned above, ways to determine the integrity of existing traces are greatly needed. Work on these questions, although called out loudly by literature studies, is scarce in practice [101,4]. The first op-

tion is given with manual annotation or vetting of trace links to inform about their level of reliability. Annotations allow a qualitative and quantitative evaluation [18]. This is the case for back-propagation of verification and validation results between design and requirements [42]. Some approaches enable the definition of invariant rules while manipulating traces or their targets [19]. If the invariant is violated, an exception for that trace is automatically generated. For example, we could define a rule that is violated when a change occurs in an artefact targeted by a trace if the corresponding link was identified more than two versions prior to the current version. In the same vein, Heisig *et al.* tag trace links when their target (or source) artefacts are modified or deleted [43]. Thanks to the ubiquitous integration of the tool, warning is raised consequently in EMF.

Trace persistence Many different storage alternatives exist for traceability artefacts. An option is to use SQL-like grammar to store and retrieve traces with the power of database tooling, or to use XML documents to represent trace matrix in a transformable format [57,27]. The industry uses a lot of informal format and link representations often remains implemented in spreadsheets, text files, databases or requirement management tools. These links deteriorate quickly during a project as time pressured team members fail to update them. Researchers aiming at a reusable approach favour model-based representations able to express specifically defined concepts related to traceability (often in a specific domain of application). The burden of maintaining traces coherent is eased in model-based solutions [21].

Another concern lies in the recording of trace evolution. The trace creation should be recorded, with the successive changes that affect it, for evolution analysis. Integrity measures respective to evolution events (*e.g.,* creation, modification) should be recorded as well to evaluate their evolution during a period of time. Rahimi *et al.* ensure the co-evolution of artefacts and traces [80] using a set of heuristics coupled with refactoring detection and information retrieval technique to detect change scenarios between contiguous versions of software systems.

System integration Like most of the MDE approaches, Helming *et al.* use the same modeling language for both traceability and system artefacts [44]. Tracing features are embedded in the language. The conjunct use of EMF and a dedicated traceability metamodel (both written in Ecore) facilitates the integration of traceability features including graphical versions to stimulate human perception and standard analysis of traces in native environment. Galvao *et al.* in their seminal work on traceability and MDE call for more loosely coupled traceability support that can integrate external relationship with independent representations (in another, ideally common language) [32] as also elaborated by Azevedo *et al.* [6]. Finally, the SysMLv2 implementation committee is calling for *orthogonal* implementation of features such as traceability, annotations and comment through meta-level libraries in order to keep concerns separated at design level.

6 Discussion

The feature model is a first step towards the shared understanding of all dimensions involved in a traceability solution. Ideally, a company interested in a certain set of such dimensions could try to create its perfect traceability solution by combining the top solutions for each dimension. But this is not yet a real possibility as those solution would be difficult to combine and, more importantly, several of the features in the feature model do not really have a great solution yet. This section elaborates on this discussion by presenting some open challenges in software traceability research.

Common traceability metamodel. We have counted over 20 different traceability metamodel proposals. Nevertheless, some are solutions limited to the specific problems the authors present as case studies. And these metamodels are rarely reused, if ever. This proliferation is a challenge to make different traceability solutions interoperate. The research community should agree in a unified proposal that facilitates the composability of traceability solutions.

Security of trace data. Considering that traceability is a major aspect in certification and other critical applications, it is surprising to see so little interest in security concerns in relationship to trace artefacts. We believe security mechanisms (even simple rule-based access control) for traceability are needed to control who can modify what trace data, given the implication such changes can have.

Library of trace types and semantics. We already mentioned the importance of having a rich set of types for traces to let engineers express the reasons behind the creation of a given trace. But at the same time, complete freedom makes reusability of analysis techniques difficult. We would like to see a rich yet predefined set of types for traces that could then be imported in new traceability projects.

Usefulness of identified traces. Managing a large number of traces is time-consuming. As such, we should make sure every explicit trace is actually useful. So far, algorithms aimed at automatically identifying traces are compared based on standard properties like precision and recall. But they should be evaluated on "usefulness": are those traces useful for the end-user? or are they simply redundant noise?

Verification, validation and testing of traces. Our ample literature on verification, validation and testing methods for software engineering should be extended to deal with trace data, especially from a temporal perspective. Reasoning on outdated and potentially incorrect trace data could have strong damaging impacts on the system as a whole. So far, very few approaches target these aspects except in coevolution in model-driven engineering. A recent study shows that the ability to justify with evidences and uncertainty evaluation the quality and integrity of traces is a prerequisite to robust and reliable traceability [8]. Given the effort required to create traces in the first place, it is important to instill more confidence to practitioners unsure if creating traces is worthwhile.

Traceability as core concern in general languages. Another important step towards the mainstream adoption of traceability in industry is the integration of the common traceability metamodel in popular modeling languages like UML or SysML, in the form of a profile (to be able to directly reuse existing modeling tools available for those languages) or new packages in the respective standards. This way, traceability would become a core concern and a primary class modeling primitive in software develop-

ment while still being a rich concept and not just a variation of the simple generic plain dependency relationship we can use right now in those languages.

Working together with the industry. Orthogonal to all the others, we (the research community) should aim at more frequent exchanges with practitioners to better understand why they still create traces manually instead of reusing any of the dozens of existing solutions. Some reasons have been already hinted in this paper, but there might be others we are not aware of. If we want traceability research to transfer to industry, more and better communication flows should be part of the agenda.

7 Conclusion

Our survey reveals a continuous interest in traceability even if, often, it does not have the spotlight it deserves given the key role it plays in a good deal of software engineering tasks [4]. Work relating to traceability is indeed disseminated within established research communities (e.g., debugging, SPL). Existing conceptualizations vary greatly depending on the community to which its authors belong to as well as the objectives they aim at. As a consequence, a clear and measurable idea of the costs and benefits to software traceability is slow to emerge. To help visualize, classify and compare the different traceability approaches, we propose a feature model covering all important traceability aspects, as derived from a thorough analysis of the traceability literature. Following the existing body of work, we put special emphasis in separating how traces are represented from how they are identified and managed.

Beyond the feature model, our analysis highlights several limitations of current traceability approaches that should be further developed. We believe advancing on those aspects is especially important, even more given the new traceability challenges posed by the growing use of AI in Software Engineering (e.g. in terms of reproducibility and explainability of the AI decisions) [90,99]. In this sense, we hope this paper serves as a "wake-up call" to make sure new AI for SE proposals come together with a proper traceability mechanism that assists engineers in evaluating and understanding the impact of the new AI components in the software engineering process instead of having to blindly trust them.

As further work, we plan to start working on the above-mentioned aspects starting with a collaboration with some of the authors of other proposals to map and bridge their algorithms and techniques to our modular and quality-focused metamodel in order to combine the benefits of a unified and generic approach with those of a more domain-specific representation. We will also study how better embed traceability concepts into mainstream modeling languages (like UML or SysML) to further facilitate its adoption.

Acknowledgements: This work has been partially funded by the Spanish government (LOCOSS project - PID2020-114615RB-I00), and receives support from the ECSEL Joint Undertaking (AIDOaRt - grant agreement No 101007350).

[4] As an example, ICSE'18 awarded a trace-based paper as the most influential paper in the past 10 years [50]. The work introduced a novel trace-based approach to debugging. Though the focus was on the debugging aspect of the paper, traceability was the key to achieve that debugging improvement. The word "trace" alone is mentioned 46 times in the 10 pages paper.

References

1. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Systems Journal **45**(3), 515–526 (2006). https://doi.org/10.1147/sj.453.0515, https://doi.org/10.1147/sj.453.0515
2. Amar, B., Leblanc, H., Coulette, B., Dhaussy, P.: Automatic co-evolution of models using traceability. Communications in Computer and Information Science **170** (2013)
3. Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.C., Rummler, A., Sousa, A.: A model-driven traceability framework for software product lines. Software and Systems Modeling **9**(4), 427–451 (2010), https://doi.org/10.1007/s10270-009-0120-9
4. Antoniol, G., Cleland-Huang, J., Hayes, J.H., Vierhauser, M.: Grand challenges of traceability: The next ten years. CoRR **abs/1710.03129** (2017), http://arxiv.org/abs/1710.03129
5. Arunthavanathan, A., Shanmugathasan, S., Ratnavel, S., Thiyagarajah, V., Perera, I., Meedeniya, D., Balasubramaniam, D.: Support for traceability management of software artefacts using natural language processing. In: 2016 Moratuwa Engineering Research Conference (MERCon). pp. 18–23 (April 2016). https://doi.org/10.1109/MERCon.2016.7480109
6. Azevedo., B., Jino., M.: Modeling traceability in software development: A metamodel and a reference model for traceability. In: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,. pp. 322–329. INSTICC, SciTePress (2019). https://doi.org/10.5220/0007715103220329
7. Badreddin, O., Sturm, A., Lethbridge, T.C.: Requirement traceability: A model-based approach. In: 2014 IEEE 4th International Model-Driven Requirements Engineering Workshop (MoDRE). pp. 87–91 (Aug 2014). https://doi.org/10.1109/MoDRE.2014.6890829
8. Batot, E.R., Gerard, S., Cabot, J.: (Not) yet another metamodel for software traceability. In: Proceedings of the 13th System Analysis and Modelling Conference. p. 1–10. SAM '21, Association for Computing Machinery (2021)
9. Bjarnason, E., Smolander, K., Engström, E., Runeson, P.: A theory of distances in software engineering. Inf. Softw. Technol. **70**(C), 204–219 (February 2016). https://doi.org/10.1016/j.infsof.2015.05.004, https://doi.org/10.1016/j.infsof.2015.05.004
10. advisory board, T.D.: The dblp team: Monthly snapshot release of july 2020. DBLP - Computer science bibliography. (July 2020), https://dblp.org/xml/release/dblp-2020-0701.xml.gz, https://dblp.org/xml/release/dblp-2020-0701.xml.gz
11. Bondé, L., Boulet, P., Dekeyser, J.L.: Traceability and Interoperability at Different Levels of Abstraction in Model-Driven Engineering, pp. 263–276. Springer Netherlands, Dordrecht (2006)
12. Borg, M., Runeson, P., Brodén, L.: Evaluation of traceability recovery in context: A taxonomy for information retrieval tools. In: 16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012). pp. 111–120 (May 2012). https://doi.org/10.1049/ic.2012.0014
13. Borg, M., Runeson, P., Ardö, A.: Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. Empirical Software Engineering **19**(6), 1565–1616 (2014), https://doi.org/10.1007/s10664-013-9255-y
14. Borillo, M., Borillo, A., Castell, N., Latour, D., Toussaint, Y., Verdejo, M.F.: Applying linguistic engineering to spatial software engineering: The traceability problem. In: Proceedings of the 10th European Conference on Artificial Intelligence. p. 593–595. ECAI '92, USA (1992)
15. Bouillon, E., Mäder, P., Philippow, I.: A survey on usage scenarios for requirements traceability in practice. In: Requirements Engineering: Foundation for Software Quality, pp. 158–173. Springer Berlin Heidelberg (2013), https://doi.org/10.1007/978-3-642-37422-7_12

16. Bourque, P., Fairley, R.E. (eds.): SWEBOK: Guide to the Software Engineering Body of Knowledge. IEEE Computer Society, Los Alamitos, CA, version 3.0 edn. (2014), http://www.swebok.org/
17. Brunelière, H., Burger, E., Cabot, J., Wimmer, M.: A feature-based survey of model view approaches. Softw. Syst. Model. **18**(3), 1931–1952 (2019). https://doi.org/10.1007/s10270-017-0622-9, https://doi.org/10.1007/s10270-017-0622-9
18. Buchmann, R.A., Karagiannis, D.: Modelling mobile app requirements for semantic traceability. Requirements Eng **22**(1), 41–75 (jul 2015). https://doi.org/10.1007/s00766-015-0235-1, https://doi.org/10.1007%2Fs00766-015-0235-1
19. Bünder, H., Rieger, C., Kuchen, H.: A domain-specific language for configurable traceability analysis. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development. SCITEPRESS - Science and Technology Publications (2017). https://doi.org/10.5220/0006138503740381, https://doi.org/10.5220%2F0006138503740381
20. Charalampidou, S., Ampatzoglou, A., Karountzos, E., Avgeriou, P.: Empirical studies on software traceability: A mapping study. Journal of Software: Evolution and Process (2020). https://doi.org/https://doi.org/10.1002/smr.2294, https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2294, e2294 JSME-19-0120.R2
21. Cleland-Huang, J., Berenbach, B., Clark, S., Settimi, R., Romanova, E.: Best practices for automated traceability. Computer **40**(6), 27–35 (2007)
22. Cleland-Huang, J., Gotel, O.C.Z., Huffman Hayes, J., Mäder, P., Zisman, A.: Software traceability: Trends and future directions. In: Future of Software Engineering Proceedings. p. 55–69. FOSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2593882.2593891, https://doi.org/10.1145/2593882.2593891
23. De Lucia, A., Marcus, A., Oliveto, R., Poshyvanyk, D.: Information retrieval methods for automated traceability recovery. Software and Systems Traceability pp. 71–98 (2012), https://doi.org/10.1007/978-1-4471-2239-5_4
24. Dietrich, T., Cleland-Huang, J., Shin, Y.: Learning effective query transformations for enhanced requirements trace retrieval. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 586–591 (Nov 2013). https://doi.org/10.1109/ASE.2013.6693117
25. Drivalos, N., Kolovos, D.S., Paige, R.F., Fernandes, K.J.: Engineering a dsl for software traceability. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) Software Language Engineering. pp. 151–167. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
26. Drivalos-Matragkas, N., Kolovos, D.S., Paige, R.F., Fernandes, K.J.: A state-based approach to traceability maintenance. In: Proceedings of the 6th ECMFA Traceability Workshop. p. 23–30. ECMFA-TW '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1814392.1814396, https://doi.org/10.1145/1814392.1814396
27. Elamin, R., Osman, R.: Implementing traceability repositories as graph databases for software quality improvement. In: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). pp. 269–276 (2018). https://doi.org/10.1109/QRS.2018.00040
28. Feldmann, S., Kernschmidt, K., Wimmer, M., Vogel-Heuser, B.: Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering. In: Software Engineering 2020. vol. 153, pp. 105–134 (2019). https://doi.org/10.1016/j.jss.2019.03.060, https://doi.org/10.1016/j.jss.2019.03.060
29. Fittkau, F., Waller, J., Wulf, C., Hasselbring, W.: Live trace visualization for comprehending large software landscapes: The explorviz approach. In: 2013 First IEEE Working Conference on Software Visualization (VISSOFT). pp. 1–4 (Sep 2013). https://doi.org/10.1109/VISSOFT.2013.6650536

30. Florez, J.M.: Automated fine-grained requirements-to-code traceability link recovery. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 222–225 (May 2019). https://doi.org/10.1109/ICSE-Companion.2019.00087
31. la Fosse, T.B., Tisi, M., Mottu, J.M.: Injecting execution traces into a model-driven framework for program analysis. In: Software Technologies: Applications and Foundations, pp. 3–13. Springer International Publishing (2018), https://doi.org/10.1007%2F978-3-319-74730-9_1
32. Galvao, I., Goknil, A.: Survey of traceability approaches in model-driven engineering. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007). pp. 313–313 (Oct 2007). https://doi.org/10.1109/EDOC.2007.42
33. Gannous, A., Andrews, A.: Integrating safety certification into model-based testing of safety-critical systems. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). pp. 250–260 (Oct 2019). https://doi.org/10.1109/ISSRE.2019.00033
34. Goknil, A., Kurtev, I., van den Berg, K., Spijkerman, W.: Change impact analysis for requirements: A metamodeling approach. Information and Software Technology **56**(8), 950 – 972 (2014). https://doi.org/https://doi.org/10.1016/j.infsof.2014.03.002, http://www.sciencedirect.com/science/article/pii/S0950584914000615
35. Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Proceedings of IEEE International Conference on Requirements Engineering. pp. 94–101 (April 1994). https://doi.org/10.1109/ICRE.1994.292398
36. Gotel, O., Cleland-Huang, J., Hayes, J.H., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., Mäder, P.: Traceability Fundamentals - Software and Systems Traceability, pp. 3–22. Springer London, London (2012), https://doi.org/10.1007/978-1-4471-2239-5_1
37. Grammel, B., Kastenholz, S., Voigt, K.: Model matching for trace link generation in model-driven software development. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7590, pp. 609–625. Springer (2012). https://doi.org/10.1007/978-3-642-33666-9_39, https://doi.org/10.1007/978-3-642-33666-9_39
38. Guana, V., Stroulia, E.: End-to-end model-transformation comprehension through fine-grained traceability information. Softw Syst Model Systems Modeling **18**(2), 1305–1344 (jun 2017). https://doi.org/10.1007/s10270-017-0602-0
39. Guo, J., Cheng, J., Cleland-Huang, J.: Semantically enhanced software traceability using deep learning techniques. In: Proceedings of the 39th International Conference on Software Engineering. p. 3–14. ICSE '17, IEEE Press (2017). https://doi.org/10.1109/ICSE.2017.9, https://doi.org/10.1109/ICSE.2017.9
40. Guo, Q., Chen, S., Xie, X., Ma, L., Hu, Q., Liu, H., Liu, Y., Zhao, J., Li, X.: An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE (nov 2019)
41. Haidrar, S., Anwar, A., Roudies, O.: Towards a generic framework for requirements traceability management for SysML language. In: 2016 4th IEEE International Colloquium on Information Science and Technology (CiSt). IEEE (oct 2016). https://doi.org/10.1109/cist.2016.7805044, https://doi.org/10.1109%2Fcist.2016.7805044
42. Hegedus, A., Bergmann, G., Rath, I., Varro, D.: Back-annotation of simulation traces with change-driven model transformations. In: 2010 8th IEEE International Conference on Software Engineering and Formal Methods. IEEE (sep 2010). https://doi.org/10.1109/sefm.2010.28, https://doi.org/10.1109%2Fsefm.2010.28

43. Heisig, P., Steghöfer, J.P., Brink, C., Sachweh, S.: A generic traceability metamodel for enabling unified end-to-end traceability in software product lines. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. p. 2344–2353. SAC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297510, https://doi.org/10.1145/3297280.3297510
44. Helming, J., Koegel, M., Naughton, H., David, J., Shterev, A.: Traceability-based change awareness. In: Model Driven Engineering Languages and Systems. vol. 5795, pp. 372–376. Springer Berlin Heidelberg (10 2009)
45. Holtmann, J., Steghöfer, J.P., Rath, M., Schmelter, D.: Cutting through the jungle: Disambiguating model-based traceability terminology. In: 2020 IEEE 28th International Requirements Engineering Conference (RE). pp. 8–19 (Aug 2020). https://doi.org/10.1109/RE48521.2020.00014
46. ISO: Road vehicles – Functional safety (2011)
47. Jiménez, Á., Vara, J.M., Bollati, V.A., Marcos, E.: Model-driven development of model transformations supporting traces generation. In: Building Sustainable Information Systems, pp. 233–245. Springer US (2013), https://doi.org/10.1007%2F978-1-4614-7540-8_18
48. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: Form: A feature-;oriented reuse method with domain-;specific reference architectures. Annals of Software Engineering **5**(1), 143 (1998). https://doi.org/10.1023/A:1018980625587, https://doi.org/10.1023/A:1018980625587
49. Kitchenham, B., Pearl Brereton, O., Budgen, D., Turner, M., Bailey, J., Linkman, S.: Systematic literature reviews in software engineering – a systematic literature review. Information and Software Technology **51**(1), 7 – 15 (2009). https://doi.org/https://doi.org/10.1016/j.infsof.2008.09.009, http://www.sciencedirect.com/science/article/pii/S0950584908001390
50. Ko, A.J., Myers, B.A.: Debugging reinvented: Asking and answering why and why not questions about program behavior. In: Proceedings of the 30th International Conference on Software Engineering. p. 301–310. ICSE '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1368088.1368130, https://doi.org/10.1145/1368088.1368130
51. Kokaly, S., Salay, R., Chechik, M., Lawford, M., Maibaum, T.: Safety case impact assessment in automotive software systems: An improved model-based approach. In: Lecture Notes in Computer Science, pp. 69–85. Springer International Publishing (2017), https://doi.org/10.1007/978-3-319-66266-4_5
52. Königs, S.F., Beier, G., Figge, A., Stark, R.: Traceability in systems engineering – review of industrial practices, state-of-the-art technologies and new research solutions. Advanced Engineering Informatics **26**(4), 924 – 940 (2012). https://doi.org/https://doi.org/10.1016/j.aei.2012.08.002, http://www.sciencedirect.com/science/article/pii/S1474034612000766
53. Li, W., Hayes, J.H., Yang, F., Imai, K., Yannelli, J., Carnes, C., Doyle, M.: Trace matrix analyzer (tma). In: 2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE). pp. 44–50 (May 2013). https://doi.org/10.1109/TEFSE.2013.6620153
54. Lindval, M., Sandahl, K.: Practical implications of traceability. Software: Practice and Experience **26**(10), 1161–1180 (1996)
55. Mader, P., Gotel, O., Philippow, I.: Motivation matters in the traceability trenches. In: 2009 17th IEEE International Requirements Engineering Conference. pp. 143–148 (Aug 2009). https://doi.org/10.1109/RE.2009.23

56. Mader, P., Philippow, I., Riebisch, M.: A traceability link model for the unified process. In: Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007). vol. 3, pp. 700–705 (July 2007). https://doi.org/10.1109/SNPD.2007.342
57. Maletic, J.I., Collard, M.L., Simoes, B.: An xml based approach to support the evolution of model-to-model traceability links. In: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering. p. 67–72. TEFSE '05, Association for Computing Machinery (2005)
58. Marcén, A.C., Lapeña, R., Pastor, O., Cetina, C.: Traceability link recovery between requirements and models using an evolutionary algorithm guided by a learning to rank algorithm: Train control and management case. J. Syst. Softw. **163**, 110519 (2020). https://doi.org/10.1016/j.jss.2020.110519, https://doi.org/10.1016/j.jss.2020.110519
59. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: 25th International Conference on Software Engineering, 2003. Proceedings. pp. 125–135 (May 2003). https://doi.org/10.1109/ICSE.2003.1201194
60. Maro, S., Steghöfer, J.P., Bozzelli, P., Muccini, H.: TracIMo: a traceability introduction methodology and its evaluation in an agile development team. Requirements Engineering (August 2021)
61. McMillan, C., Poshyvanyk, D., Revelle, M.: Combining textual and structural analysis of software artifacts for traceability link recovery. In: 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering. pp. 41–48 (May 2009). https://doi.org/10.1109/TEFSE.2009.5069582
62. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B.: Testing or formal verification: Do-178c alternatives and industrial experience. IEEE Software **30**(3), 50–57 (2013). https://doi.org/http://doi.ieeecomputersociety.org/10.1109/MS.2013.43
63. Mustafa, N., Labiche, Y.: The need for traceability in heterogeneous systems: A systematic literature review. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC). vol. 1, pp. 305–310 (July 2017). https://doi.org/10.1109/COMPSAC.2017.237
64. Mäder, P., Gotel, O., Philippow, I.: Rule-based maintenance of post-requirements traceability relations. In: 2008 16th IEEE International Requirements Engineering Conference. pp. 23–32 (Sep 2008). https://doi.org/10.1109/RE.2008.24
65. Nejati, S., Sabetzadeh, M., Falessi, D., Briand, L., Coq, T.: A sysml-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. Information and Software Technology **54**(6), 569 – 590 (2012). https://doi.org/https://doi.org/10.1016/j.infsof.2012.01.005, http://www.sciencedirect.com/science/article/pii/S095058491200016X, special Section: Engineering Complex Software Systems through Multi-Agent Systems and Simulation
66. Nentwich, C., Capra, L., Emmerich, W., Finkelsteiin, A.: Xlinkit: A consistency checking and smart link generation service. ACM Trans. Internet Technol. **2**(2), 151–185 (May 2002)
67. Neumuller, C., Grunbacher, P.: Automating software traceability in very small companies: A case study and lessons learne. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). pp. 145–156 (2006). https://doi.org/10.1109/ASE.2006.25
68. Olivé, A.: Representation of generic relationship types in conceptual modeling. In: Pidduck, A.B., Ozsu, M.T., Mylopoulos, J., Woo, C.C. (eds.) Advanced Information Systems Engineering. pp. 675–691. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
69. Oliver, A., Odena, A., Raffel, C., Cubuk, E.D., Goodfellow, I.J.: Realistic evaluation of deep semi-supervised learning algorithms. CoRR **abs/1804.09170** (2018), http://arxiv.org/abs/1804.09170

70. Paige, R., Olsen, G., Kolovos, D., Zschaler, S., Power, C.: Building model-driven engineering traceability classifications. In: Computer Science (01 2010)
71. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous identification and encoding of trace-links in model-driven engineering. Software & Systems Modeling **10**(4), 469–487 (2011), https://doi.org/10.1007/s10270-010-0158-8
72. Paige, R.F., Zolotas, A., Kolovos, D.: The changing face of model-driven engineering. In: Present and Ulterior Software Engineering, pp. 103–118. Springer International Publishing (2017), https://doi.org/10.1007/978-3-319-67425-4_7
73. Panichella, A., Dit, B., Oliveto, R., Penta, M.D., Poshynanyk, D., Lucia, A.D.: How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 522–531 (May 2013). https://doi.org/10.1109/ICSE.2013.6606598
74. Panichella, A., McMillan, C., Moritz, E., Palmieri, D., Oliveto, R., Poshyvanyk, D., Lucia, A.D.: When and how using structural information to improve ir-based traceability recovery. In: 2013 17th European Conference on Software Maintenance and Reengineering. pp. 199–208 (March 2013). https://doi.org/10.1109/CSMR.2013.29
75. Panis, M.C.: Successful deployment of requirements traceability in a commercial engineering organization...really. In: 2010 18th IEEE International Requirements Engineering Conference. pp. 303–307 (Sep 2010). https://doi.org/10.1109/RE.2010.43
76. Paz, A., El Boussaidi, G.: A requirements modelling language to facilitate avionics software verification and certification. In: 2019 IEEE/ACM 6th International Workshop on Requirements Engineering and Testing (RET). pp. 1–8 (May 2019). https://doi.org/10.1109/RET.2019.00008
77. Pérez, F., Ziadi, T., Cetina, C.: Utilizing Automatic Query Reformulations as Genetic Operations to Improve Feature Location in Software Models. IEEE Transactions on Software Engineering (2020). https://doi.org/10.1109/TSE.2020.3000520, https://hal.sorbonne-universite.fr/hal-02852488
78. Pfeiffer, R., Reimann, J., Wąsowski, A.: Language-independent traceability with lassig. In: Modelling Foundations and Applications, pp. 148–163. Springer International Publishing (2014), https://doi.org/10.1007%2F978-3-319-09195-2_10
79. Poshyvanyk, D., Gueheneuc, Y., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Transactions on Software Engineering **33**(6), 420–432 (2007)
80. Rahimi, M., Cleland-Huang, J.: Evolving software trace links between requirements and source code. In: 2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST). pp. 12–12 (May 2019). https://doi.org/10.1109/SST.2019.00012
81. Randel, B.: Towards a methodology of computing system design. NATO Software Engineering Conference **Brussels, Scientific Affairs Division, NATO (Published 1969)**, pp. 204–208 (1968)
82. Rath, M., Rendall, J., Guo, J.L.C., Cleland-Huang, J., Mäder, P.: Traceability in the wild: Automatically augmenting incomplete trace links. In: Proceedings of the 40th International Conference on Software Engineering. p. 834–845. ICSE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180207, https://doi.org/10.1145/3180155.3180207
83. Saada, H., Huchard, M., Nebut, C., Sahraoui, H.: Recovering model transformation traces using multi-objective optimization. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE (nov 2013). https://doi.org/10.1109/ase.2013.6693134, https://doi.org/10.1109%2Fase.2013.6693134

84. Santiago, I., Vara, J.M., de Castro, V., Marcos, E.: Measuring the effect of enabling traces generation in ATL model transformations. In: Communications in Computer and Information Science, pp. 229–240. Springer Berlin Heidelberg (2013), https://doi.org/10.1007%2F978-3-642-54092-9_17
85. Santiago, I., Vara, J.M., de Castro, M.V., Marcos, E.: Towards the effective use of traceability in model-driven engineering projects. In: Conceptual Modeling. pp. 429–437. Berlin, Heidelberg (2013)
86. Santiago, I., Álvaro Jiménez, Vara, J.M., Castro, V.D., Bollati, V.A., Marcos, E.: Model-driven engineering as a new landscape for traceability management: A systematic literature review. Information and Software Technology **54**(12), 1340 – 1356 (2012). https://doi.org/https://doi.org/10.1016/j.infsof.2012.07.008, http://www.sciencedirect.com/science/article/pii/S0950584912001346, special Section on Software Reliability and Security
87. Schwarz, H., Ebert, J., Winter, A.: Graph-based traceability: a comprehensive approach. Software & Systems Modeling **9**(4), 473–492 (2010), https://doi.org/10.1007/s10270-009-0141-4
88. Seibel, A., Neumann, S., Giese, H.: Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. Software & Systems Modeling **9**(4), 493–528 (2010), https://doi.org/10.1007/s10270-009-0146-z
89. Seiler, M., Hübner, P., Paech, B.: Comparing traceability through information retrieval, commits, interaction logs, and tags. In: 2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST). pp. 21–28 (May 2019). https://doi.org/10.1109/SST.2019.00015
90. Shafiq, S., Mashkoor, A., Mayr-Dorn, C., Egyed, A.: A literature review of using machine learning in software development life cycle stages. IEEE Access **9**, 140896–140920 (2021)
91. Shin, Y., Hayes, J.H., Cleland-Huang, J.: Guidelines for benchmarking automated software traceability techniques. In: 2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability. pp. 61–67 (May 2015). https://doi.org/10.1109/SST.2015.13
92. Slotosch, O., Abu-Alqumsan, M.: Modeling and safety-certification of model-based development processes. In: Schaefer, I., Karagiannis, D., Vogelsang, A., Méndez, D., Seidl, C. (eds.) Modellierung 2018. pp. 261–273. Gesellschaft für Informatik e.V., Bonn (2018)
93. Spanoudakis, G., Zisman, A., Pérez-Miñana, E., Krause, P.: Rule-based generation of requirements traceability relations. Journal of Systems and Software **72**(2), 105 – 127 (2004). https://doi.org/https://doi.org/10.1016/S0164-1212(03)00242-5, http://www.sciencedirect.com/science/article/pii/S0164121203002425
94. Szabo, C., Chen, Y.: A model-driven approach for ensuring change traceability and multi-model consistency. In: 2013 22nd Australian Software Engineering Conference. IEEE (jun 2013). https://doi.org/10.1109/aswec.2013.24, https://doi.org/10.1109%2Faswec.2013.24
95. Tekinerdoğan, B., Hofmann, C., Akşit, M., Bakker, J.: Metamodel for tracing concerns across the life cycle. In: Moreira, A., Grundy, J. (eds.) Early Aspects: Current Challenges and Future Directions. pp. 175–194. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
96. Vale, T., de Almeida, E.S., Alves, V., Kulesza, U., Niu, N., de Lima, R.: Software product lines traceability: A systematic mapping study. Information and Software Technology **84**, 1 – 18 (2017). https://doi.org/https://doi.org/10.1016/j.infsof.2016.12.004, http://www.sciencedirect.com/science/article/pii/S0950584916304463
97. Vara, J.M., Bollati, V.A., Jiménez, Á., Marcos, E.: Dealing with traceability in the mddof model transformations. IEEE Trans. Software Eng. **40**(6), 555–583 (2014). https://doi.org/10.1109/TSE.2014.2316132, https://doi.org/10.1109/TSE.2014.2316132
98. von Knethen, A.: Change-oriented requirements traceability. support for evolution of embedded systems. In: International Conference on Software Maintenance, 2002. Proceedings. pp. 482–485 (Oct 2002). https://doi.org/10.1109/ICSM.2002.1167808

99. Watson, C., Cooper, N., Palacio, D.N., Moran, K., Poshyvanyk, D.: A systematic literature review on the use of deep learning in software engineering research (2020), arXiv-2009.06520
100. Watts, D.J.: Should social science be more solution-oriented? Nature Human Behaviour **1**(1), 0015 (2017), https://doi.org/10.1038/s41562-016-0015
101. Winkler, S., von Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Software and Systems Modeling **9**(4), 529–565 (2010), https://doi.org/10.1007/s10270-009-0145-0
102. Wohlrab, R., Knauss, E., Steghöfer, J.P., Maro, S., Anjorin, A., Pelliccione, P.: Collaborative traceability management: a multiple case study from the perspectives of organization, process, and culture. Requirements Engineering **25**(1), 21–45 (2020), https://doi.org/10.1007/s00766-018-0306-1
103. Ziegenhagen, D., Speck, A., Pulvermueller, E.: Expanding tracing capabilities using dynamic tracing data. In: Communications in Computer and Information Science, pp. 319–340. Springer International Publishing (2020), https://doi.org/10.1007/978-3-030-40223-5_16
104. Ziegenhagen., D., Speck., A., Pulvermüller., E.: Using developer-tool-interactions to expand tracing capabilities. In: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,. pp. 518–525. INSTICC, SciTePress (2019). https://doi.org/10.5220/0007762905180525

Construction of Verifier Combinations Based on Off-the-Shelf Verifiers

Dirk Beyer[1], Sudeep Kanav[1], and Cedric Richter[2]

[1] LMU Munich, Munich, Germany
[2] Carl von Ossietzky University, Oldenburg, Germany

Abstract. Software verifiers have different strengths and weaknesses, depending on properties of the verification task. It is well-known that combinations of verifiers via portfolio and selection approaches can help to combine the strengths. In this paper, we investigate (a) how to easily compose such combinations from *existing*, 'off-the-shelf' verification tools without changing them and (b) how much performance improvement easy combinations can yield, regarding the effectiveness (number of solved problems) and efficiency (consumed resources). First, we contribute a method to systematically and conveniently construct verifier combinations from existing tools, using the composition framework CoVeriTeam. We consider sequential portfolios, parallel portfolios, and algorithm selections. Second, we perform a large experiment on 8 883 verification tasks to show that combinations can improve the verification results *without* additional computational resources. All combinations are constructed from off-the-shelf verifiers, that is, we use them as published. The result of our work suggests that users of verification tools can achieve a significant improvement at a negligible cost (only configure our composition scripts).

Keywords: Software verification · Program analysis · Cooperative verification · Tool Combinations · Portfolio · Algorithm Selection · CoVeriTeam

1 Introduction

Automatic software verification has been an active area of research for many decades and various tools and techniques have been developed to solve the problem of verifying software [3, 7, 9, 25, 34, 37]. The research has also been adopted in practice [2, 22, 24, 39]. Each tool and technique has its own strengths in specific areas. In such a scenario, it becomes obvious to combine these tools to benefit from the strengths of individual tools, leading to a 'meta verifier' that solves more problems. Most current combination approaches are hardcoded, that is, the choice of the tools and the way to combine them is specifically programmed.

We contribute a method to construct combinations in a systematic way, independently from the set of tools to use. As for the types of combinations, we considered sequential and parallel portfolio [36], and algorithm selection [47]. The combinations are composed and executed with the tool CoVeriTeam [15].[1]

[1] https://gitlab.com/sosy-lab/software/coveriteam/

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 49–70, 2022.
https://doi.org/10.1007/978-3-030-99429-7_3

CoVeriTeam is a tool that is based on off-the-shelf atomic actors, which are executable units based on tool archives. It provides a simple language to construct tool combinations, and manages the download and execution of the existing tools on the provided input. CoVeriTeam provides a library of atomic actors for many well-known and publicly available verification tools. A new verification tool can be easily integrated into CoVeriTeam within a few minutes of effort.

For our experimental evaluation, we selected eight of the verification tools that participated in the 10th competition on software verification [6]. We reused the archives submitted to this competition, and composed combinations of three types (sequential and parallel portfolio, algorithm selection) with 2, 3, 4, and 8 verification tools: in total 12 combinations. We evaluated these 12 combinations on a large benchmark set consisting of 8 883 verification tasks in total and compared the results of the combinations against the results of the existing tools.

We show that all three combination approaches can lead to considerable improvements of the performance regarding effectiveness (number of correctly solved instances) and efficiency (consumed resources).

Contributions. We make the following contributions:

1. We show how to conveniently construct combination approaches from off-the-shelf verification tools in a modular manner, without changing the tools.
2. We perform an extensive comparative evaluation of sequential portfolio, parallel portfolio, and algorithm selection approaches.
3. A reproduction package containing the tools and experiment data.

2 Improving Verification by Verifier Combinations

In this study, we explore different strategies for combining verifiers to improve the overall verification effectiveness. We focus on the most commonly applied *black-box* combinations (i.e., combinations that do neither require any changes to the existing tools nor communication between verification tools) which we briefly describe in the following.

Verifier Combinations. Existing strategies for combining verifiers can be generally classified into one of the following three categories: *sequential portfolios* [17, 33, 53], *parallel portfolios* [35, 36, 40], and *algorithm selectors* [8, 28, 47, 48, 50]. We provide an overview over these composition strategies in Figs. 1 and 2.

Sequential Portfolio. Portfolios combine several verification algorithms by executing them either sequentially or in parallel. A sequential portfolio (Fig. 1) executes a set of verifiers sequentially by running one verifier after another. In this setting, each verifier is assigned a specific time limit and the verifier runs until it finds a solution or reaches the time limit. If the current verifier is able to solve the given verification task, the sequential composition is stopped and the solution is emitted. Otherwise, if a verifier runs into a timeout without, the current algorithm is stopped and the next one is started. CPA-Seq [17, 53] and Ultimate Automizer [33] are examples of sequential portfolios.

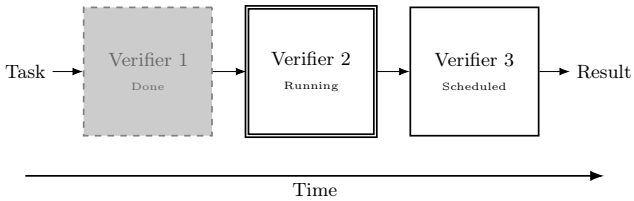


Fig. 1: Sequential portfolio of verifiers. Each verifier runs for a certain amount of time. If a verifier stops without computing a result (grey box), the next one is started (white box with double borders).

Parallel Portfolio. In contrast to sequential portfolios, a parallel portfolio (Fig. 2(a)) executes all verification algorithms in parallel, while sharing all system resources like CPU time and memory. As soon as one algorithm solves the given verification problem, the portfolio is stopped. Based on the assumption that all verifiers provide only sound solutions, we can safely take the first solution computed as the final result of the overall portfolio. PredatorHP [35, 40] is an example of a parallel portfolio.

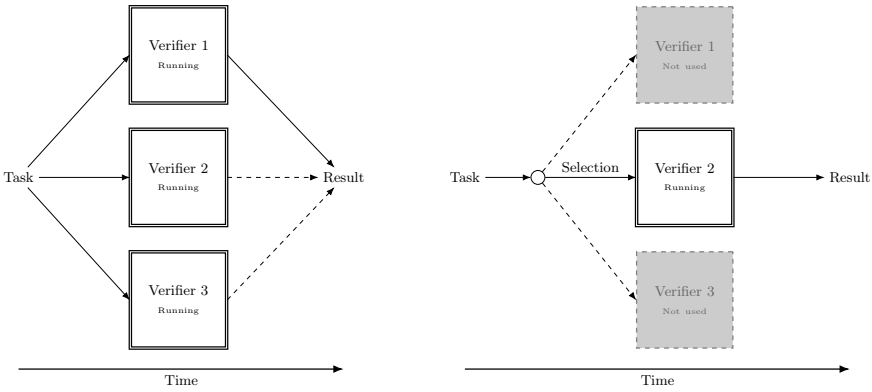
Algorithm Selection. To reduce spending resources on unsuccessful verifiers, algorithm selectors (Fig. 2(b)) are designed to select the verification algorithm that is likely well suited to solve a given verification task. More precisely, the algorithm selector analyzes the given verification problem for common characteristics (typically program features like the existence of a loop or an array) and based on these features, selects a verification algorithm likely suited for the given problem. Then the selected verifier is executed. Algorithm selectors were recently explored for selecting a task-dependent verification algorithm (e.g., in PeSCo [48, 50]) or a complete verification strategy (e.g., in CPAchecker [8]).

The above combination types have their own advantages and limitations when applied in real-world scenarios. While algorithm selectors omit the necessity of sharing resources, the approach heavily relies on the used selection algorithm. If the selection algorithm is not powerful enough or the selection task is too difficult, the selector fails to identify a verifier equipped for the given task. Although portfolios omit this problem by assigning the verification task to several verifiers, each verifier gets less resources, which could lead to out-of-resource failures.

3 Construction of Verifier Combinations with CoVeriTeam

CoVeriTeam [15] is a tool for creating and executing tool combinations for cooperative verification [20]. It consists of a language for tool composition, and an execution engine for this language. Tools are considered as verification actors (verifiers, validators, testers, transformers), and the inputs consumed and outputs produced by the tools as verification artifacts (programs, specifications, witnesses, results). Verification artifacts are seen as basic objects, verification actors as basic operations, and tool combinations as composition of these operations.

CoVeriTeam supports execution of most of the well known automated verification tools that are publicly available. The composition operators supported by



(a) Parallel Portfolio **(b)** Algorithm Selection

Fig. 2: Comparison of parallel portfolio and algorithm selection

CoVeriTeam are: SEQUENCE, PARALLEL, REPEAT, and ITE. SEQUENCE executes the composed tools sequentially, PARALLEL in parallel, REPEAT repeatedly till a termination condition is satisfied, ITE is an *if-then-else* that executes one tool if the provided condition is true and otherwise the other. The work in this paper uses SEQUENCE, PARALLEL, ITE, and a newly developed PORTFOLIO.

3.1 Verifier Based on Sequential Portfolio

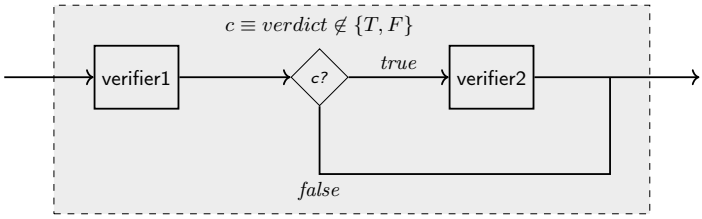


Fig. 3: Verifier based on sequential portfolio

Figure 3 shows the construction of a sequential portfolio of two verifiers verifier1 and verifier2 using CoVeriTeam. This construction uses two kinds of compositions: SEQUENCE and ITE. At the outermost level, it is a sequence of verifier1 and an actor that in itself is a composition—an ITE composition. Let us call it ite_verifier. When we execute this composition, first, verifier1 is executed and then ite_verifier. ite_verifier first checks if verifier1 was successful in verification or not (i.e., $verdict \notin \{T, F\}$). If verifier1 was successful, then it forwards the results, otherwise, verifier2 is executed and its results are taken. This construction can be generalized to create sequential portfolios of arbitrary sizes. We used it to create sequential portfolios of 2, 3, 4, and 8 verifiers.

3.2 Verifier Based on Parallel Portfolios

We developed a composition operator for parallel portfolio in CoVeriTeam. In this composition, multiple tools are executed in parallel and the result of the

one that succeeds first is taken. The composition consists of a set of verification actors of the same type (verifiers, testers, etc.), and a success condition defined over the artifacts produced by these actors. When one actor finishes, the success condition is evaluated: if it holds then the output of this actor is taken and the execution of the remaining actors is stopped. Otherwise, the portfolio waits for the next actor to finish and repeats the check. If none of the actors produce the output that satisfies the success condition, the result of the last one is taken.

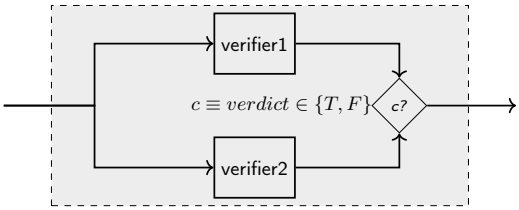


Fig. 4: Verifier based on parallel portfolio

Figure 4 shows a parallel portfolio of two verifiers verifier1 and verifier2. In this case, both the verifiers are executed simultaneously. When one verifier finishes, its result is checked for the success condition (i.e., $verdict \in \{T, F\}$). If the success condition holds then the result is forwarded, otherwise, the result is discarded and we wait for the second verifier to finish. Once a successful result is available, the remaining executing verifiers are terminated. For our experiments, we created parallel portfolios of 2, 3, 4, and 8 tools.

3.3 Verifier Based on Algorithm Selection

We designed and implemented a generic selection framework in CoVeriTeam for selecting verifiers. The framework decomposes the algorithm-selection process into two phases: (1) a *feature-extraction phase*, in which a feature encoder extracts a set of predefined features for a given verification task (i.e., certain characteristics that are believed to indicate difficulty for a verifier), and (2) *selection* to identify an appropriate verifier based on the extracted features. Each phase is constructed using CoVeriTeam actors (explained below in more detail). Figure 5 shows the CoVeriTeam composition of a verifier based on algorithm selection.

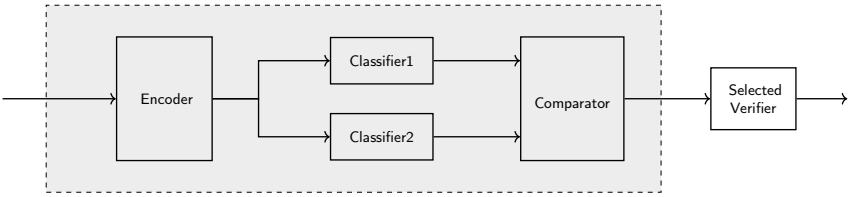


Fig. 5: Verifier based on algorithm selection

Feature Encoder. The first component of our framework is the feature *encoder*. Given a verification task consisting of a program P and a specification S, the goal of the feature encoder is to encode the problem into a meaningful feature-vector

(FV) representation, which we can later use to select a verification tool. Typically, the representation encodes certain features of a program which might correlate with the performance of a verifier such as the occurrence of specific loop patterns [28] or variable types [29]. In this study, we encode verification problems via a learning-based feature encoder by employing a pretrained *CSTTransformer* [50]. The CSTTransformer first parses a given program P into a simplified abstract syntax tree (AST) representation. Afterwards, a specific type of neural network processes the AST structure to produce a vector representation. The last encoding step is learned by pretraining the neural network on selecting various verification *tools*. While this approach was originally developed to learn a vector representation optimized for a specific verifier composition, the authors showed that the learned encoder can be effectively reused across many new selection tasks, often outperforming other hand-crafted feature encoders.

Selection of Verifiers Based on the Individual Difficulty of the Tasks. The same task might be solved with one tool in a few seconds, while another is not able to find a solution within the given resource constraints. Therefore, to avoid wasting resources on tools that are not well suited for a given task, the algorithm selector aims to predict the difficulty of a task before executing a tool. Then, the tool that is predicted to be the best suited tool for the task is executed.

Similar to previous work [28, 50], we learn to predict the difficulty of task with *hardness models* [55]. Based on the previously computed vector representation, a hardness model learns to predict the hardness of a given task for a specific tool. In our case, this reduces to a binary classification problem of predicting whether a tool can solve a task or not. We address this by training logistic regression classifiers. The classifier's confidence that a verifier will fail a particular task then determines the hardness of the task.

Now, given a set of hardness models —each accessing the hardness of a verification task for a specific tool— a verification tool is selected for which the task is likely easy (i.e., the respective model outputs the lowest hardness score). The final selection is done by a comparator implemented in CoVeriTeam that selects a tool by comparing the hardness scores.

3.4 Extensibility

To facilitate future research and the design of novel combinations, we implemented all combination types such that they can be easily configured and extended. Extending a combination with a new verifier only requires an actor definition for that verifier in CoVeriTeam. Afterwards, this actor can be put in a sequential or parallel portfolio by adding it to the composition. While our algorithm selector can be easily used with all tools employed during our experiments, extending a combination based on algorithm selection with a new verifier requires a bit more effort. However, by using hardness models together with a common feature representation we simplified the process required for configuring algorithm selection. In fact, we are able to modify the set of verifiers to select from by simply adding or removing individual hardness models. While previous approaches to

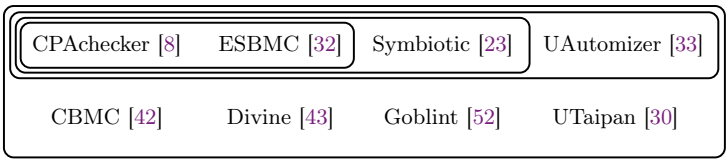


Fig. 6: Subsets of verification tools used for composition

verifier selection often require training the complete selector from scratch, our combination can be extended by training a single hardness model.[2] For training a new model, we provide all training scripts that were used for training our hardness models and a precomputed dataset of vector representations for SV-COMP 2021. Therefore, to integrate a new tool in our algorithm selector, one only requires to run the respective verifier once on (a subset of) the benchmark set. The results then act as training examples.

4 Evaluation

We perform a thorough experimental evaluation on a large benchmark set in order to show the potential of combinations. We address the following research questions concerning the comparative evaluation of combinations against standalone tools:

RQ 1. Can a CoVeriTeam-based sequential portfolio of verifiers perform significantly better than standalone tools with respect to
(a) number of solved verification tasks, and
(b) resource consumption?

RQ 2. Can a CoVeriTeam-based parallel portfolio of verifiers perform significantly better than standalone tools with respect to
(a) number of solved verification tasks, and
(b) resource consumption?

RQ 3. Can a CoVeriTeam-based algorithm selection of verifiers perform significantly better than standalone tools with respect to
(a) number of solved verification tasks, and
(b) resource consumption?

4.1 Experimental Setup

Selection of Existing Verifiers. We selected eight existing verification tools that performed well in a recent competition on software verification (SV-COMP 2021) [6]. We excluded two verifiers from consideration: VeriAbs [27] and PeSCo [49]. VeriAbs was excluded because its license does not allow us to use it for scientific evaluation, and PeSCo because it is a derivate of CPAchecker that would not contribute to diversity of technology in the combinations. The chosen set of verifiers used for the tool combinations is depicted in Fig. 6.

[2] A single hardness model can be trained within a few minutes on a modern CPU.

Tool Combinations. We evaluated twelve verifier combinations: for each of sequential portfolio, parallel portfolio, and algorithm selection, we constructed a combination of 2, 3, 4, and 8 verifiers. These variants of combinations with different numbers of verifiers allowed us to quantify the influence of the number of verifiers on the performance. We constructed these subsets of verifiers to maximize the number of tasks (from our benchmark set) that can be solved by at least one tool in the subset. For sequential portfolios, we additionally rank the verifiers in descending order of their success on the benchmark. We used the results from SV-COMP 2021 to achieve this. Figure 6 illustrates the sets of verifiers that we composed in different types of combinations.

Execution Environment. Our experiments were executed on machines with the following configuration: one 3.4 GHz CPU (IntelXeon E3-1230 v5) with 8 processing units (virtual cores), 33 GB RAM, operating system Ubuntu 20.04. Each verification run (execution of one tool or combination on one verification task) was limited to 8 processing units, 15 min of CPU time, and 15 GB memory. This configuration is the same as the configuration used in SV-COMP 2021 allowing us to use the competition results of the standalone tools for comparison.

Benchmark Selection. Our benchmark set consists of all the verification tasks with specification `unreach-call` from the open-source collection of verification tasks SV-Benchmarks[3]. Each verification task consists of a program written in C and a specification. The specification is a safety property describing that an error location should never be reached. The benchmark set includes all verification tasks of the competition categories *ReachSafety* and *Concurrency*, and a part of the verification tasks in category *SoftwareSystems*. In total, there were 8 883 verification tasks in our benchmark set. We evaluated our combinations on the version of the benchmark set that was used in SV-COMP 2021 (tag `svcomp21`).

Scoring Schema. We not only count the number of results of each kind[4] for the verification tasks, but also the scores as used in the competition, because this models what the community considers as quality. A verifier is rewarded score points as follows: 2 score points for each correct proof, 1 score point for each correct alarm, -32 score points for wrong proofs, and -16 score points for wrong alarms. This schema has been used in SV-COMP [6] since a few years and has been accepted by the verification community for judging the quality of results.

Resource Measurement and Benchmark Execution. We used the state-of-the-art benchmarking framework BenchExec [18] for executing our benchmarks. It executes tools in isolation, reports the resource consumption, and also enforces the resource limitations. It provides measurements of the consumption of CPU time, wall time, memory, and CPU energy during an execution of a tool.

4.2 Results of Existing Verifiers as Standalone

Table 1 shows the summary of results of the execution of the standalone tools on the selected benchmark set. These results are publicly available in the respective

[3] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks
[4] Either claims of program correctness or alarms of specification violations.

Table 1: Standalone verifiers

Verifier	CPAchecker	Esbmc	Symbiotic	UAutomizer	Cbmc	Divine	Goblint	UTaipan
Score	**9 040**	6 623	4 878	7 146	4 663	3 679	2 770	5 338
Correct results	**5 652**	4 481	3 001	4 358	3 484	2 922	1 385	3 725
Correct proofs	**3 516**	2 958	1 909	2 836	1 499	1 605	1 385	2 365
Correct alarms	2 136	1 523	1 092	1 522	1 985	1 317	0	1 360
Wrong results	8	29	2	2	19	41	**0**	24
Wrong proofs	**0**	22	**0**	1	1	12	**0**	23
Wrong alarms	8	7	2	**1**	18	29	0	**1**
Total resource consumption for correct results								
CPU time (h)	190	57	22	97	31	60	**11**	81
Wall time (h)	140	57	22	59	31	15	**11**	52
Memory (GB)	7 000	1 800	770	4 300	1 300	2 000	**120**	2 700
CPU Energy (KJ)	7 700	2 500	1 000	3 500	1 300	1 500	**560**	3 000
Median resource consumption for correct results								
CPU time (s)	61	0.84	0.81	36	**0.70**	17	0.78	39
Wall time (s)	32	0.84	0.84	12	**0.69**	9.1	0.80	13
Memory (MB)	600	53	**25**	450	44	670	**25**	430
CPU Energy (J)	590	11	11	310	**9.2**	150	11	330
Resource consumption of correct results per score point								
CPU time (s/sp)	77	31	16	49	24	59	**15**	55
Wall time (s/sp)	55	31	16	30	24	**14**	15	35
Memory (MB/sp)	780	270	160	600	280	540	**42**	500
Energy (J/sp)	850	380	210	490	280	420	**200**	560

reproduction package of the competition [5] and on the competition web site[5]. We only adjust the presentation to our needs here.

Figure 7 shows the quantile plots of the results, where the x-coordinate represents the quantile of score obtained by the tool below the run time represented by y-coordinate. We used a logarithmic scale for time ranges between 1 and 1000 seconds, and linear scale between 0 and 1 second. The graph of a tool that solves more verification tasks will be farther to the right, and the plot of the faster tools would be lower. The farther on the right side a plot goes and the lower a plot remains. the better it is. More details about these plots are given elsewhere [4].

Figure 8 shows the resource consumption for standalone tools using a parallel-coordinates plot (each parallel coordinate represents a different variable). The plot shows the number of unsolved tasks, and resource consumption per score point. The lower the plot of a tool is the better it is for the user.

4.3 RQ 1: Evaluation of Sequential-Portfolio Verifier

We now present the results of the sequential-portfolio verifier against the existing standalone verifier with the highest score: CPAchecker.

[5] https://sv-comp.sosy-lab.org/2021/results/results-verified

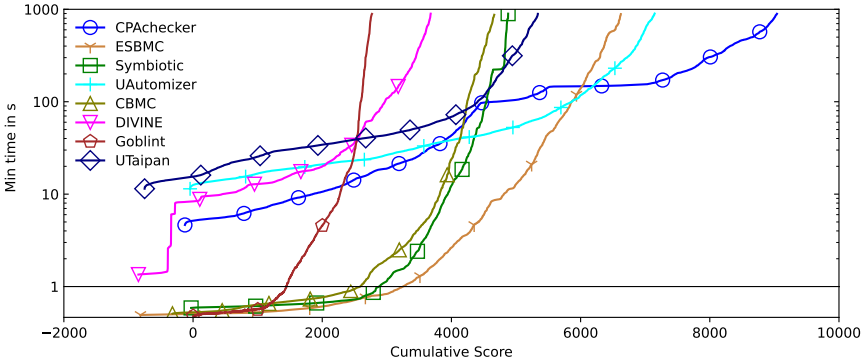


Fig. 7: Standalone verifiers: Score-based quantile plot for results

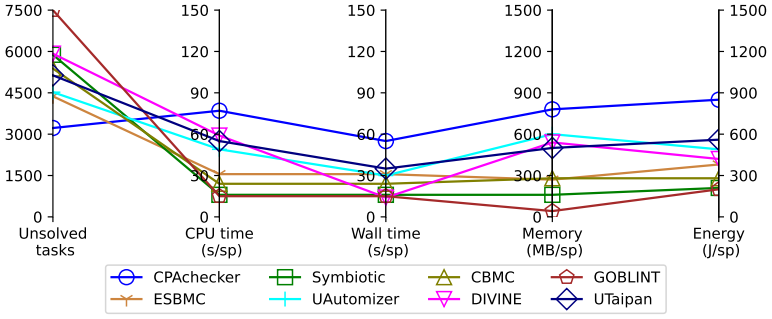


Fig. 8: Standalone verifiers: Parallel-coordinates plot showing unsolved tasks and resource consumption per score point

Table 2 shows the summary of results for the sequential verifiers. The sequential portfolio, in general achieves better score than the best performing standalone tool. The portfolio with 8 tools performs worst, which is expected because as we increase the size of the portfolio, the amount of time allocated to each verifier also decreases. This means that the verifiers can only solve relatively easier tasks. The table also shows that the portfolio requires more resources to solve the tasks. This is a side effect of the sequential portfolio, as all the resources consumed by unsuccessful attempts to solve a given task by the verifiers in a sequence are still counted in the resource consumption. Also, the portfolio with 8 tools has a considerably large number of wrong results as it is reduced to fast results, instead of the verifier earlier in the sequence. The index at which a verifier is placed plays a key role in the performance of the sequential portfolio. If we put a verifier that produces results fast but has more wrong results first in the sequential portfolio, then the overall results are going to have a lot of wrong results.

Figure 9 shows the quantile plot of scores. As a portfolio is biased towards the verifiers that compute results fast and not towards correctness, we see the sequential portfolio combinations starting from farthest in the left, i.e., having the most negative score, or most wrong results. CPAchecker has the least number of wrong results, and because of it its starting point is farthest to the right.

Table 2: Sequential portfolios of different sizes with CPAchecker

Verifier	CPAchecker	Sequential Portfolio of 2	3	4	8
Score	9 040	9 198	9 519	**9 522**	8 349
Correct results	5 652	6 058	6 239	**6 275**	6 084
Correct proofs	3 516	3 780	**3 920**	3 903	3 721
Correct alarms	2 136	2 278	2 319	**2 372**	2 363
Wrong results	**8**	26	26	27	61
Wrong proofs	**0**	14	14	14	30
Wrong alarms	**8**	12	12	13	31
Total resource consumption for correct results					
CPU time(h)	**190**	240	260	240	190
Wall time (h)	**140**	190	210	190	150
Memory (GB)	**7 000**	8 900	8 600	8 500	7 600
CPU Energy (KJ)	**7 700**	9 700	11 000	10 000	7 900
Median resource consumption for correct results					
CPU time(s)	**61**	95	100	100	97
Wall time (s)	**32**	54	69	70	54
Memory (MB)	**600**	920	930	910	840
CPU Energy (J)	**590**	920	1 100	1 100	920
Resource consumption of correct results per score point					
CPU time (s/sp)	**77**	95	97	90	82
Wall time (s/sp)	**55**	72	78	72	64
Memory (MB/sp)	**780**	970	910	890	920
CPU Energy (J/sp)	**850**	1 100	1 100	1 100	950

Figure 10 shows that CPAchecker is more resource efficient in comparison to the sequential portfolio. The sequential combination with best score is performing worst in resource efficiency.

4.4 RQ 2: Evaluation of Parallel-Portfolio Verifier

We now present the results of the parallel-portfolio verifiers. The parallel portfolio, mostly, achieves worse score than the best performing standalone tool. But the parallel portfolio with 3 tools scores better. The parallel portfolio is affected by two aspects: (1) size of the parallel portfolio — if too many tools are used then any of them would not get enough resources to verify the task, (2) selection of tools — if there is a fast tool that produces a lot of wrong results it reduces the score. Parallel portfolio, in general, produces more wrong results; even more than sequential portfolio, as the tools are running in parallel, whereas in sequential portfolio this can be somewhat mitigated by putting a more sound tool before a less sound tool. Table 3 shows the summary of results for the parallel portfolios.

Figure 11 shows that parallel portfolios have many more wrong results when compared to CPAchecker. Interestingly, the graph for *ParPortfolio-3*, the best performing parallel portfolio, remains lower than CPAchecker, i.e., it takes less

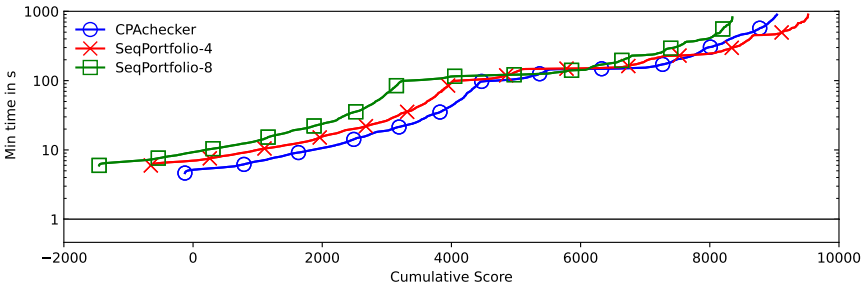


Fig. 9: Sequential portfolios: Score-based quantile plot comparing the best and the worst sequential portfolio (SeqPortfolio-4 and SeqPortfolio-8, respectively) with the best performing standalone tool (CPAchecker)

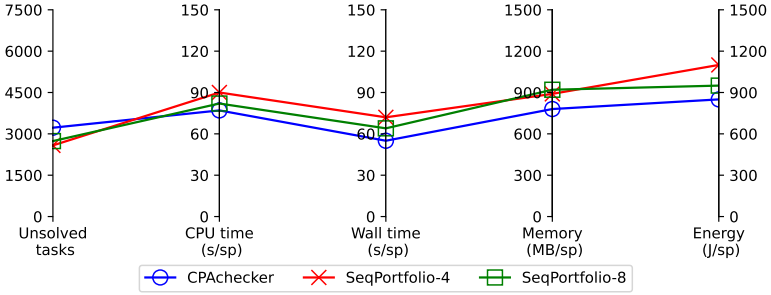


Fig. 10: Sequential portfolios: Parallel-coordinates plot showing unsolved tasks and resource consumption per score point for best and worst portfolio (SeqPortfolio-4 and SeqPortfolio-8, resp.) and the best standalone tool (CPAchecker)

CPU time. This is because the parallel portfolio takes results of the most efficient tool. Figure 12 shows that the best performing parallel portfolio performs better than CPAchecker in terms of resource efficiency except memory consumption.

4.5 RQ 3: Evaluation of Algorithm Selection Verifier

We now present the results of the algorithm-selection verifier. Table 4 shows the summary of results for algorithm selection: There is a clear trend of better results with more verifiers. This is expected because our selector that was trained using machine learning has more options to choose from, and can choose the better one. Also, an algorithm-selection verifier does not need to share resources for the verification task. It needs to perform the prediction, which takes some resources; but after this step all the provided resources are available to the verifier. The number of wrong results is also comparable with CPAchecker, as the training process is biased towards selecting the verifiers that are correct.

In Fig. 13, all the plots start from around similar scores but at different times. Initially, CPAchecker performs better with respect to CPU time, but after around half the scores, algorithm selection starts being more efficient. Figure 14 shows that algorithm selection is also more resource efficient than CPAchecker.

Table 3: Parallel portfolios of different size with CPAchecker

Verifier	CPAchecker	Parallel Portfolio of 2	3	4	8
Score	9 040	8 969	**9 459**	8 952	7 547
Correct results	5 652	6 101	**6 363**	6 001	5 367
Correct proofs	3 516	3 780	**3 992**	3 639	3 236
Correct alarms	2 136	2 321	**2 371**	2 362	2 131
Wrong results	**8**	36	35	28	42
Wrong proofs	**0**	21	21	15	24
Wrong alarms	**8**	15	14	13	18
Total resource consumption for correct results					
CPU time(h)	190	**160**	170	250	280
Wall time (h)	140	74	**61**	74	64
Memory (GB)	**7 000**	8 900	11 000	14 000	11 000
CPU Energy (KJ)	7 700	5 400	**5 200**	6 500	6 400
Median resource consumption for correct results					
CPU time(s)	61	18	**16**	70	130
Wall time (s)	32	5.2	**4.6**	16	23
Memory (MB)	600	430	**420**	1 000	1 300
CPU Energy (J)	590	140	**120**	470	780
Resource consumption of correct results per score point					
CPU time (s/sp)	77	**65**	66	99	130
Wall time (s/sp)	55	30	**23**	30	31
Memory (MB/sp)	780	**1 000**	1 200	1 500	1 400
CPU Energy (J/sp)	850	600	**550**	720	850

4.6 Discussion

The experiments show that each of the compositions has a configuration that can perform better than any standalone tool in terms of correctly solved tasks. Initially, we thought that portfolios would be less resource efficient than standalone tools, and, in particular, would not be able to solve hard tasks as the resources allocated to each tool would be less. But the experimental data support the opposite: The benchmark set had a few such tasks: for most of the tasks that were hard for one tool, there was some other tool that solved it in the given time. This was especially pronounced in the parallel portfolio. The verifiers in the portfolios have to be selected with different strengths, otherwise there is no benefit, it might even perform worse.

Both the portfolios prefer fast results, as there is no selector. To mitigate this, one needs to either select the tools carefully or add a validation step.

Our algorithm selection was based on a model trained using machine learning. The training penalized the tools that produced more incorrect results, but it did not consider the resource consumption of these tools. In comparison to both the portfolios, the verifier based on algorithm selection produced much less incorrect

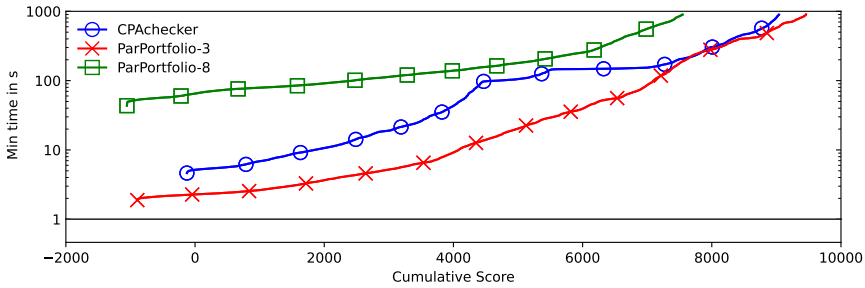


Fig. 11: Parallel portfolios: Score-based quantile plot comparing the best and the worst performing parallel portfolios (ParPortfolio-3 and ParPortfolio-8, respectively) with the best performing standalone tool (CPAchecker)

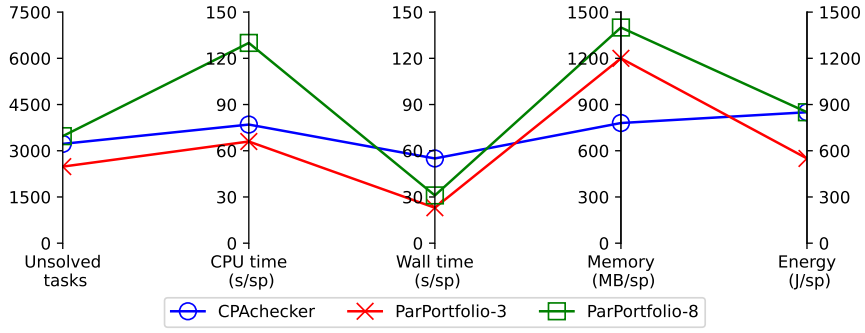


Fig. 12: Parallel portfolios: Parallel-coordinates plot showing unsolved tasks and resource consumption per score point of best and worst portfolio (ParPortfolio-3 and ParPortfolio-8, resp.) and the best standalone tool (CPAchecker)

results. We think if we used the resource consumption data in our training, the verifier based on selection would have consumed less resources. Our verifier combinations are easy to construct by simply selecting tools that complement each other well. Although this strategy is simple, we found that it still leads to successful combinations for all evaluated combination types. Nevertheless, the combinations can be further fine-tuned to achieve even better results.

The portfolio compositions are easy to construct, and with a well diversified tool selection, portfolios can perform good. Also, the portfolios should not be too large unless we are willing to increase the resources. On the other hand, training the selection requires more preliminary work but with limited resources and enough choice (number of tools) the selection-based verifier works better.

5 Threats to Validity

External Validity. A combination of tools can only be as good as the parts it is combined from. Therefore, the concrete instantiation of our tool combinations is limited by the selected tools and their configuration. We have selected eight of the

Table 4: Algorithm-selection-based verifiers of different sizes with CPAchecker

Verifier	CPAchecker	Algorithm Selection of 2	3	4	8
Score	9 040	9 226	9 689	9 816	**9 886**
Correct results	5 652	5 904	6 086	6 125	**6 214**
Correct proofs	3 516	3 658	3 843	3 867	**3 896**
Correct alarms	2 136	2 246	2 243	2 258	**2 318**
Wrong results	**8**	15	11	8	11
Wrong proofs	**0**	6	4	3	3
Wrong alarms	**8**	9	7	5	**8**
Total resource consumption for correct results					
CPU time(h)	**190**	200	200	200	210
Wall time (h)	**140**	160	160	150	170
Memory (GB)	7 000	6 900	6 900	6 200	**6 000**
CPU Energy (KJ)	**7 700**	8 200	8 600	8 400	9 000
Median resource consumption for correct results					
CPU time(s)	61	**47**	48	66	55
Wall time (s)	32	**30**	**30**	35	42
Memory (MB)	600	740	700	550	**420**
CPU Energy (J)	**590**	490	500	660	620
Resource consumption of correct results per score point					
CPU time (s/sp)	77	77	76	**73**	76
Wall time (s/sp)	**55**	61	61	56	63
Memory (MB/sp)	780	750	720	630	**600**
CPU Energy (J/sp)	**850**	890	890	**850**	910

most powerful verification tools as determined by the annual software-verification competition, and executed them in the original configuration as submitted to the competition. Furthermore, our evaluation results only hold for the given benchmark set. While we have evaluated our tool combinations on programs taken from one of the largest and diverse verification benchmarks publicly available, the performance of the evaluated combinations might differ on other sets of tasks.

Similarly, this also impacts the training of our algorithm selector. The training of a learning-based algorithm selector, which we employ for tool combinations based on algorithm selection, requires a large and diverse set of verification tasks; and each task has to be labeled with the execution results of each tool in our combination. The used benchmarks repository[6] was created by the efforts of the verification community over many years. We are not aware of any other benchmark set of verification tasks that is as diverse as this one. As a result, we had to train our algorithm selector on the same dataset that we later use for benchmarking the tool combinations. Therefore, we only showed that algorithm selection improves the performance of verification on the given benchmark set

[6] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

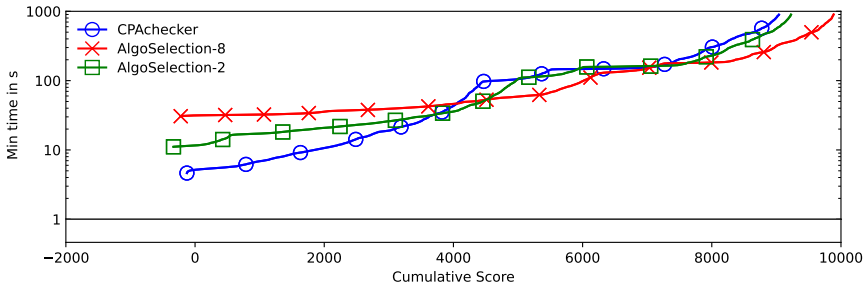


Fig. 13: Algorithm-selection-based verifiers: Score-based quantile plot comparing the best and the worst performing portfolio (AlgoSelection-8 and AlgoSelection-3, respectively) with the best performing standalone tool (CPAchecker)

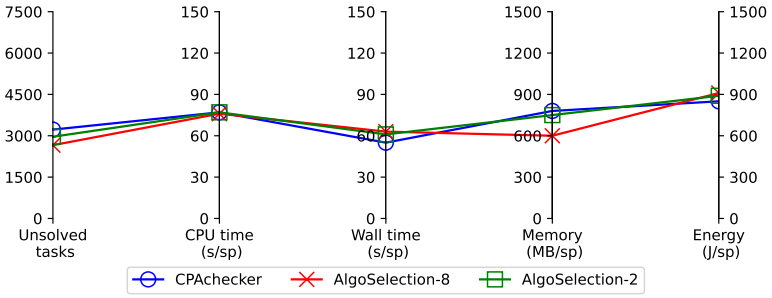


Fig. 14: Algorithm-selection-based verifiers: Parallel-coordinates plot showing unsolved tasks and resource consumption per score point of the best and the worst performing algorithm selection (AlgoSelection-8 and AlgoSelection-2, respectively) and the best performing standalone tool (CPAchecker)

and the selector might only generalize to a set of tasks with similarly distributed verification tasks. For a fair comparison, we (1) restricted the training to linear models, which are known to generalize well, (2) train only on a random subset of the benchmark, and (3) cross validated our model over multiple benchmark splits. The variance of selection performance between different splits was less than 1 %. Therefore, the performance of our trained algorithm selector is likely independent of the random subset selected for training.

Finally, the evaluation of algorithm selection is dependent on the chosen selection methodology and choosing alternative selection methods, for example, based on hand-crafted rules, might impact the evaluation. However, the design of hand-crafted methods is not straightforward and might require deep expert knowledge about the tool implementation. Depending on the human designer, this design process might in addition be biased in favor of certain tool combinations, which could also impact the experimental results.

For sequential portfolios, we ordered verifiers in sequence according to their performance in SV-COMP 2021. Changing the order of the tools might change the results with respect to resource consumption as well as soundness.

Internal Validity. We have used the same verifier archives, benchmark set, benchmarking framework, resource limits, and infrastructure to execute our experiments as was used in SV-COMP 2021. This minimizes the influence of a changing environment on our experiments, allowing us to compare results of our verifier combinations to the results of the standalone tools from SV-COMP 2021.

CoVeriTeam induces an overhead of about 0.8 s for each actor in the composition, and around 44 MB memory overhead [15]. It is possible that one can reduce this overhead by using shell scripts, but we decided in favor of using CoVeriTeam for composing tools because of the modular design. This is especially pronounced in our algorithm-selector composition. We could have saved a few seconds if we were using a monolithic algorithm selector instead of composing one.

6 Related Work

Combination Strategies for Software Verification. Combining verifiers to increase the verification performance is well established in the domain of software verification [1, 8, 20, 26, 31, 33, 46, 48, 49, 53]. In fact, the top three winning entries of the software-verification competition SV-COMP 2021 all combine various verification techniques to achieve their performance [6]. CPAchecker [8] combines up to six different verification appraoches into three sequential portfolios that are task-dependently selected with an algorithm selector. PeSCo [49] ranks verification algorithms according to their predicted likelihood of solving a given task and then executes them sequentially in descending order. Ultimate Automizer [33] employs an integrated tool chain of preprocessing and verification algorithm to solve a given task. PredatorHP [46] and UFO [1] demonstrate that parallel portfolios can also be a promising strategy when running multiple specialized algorithms at the same time. Even though previous work showed that internal combinations can be successfully applied to improve the effectiveness of a single tool, we show that similar combinations can be effectively employed to combine 'off-the-shelf' verifiers. This gives us the unique opportunity to further increase the number of verifiable programs by simply combining state-of-the-art verification tools.

Cooperative methods [20] distribute the workload of a single verification task among multiple algorithms to combine their strengths. For example, conditional model checking [11, 12, 13, 14] runs two or more verifiers in sequence, while the program is reduced after every step to the state space of program unexplored by the previous algorithm. CoVeriTest [10], a tool for test-case generation based on verification, interleaves multiple verifiers, while (partially) sharing the analysis state between algorithms. MetaVal [19] integrates verification tools for witness validation (i.e., to check whether a previous verifier obtained a comprehensible result) by instrumenting the produced witness into the verified program. While cooperative methods are effective for reducing the workload of a verification task, employing cooperative methods at tool level would require to exchange analysis information between tools. In general, existing verification tools are not well suited for this type of cooperation, which lead us to explore black-box verifier

combinations. In addition, we showed that non-cooperative methods can improve the verification effectiveness without the need to adapt the employed tools.

Combining Algorithms Beyond Software Verification. The idea of combining algorithms to improve performance have been successfully applied in many research areas including SAT solving [51, 54, 56], constraint-satisfaction programs [21, 45, 57] and combinatorial-search problems [41]. Employed approaches traditionally focused on portfolio-based approaches [21, 51, 54], but recent techniques started to integrate algorithm selectors for either selecting single algorithms [45, 56] or portfolios of algorithms [44, 57]. For example, earlier works in SAT solving [51, 54] focused on parallel-portfolio solvers, while later works such as SATzilla [56] further improves the solving process by selecting a task-dependent solver. However, existing techniques often employ hybrid strategies between portfolios and algorithm selection to achieve state-of-the-art performance. Therefore, Kashgarani and Kothoff [38] have recently shown that parallel portfolios are generally bottlenecked by the available resources and that a pure algorithm selector that selects a single algorithm performs better. While we observed that portfolios of software verifiers are also restricted by available resources (i.e., the performance generally stops to improve after a certain portfolio size), we found that all evaluated combination types yield a similar performance gain when configured correctly.

7 Conclusion

This paper describes a method to construct combinations of verification tools in a systematic and modular way. The method does not require any changes to the verification tools that are used to construct the combinations. Our experimental evaluation shows that all three considered combinations (sequential portfolio, parallel portfolio, and algorithm selection) can lead to performance improvements. The improvements can be significant although the construction does not require significant development effort, because we use CoVeriTeam for the combination and execution of verification tools. We hope that our contribution makes it easy for practitioners to get access to the best performance out of the latest research and development efforts in software verification.

Declarations

Data Availability Statement. A reproduction package including all our results is available at Zenodo [16]. Additionally, the result tables are also available on a supplementary web page for convenient browsing.[7]

Funding Statement. This work was funded in part by the Deutsche Forschungsgesellschaft (DFG) — 418257054 (Coop).

Acknowledgement. We thank Tobias Kleinert for implementing the parallel portfolio combination in CoVeriTeam.

[7] https://www.sosy-lab.org/research/coveriteam-combinations

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A framework for abstraction- and interpolation-based software verification. In: Proc. CAV, pp. 672–678. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_48
2. Ball, T., Rajamani, S.K.: The Slam project: Debugging system software via static analysis. In: Proc. POPL. pp. 1–3. ACM (2002). https://doi.org/10.1145/503272.503274
3. Beckert, B., Hähnle, R.: Reasoning and verification: State of the art and current trends. IEEE Intelligent Systems **29**(1), 20–29 (2014). https://doi.org/10.1109/MIS.2014.3
4. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS. pp. 594–609. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_43
5. Beyer, D.: Results of the 10th Intl. Competition on Software Verification (SV-COMP 2021). Zenodo (2021). https://doi.org/10.5281/zenodo.4458215
6. Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: Proc. TACAS (2). pp. 401–422. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_24, preprint available.
7. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. LNCS 13244, Springer (2022)
8. Beyer, D., Dangl, M.: Strategy selection for software verification based on boolean features: A simple but effective approach. In: Proc. ISoLA. pp. 144–159. LNCS 11245, Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_11
9. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
10. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
11. Beyer, D., Jakobs, M.C.: Fred: Conditional model checking via reducers and folders. In: Proc. SEFM. pp. 113–132. LNCS 12310, Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_7
12. Beyer, D., Jakobs, M.C., Lemberger, T.: Difference verification with conditions. In: Proc. SEFM. pp. 133–154. LNCS 12310, Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_8
13. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
14. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Combining verifiers in conditional model checking via reducers. In: Proc. SE. pp. 151–152. LNI P-292, GI (2019). https://doi.org/10.18420/se2019-46
15. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. Springer (2022)
16. Beyer, D., Kanav, S., Richter, C.: Reproduction Package for Article 'Construction of Verifier Combinations Based on Off-the-Shelf Verifiers'. Zenodo (2022). https://doi.org/10.5281/zenodo.5812021
17. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16

18. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
19. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
20. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
21. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Twenty-First International Joint Conference on Artificial Intelligence (2009)
22. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
23. Chalupa, M., Jašek, T., Novák, J., Řechtáčková, A., Šoková, V., Strejček, J.: Symbiotic 8: Beyond symbolic execution (competition contribution). In: Proc. TACAS (2). pp. 453–457. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_31
24. Chong, N., Cook, B., Eidelman, J., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow at Amazon Web Services. Softw. Pract. Exp. **51**(4), 772–797 (2021). https://doi.org/10.1002/spe.2949
25. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
26. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
27. Darke, P., Agrawal, S., Venkatesh, R.: VeriAbs: A tool for scalable verification by abstraction (competition contribution). In: Proc. TACAS (2). pp. 458–462. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32
28. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. Formal Methods in System Design **50**(2-3), 289–316 (2017). https://doi.org/10.1007/s10703-016-0264-5
29. Demyanova, Y., Veith, H., Zuleger, F.: On the concept of variable roles and its use in software analysis. In: Proc. FMCAD. pp. 226–230. IEEE (2013). https://doi.org/10.1109/FMCAD.2013.6679414
30. Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: Ultimate Taipan with symbolic interpretation and fluid abstractions (competition contribution). In: Proc. TACAS (2). pp. 418–422. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_32
31. Filliâtre, J.C., Paskevich, A.: Why3: Where programs meet provers. In: Programming Languages and Systems. pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
32. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: Esbmc v6.0: Verifying C programs using k-induction and invariant inference (competition contribution). In: Proc. TACAS (3). pp. 209–213. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_15

33. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants (competition contribution). In: Proc. TACAS (2). pp. 447–451. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
34. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. J. ACM **50**(1), 63–69 (2003). https://doi.org/10.1145/602382.602403
35. Holík, L., Kotoun, M., Peringer, P., Šoková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: Proc. HVC. pp. 202–209. LNCS 10028 (2016). https://doi.org/10.1007/978-3-319-49052-6_13
36. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(7), 51–54 (1997). https://doi.org/10.1126/science.275.5296.51
37. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
38. Kashgarani, H., Kotthoff, L.: Is algorithm selection worth it? comparing selecting single algorithms and parallel execution. In: AAAI Workshop on Meta-Learning and MetaDL Challenge. pp. 58–64. PMLR (2021)
39. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
40. Kotoun, M., Peringer, P., Šoková, V., Vojnar, T.: Optimized PredatorHP and the SV-COMP heap and memory safety benchmark (competition contribution). In: Proc. TACAS. pp. 942–945. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_66
41. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach, pp. 149–190. LNCS 10101, Springer (2016). https://doi.org/10.1007/978-3-319-50137-6_7
42. Kröning, D., Tautschnig, M.: Cbmc: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
43. Lauko, H., Štill, V., Ročkai, P., Barnat, J.: Extending Divine with symbolic verification using SMT (competition contribution). In: Proc. TACAS (3). LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_14
44. Lindauer, M., Hoos, H., Hutter, F.: From sequential algorithm selection to parallel portfolio selection. In: International Conference on Learning and Intelligent Optimization. pp. 1–16. Springer (2015). https://doi.org/10.1007/978-3-319-19084-6_1
45. Minton, S.: Automatically configuring constraint satisfaction programs: A case study. Constraints **1**(1-2), 7–43 (1996). https://doi.org/10.1007/BF00143877
46. Peringer, P., Šoková, V., Vojnar, T.: PredatorHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_30
47. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976). https://doi.org/10.1016/S0065-2458(08)60520-3
48. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x

49. Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
50. Richter, C., Wehrheim, H.: Attend and represent: a novel view on algorithm selection for software verification. In: Proc. ASE. pp. 1016–1028 (2020). https://doi.org/10.1145/3324884.3416633
51. Roussel, O.: Description of ppfolio (2011). Proc. SAT Challenge p. 46 (2012)
52. Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints (competition contribution). In: Proc. TACAS (2). pp. 438–442. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_28
53. Wendler, P.: CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proc. TACAS. pp. 613–615. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_45
54. Wotzlaw, A., van der Grinten, A., Speckenmeyer, E., Porschen, S.: pfoliouzk: Solver description. Proceedings of SAT Challenge p. 45 (2012)
55. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hierarchical hardness models for SAT. In: International Conference on Principles and Practice of Constraint Programming. pp. 696–711. Springer (2007). https://doi.org/10.1007/978-3-540-74970-7_49
56. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. JAIR **32**, 565–606 (2008). https://doi.org/10.1613/jair.2490
57. Yun, X., Epstein, S.L.: Learning algorithm portfolios for parallel execution. In: International Conference on Learning and Intelligent Optimization. pp. 323–338. Springer (2012). https://doi.org/10.1007/978-3-642-34413-8_23

On the Detection of Doped Software by Falsification*

Sebastian Biewer[1](✉) and Holger Hermanns[1,2]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
biewer@depend.uni-saarland.de
[2] Institute of Intelligent Software, Guangzhou, China

Abstract. Software doping is a phenomenon that refers to the presence of hidden software functionality, whose existence is only in the interest of the manufacturer. The most prominent example is the diesel emissions scandal. There is a need for methods that identify software doping, and such methods are bound to be applied to the final product with no or rare knowledge about its internals. Black-box analysis techniques have recently been developed for this purpose, harvesting the formal foundations of software doping. This paper integrates them with established falsification techniques for the purpose of real-world applicability. With a focus on the diesel scandal and emissions tests on chassis dynamometers we make the testing procedures significantly more effective in terms of time and cost. The theoretical results are implemented in a prototypical doping tester.

1 Introduction

Embedded software is the innovation driver of our times. Software-defined systems are permeating our communication, perception, and storage technology as well as our personal interactions with technical systems at an unprecedented pace. *"Software-defined everything"* is among the hottest buzzwords in IT technology today [2,18].

There is a tremendous problem hiding behind this apparently unstoppable trend: The owners of the physical "hull" of *everything* will not be the ones owning the software defining *everything*, nor will they have the right to look at what and how *everything* is defined. This is because commercial software typically is protected by intellectual property rights of the software manufacturer. This prohibits any attempt to disassemble the software or to reconstruct its inner working, albeit it is the very software that is forecasted to be defining *everything*. The use of machine-learnt software components amplifies the problem considerably. Since commercial interests of the software manufacturers seldomly are aligned

* This work is partly supported by DFG grant 389792660 as part of TRR 248 – CPEC, the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233, and by the Key-Area Research and Development Program Grant 2018B010107004 of Guangdong Province.

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 71–91, 2022.
https://doi.org/10.1007/978-3-030-99429-7_4

with the interest of end users, the promise of *software-defined everything* might well become a dystopia from the perspective of individual digital sovereignty.

A massive example of software-defined collective damage is the diesel emissions scandal. Over a period of more than 10 years, millions of diesel-powered cars have been equipped with illegal software that altogether polluted the environment for the sake of commercial advantages of the car manufacturers. At its core, this was made possible by the fact that only a single, precisely defined test setup was put in place for checking conformance with exhaust emissions regulations. This made it a trivial software engineering task to identify the test particularities and to turn off emission cleaning outside these particular conditions. This is an archetypical instance of software doping.

Against this background, there is an urgent need to establish stronger and enforceable requirements on the systems we are interacting with, and this is indeed echoed in legislatory frameworks [24]. However, the roll-out of such requirements in everyday practice needs a firm understanding of the technological basis for enforcing such requirements, respectively for identifying violations thereof.

This paper is part of ongoing research addressing this challenge. It harvests the outcomes of three recent scientific achievements: (*i*) formal definitions of software doping based on contracts enforcing well-defined software behaviour in the vicinity of standardised behaviour [12], (*ii*) a solid foundation for doping tests to be carried out in practice [6], and (*iii*) probabilistic falsification techniques developed to guide the search for property violations in cyber-physical system engineering [1, 20]. By combining the above ingredients, this paper addresses the question how to perform cost-effective doping tests that are indeed likely to succeed in uncovering actual cases of doped software. It approaches this question both from a foundational and from a practical perspective. On the foundational side, we introduce a temporal hyperlogic to reason about signals which we use to characterise the falsifiable fragment of a software doping contract. Great care is taken for this to work on the actual time-discrete traces that are recorded from the real system which itself is running in continuous time. On the practical side, we discuss a novel approach to probabilistic falsification that overcomes the problem that in many practical cases the possibility to carry out masses of highly-controlled experiments with a physical system is severely limited by cost or time budgets. To account for this, we add a passive recording component to the concept of falsification which observes the system in-the-wild to propose only few candidate traces to be inspected under lab conditions. All this is instantiated in the context of automotive emissions, where lab conditions correspond to expensive test runs on a chassis dynamometer, while observing the system in-the-wild is nothing else than collecting statistics while driving on normal roads.

The paper makes the following distinguished contributions: (*i*) a linear temporal logic for hyperproperties over continuous signals that enables quantitative reasoning across traces, (*ii*) a logical reformulation of the falsifiable fragment of a software doping contract, (*iii*) a probabilistic falsification technique that uses passive recording for cost-effective doping testing, and (*iv*) an exemplary instantiation of these concepts in the context of automotive emissions.

Related Work. Software doping theory provides a formal basis for enlarging the requirements on vehicle exhaust emissions beyond too narrow lab test conditions. That conceptual limitation has by now been addressed by the official authorities responsible for car type approval [24,25]: The old NEDC-based test procedure is replaced by the newer *Worldwide Harmonised Light Vehicles Test Procedure* (WLTP), which is deemed to be more realistic. WLTP replaces the NEDC test by a new WLTC test, but WLTC still is just a single test scenario. In addition, WLTP embraces so called *Real Driving Emissions* (RDE) tests to be conducted on public roads. A recently launched mobile phone app [8], LolaDrives, harvests runtime monitoring technology for making low-cost RDE tests accessible to everyone.

Learning or approximating the behaviour of a system under test has been studied intensively. Meinke and Sindhu [19] were among the first to present a testing approach incrementally learning a Kripke structure representing a reactive system. Volpato and Tretmans [27] propose a learning approach which gradually refines an under- and over-approximation of an input-output transition system representing the system under test. The correctness of this approach needs several assumptions, e.g., an oracle indicating when, for some trace, all outputs, which extend the trace to a valid system trace, have been observed.

2 Background

This section introduces the necessary background regarding temporal logics for hyperproperties and for continuous signals, probabilistic falsification basics, and reviews the formal definitions of software doping.

2.1 Temporal Logics

Linear Temporal Logic (LTL) [22] is a popular formalism to reason about properties of traces. A trace is an infinite word where each literal is a subset of AP, the set of atomic propositions. Programs are interpreted as sets $\mathsf{S}_{\mathsf{LTL}} \subseteq (2^{\mathsf{AP}})^\omega$ of such traces. LTL provides expressive means to characterise sets of traces, often called *trace properties*.

Temporal Logics for Hyperproperties. For some set of traces T, a trace property defines a subset of T, whereas a *hyperproperty* defines a *set of* subsets of T. In this way it specifies which traces are valid in combination with one another. Many temporal logics have been extended to corresponding hyperlogics supporting the specification of hyperproperties. HyperLTL [11] is such a temporal logic for the specification of hyperproperties of reactive systems. It extends LTL with trace quantifiers and trace variables that make it possible to refer to multiple traces within a logical formula. A *HyperLTL formula* is defined by the following grammar where π is drawn from a set $\mathcal{V}$ of *trace variables*:

$$\begin{aligned} \psi &::= \exists \pi.\, \psi \mid \forall \pi.\, \psi \mid \phi \\ \phi &::= a_\pi \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{X}\,\phi \mid \phi\,\mathsf{U}\,\phi \end{aligned}$$

The quantifiers $\exists$ and $\forall$ quantify existentially and universally, respectively, over the set of traces. For example, the formula $\forall\pi.\,\exists\pi'.\,\phi$ means that for every trace π there exists another trace π' such that ϕ holds over the pair of traces. To account for distinct valuations of atomic propositions across distinct traces, the atomic propositions are indexed with trace variables: for some atomic proposition $a \in \mathsf{AP}$ and some trace variable $\pi \in \mathcal{V}$, a_π states that a holds in the initial position of trace π. The temporal operators and Boolean connectives are interpreted as usual for LTL. In particular, $\mathsf{X}\,\phi$ means that ϕ holds in the next state of every trace under consideration. Likewise, $\phi\,\mathsf{U}\,\phi'$ means that ϕ' eventually holds in every trace under consideration at the same point in time, provided ϕ holds in every previous instant in all such traces. Further operators are derivable: $\mathsf{F}\,\phi \equiv \mathsf{true}\,\mathsf{U}\,\phi$ enforces ϕ to eventually hold in the future, $\mathsf{G}\,\phi \equiv \neg\,\mathsf{F}\,\neg\phi$ enforces ϕ to always hold, and the weak-until operator $\phi\,\mathsf{W}\,\phi' \equiv \phi\,\mathsf{U}\,\phi' \vee \mathsf{G}\,\phi$ allows ϕ to always hold as an alternative to the obligation for ϕ' to eventually hold. We refer to [11] for the formal semantics.

Temporal Logics over Continuous Domains. LTL enables reasoning over traces $\sigma \in (2^{\mathsf{AP}})^\omega$ which are of discrete nature with respect to the time domain they represent. With each literal in the trace representing a time step, σ can equivalently be viewed as a function $\mathbb{N} \to 2^{\mathsf{AP}}$. One extension of LTL is *Signal Temporal Logic* (STL) [13,17], which instead is used for reasoning over real-valued signals that may change in value along an underlying time domain. A signal is a function $s : \mathcal{T} \to \mathbb{R}$ where $\mathcal{T}$ is the time domain. The time domain $\mathcal{T}$ can be either $\mathbb{N}$ (*discrete-time* signals), or $\mathbb{R}_{\geq 0}$ (*continuous-time* signals). This can be lifted to multi-dimensional signals $w(t) = (s_1(t), \ldots, s_n(t))$, mapping each time point to some element of $\mathbb{R}^n$. We refer to such a $w : \mathcal{T} \to \mathbb{R}^n$ as a (discrete-time or continuous-time) *trace* of *width* n in the sequel.

STL formulas can express properties of systems modelled as sets $\mathsf{S}_{\mathsf{STL}} \subseteq (\mathcal{T} \to \mathbb{R}^n)$ of traces of some fixed width n, basically by making the atomic properties refer to booleanizations of the signal values. The syntax of the variant of STL that we use in this paper is as follows, where $f \in \mathbb{R}^n \to \mathbb{R}$:

$$\phi ::= \top \mid f > 0 \mid \neg\phi \mid \phi \wedge \phi \mid \phi\,\mathsf{U}\,\phi\,.$$

STL replaces atomic propositions by *threshold predicates* of the form $f > 0$, which hold if and only if function f applied to the signal values at the current time returns a positive value. The Boolean operators and the Until operator U are very similar to those of HyperLTL. The Next operator X is not part of STL, because "next" is without precise meaning in continuous time. The definitions of the derived operators F, G and W are the

$$\begin{aligned}
&w, t \models \top \\
&w, t \models f > 0 \quad &&\text{iff} \quad f(s_1(t), \ldots, s_n(t)) > 0 \\
&w, t \models \neg\phi \quad &&\text{iff} \quad w, t \not\models \phi \\
&w, t \models \phi \wedge \psi \quad &&\text{iff} \quad w, t \models \phi \text{ and } w, t \models \psi \\
&w, t \models \phi\,\mathsf{U}\,\psi \quad &&\text{iff} \quad \text{exists } t' \geq t \text{ s.t. } w, t' \models \psi \text{ and} \\
& && \quad \text{for all } t'' \in [t, t'),\ w, t'' \models \phi
\end{aligned}$$

Fig. 1: Boolean semantics of STL formulas

same as for HyperLTL. Formally, the *Boolean semantics* of an STL formula ϕ at time point $t \in \mathcal{T}$ for a trace $w = (s_1, \ldots, s_n)$ is defined inductively in Fig. 1.

Quantitative Interpretation. STL has been extended by a *quantitative semantics* [1,13,14] as presented in Fig. 2. This semantics is designed in such a way that whenever $\rho(\phi, w, t) \neq 0$, its sign indicates whether $w, t \models \phi$ holds in the Boolean semantics. For any STL formula ϕ, trace w and time t, if $\rho(\phi, w, t) > 0$, then $w, t \models \phi$ holds, and if $\rho(\phi, w, t) < 0$, then $w, t \models \phi$ does not hold. For the scope of this paper, we work with the untimed Until operator, instead of allowing $\mathsf{U}_{[a,b]}$ for arbitrary bounds $a, b \in \mathbb{R}$. With only the untimed Until operator, the continuous and discrete semantics [14] coincide.

$$
\begin{aligned}
\rho(\top, w, t) &= \infty \\
\rho(f > 0, w, t) &= f(s_1(t), \ldots, s_n(t)) \\
\rho(\neg\phi, w, t) &= -\rho(\phi, w, t) \\
\rho(\phi \wedge \psi, w, t) &= \min(\rho(\phi, w, t), \rho(\psi, w, t)) \\
\rho(\phi \,\mathsf{U}\, \psi, w, t) &= \sup_{t' \geq t} \min\{\rho(\psi, w, t'), \inf_{t'' \in [t, t')} \rho(\phi, w, t'')\}
\end{aligned}
$$

Fig. 2: Quantitative semantics of STL formulas

Robustness and Falsification. The value of the quantitative semantics can serve as a *robustness estimate* and as such be used to search for a violation of the property at hand, i.e., to falsify it. The robustness of STL formula ϕ is its quantitative value at time 0, that is, $\mathcal{R}_\phi(w) := \rho(\phi, w, 0)$. So, falsifying a formula ϕ for a system $\mathsf{S_{STL}}$ boils down to a search problem with the goal condition $\mathcal{R}_\phi(w) < 0$. Successful falsification algorithms solve this problem by understanding it as the optimisation problem $\mathsf{minimise}_{w \in \mathsf{S_{STL}}} \mathcal{R}_\phi(w)$. Algorithm 1 [1,20] sketches an algorithm for Monte-Carlo Markov Chain falsification, which is based on acceptance-rejection sampling [10]. Our version of the algorithm works on system traces instead of an input space. An input to the algorithm is an initial trace w and a computable robustness function $\mathcal{R}$. Robustness computation for finite timed traces of simulations of a system has been discussed in the literature [13,14]; we omit this discussion here. The third input PS is a proposal scheme that proposes a new trace to the algorithm based on the previous one (line 2). The parameter β (used in line 3) can be adjusted during the search and is a means to avoid being trapped in local minima, preventing to find a global minimum. Any two traces w and $w' \in \mathsf{S_{STL}}$ with robustness values $\mathcal{R}(w)$ and $\mathcal{R}(w')$ are sampled with probability proportional to

Algorithm 1 Monte-Carlo falsification

Input: w: Initial trace, $\mathcal{R}$: Robustness function, PS: Proposal Scheme
Output: $w \in \mathsf{S_{STL}}$

1: **while** $\mathcal{R}(w) > 0$ **do**
2: $w' \leftarrow \mathsf{PS}(w)$
3: $\alpha \leftarrow \exp(-\beta(\mathcal{R}(w') - \mathcal{R}(w)))$
4: $r \leftarrow \mathsf{UniformRandomReal}(0, 1)$
5: **if** $r \leq \alpha$ **then**
6: $w \leftarrow w'$
7: **end if**
8: **end while**

$\frac{e^{-\beta\mathcal{R}(w)}}{e^{-\beta\mathcal{R}(w')}}$ (lines 3-6). The algorithm seeks to minimise $\mathcal{R}$ over the system's traces $\mathsf{S}_{\mathsf{STL}}$, and terminates when it finds a trace with a negative robustness value, i.e., a trace that violates the STL property from which $\mathcal{R}$ is derived.

2.2 Software Doping

Contracts and Robustness. Earlier work [12] has developed a formal basis for the purpose of characterising software doping, by providing precise definitions of when the system's behaviour is *clean*, i.e., does not contain hidden functionalities not in the interest of the user. If a program exhibits behaviour that is not clean, it is *doped*.

All cleanness definitions are based on the assumption that there is some well-defined and agreed standard input/output behaviour of the system. *Robust cleanness*, the cleanness definition that we work with in this paper, extends this behaviour to the vicinity around the inputs and outputs close to the standard behaviour. The definition of "vicinity" and of "standard behaviour" is assumed to be part of a *contract* between software manufacturer and user. The contract entails the standard behaviour, distance functions for input and output values, and distance thresholds to define the input and output vicinity, respectively. With this, a system behaviour is considered clean, if its output is (or stays) in the output vicinity of the standard, unless the input is (or moves) outside the standard's input vicinity.

Example 1. A concrete contract for diesel-powered cars will, for instance, enforce bounded deviations in exhaust emissions provided the driving profile stays in the bounded vicinity of the standardised tests (such as NEDC or WLTC). Recent experiments [6] have considered contracts based on NEDC with speed values as inputs and NO_x emissions as output values, together with distance functions computing the absolute difference of speed inputs and NO_x outputs, respectively, and value thresholds were 15 km/h for inputs and 80 mg/km for outputs.

A function $d : X \times X \to \mathbb{R}_{\geq 0}$ is a pseudometric function if it satisfies $d(x, x) = 0$, $d(x, y) = d(y, x)$ and $d(x, y) \leq d(x, z) + d(z, y)$ for all x, y, $z \in X$. We let $\sigma[k]$ denote the k-th literal of the infinite word σ.

Reactive Execution Model. We can view a (nondeterministic) reactive program as a function $\mathsf{S}_\mathsf{R} : \mathsf{In}^\omega \to 2^{(\mathsf{Out}^\omega)}$ perpetually mapping inputs In to sets of outputs Out [12]. A *contract* is a tuple $\mathcal{C} = \langle \mathsf{StdIn}, d_\mathsf{In}, d_\mathsf{Out}, \kappa_\mathsf{i}, \kappa_\mathsf{o} \rangle$ where $\mathsf{StdIn} \subseteq \mathsf{In}^\omega$ is the input space of the system designated to define the standard behaviour, $d_\mathsf{In} : (\mathsf{In}^* \times \mathsf{In}^*) \to \mathbb{R}_{\geq 0}$ and $d_\mathsf{Out} : (\mathsf{Out}^* \times \mathsf{Out}^*) \to \mathbb{R}_{\geq 0}$ are pseudometric distance functions on finite words over inputs, respectively outputs, and $\kappa_\mathsf{i} \in \mathbb{R}_{\geq 0}$ is a constant defining the maximum distance to the standard input allowed, and similarly $\kappa_\mathsf{o} \in \mathbb{R}_{\geq 0}$ is the maximum distance between two outputs such that they are still considered sufficiently close. For the purpose of this paper, we assume the distance functions to be induced by pointwise pseudometric functions of the form $d_\mathsf{In} : (\mathsf{In} \times \mathsf{In}) \to \mathbb{R}_{\geq 0}$ and $d_\mathsf{Out} : (\mathsf{Out} \times \mathsf{Out}) \to \mathbb{R}_{\geq 0}$ in a past-forgetful manner.

Definition 1. *A reactive program* $\mathsf{S_R} : \mathsf{In}^\omega \to 2^{(\mathsf{Out}^\omega)}$ *is* robustly clean *w.r.t. to contract* $\mathcal{C} = \langle \mathsf{StdIn}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_\mathsf{i}, \kappa_\mathsf{o} \rangle$ *if for all input sequences* $\mathsf{i}, \mathsf{i}' \in \mathsf{In}^\omega$ *with* $\mathsf{i} \in \mathsf{StdIn}$*, it holds for arbitrary* $k \geq 0$ *that whenever* $d_{\mathsf{In}}(\mathsf{i}[j], \mathsf{i}'[j]) \leq \kappa_\mathsf{i}$ *for all* $j \leq k$*, then*

1. *for all* $\mathsf{o} \in \mathsf{S_R}(\mathsf{i})$ *there exists* $\mathsf{o}' \in \mathsf{S_R}(\mathsf{i}')$ *such that* $d_{\mathsf{Out}}(\mathsf{o}[k], \mathsf{o}'[k]) \leq \kappa_\mathsf{o}$*, and*
2. *for all* $\mathsf{o}' \in \mathsf{S_R}(\mathsf{i}')$ *there exists* $\mathsf{o} \in \mathsf{S_R}(\mathsf{i})$ *such that* $d_{\mathsf{Out}}(\mathsf{o}[k], \mathsf{o}'[k]) \leq \kappa_\mathsf{o}$.

The definition enforces that whenever an input i' remains within κ_i vicinity around the standard input i, then the output sets generated by i and i' are at most κ_o away from each other.

HyperLTL Characterisation. D'Argenio et al. [12] prove that the following two HyperLTL formulas characterise robust cleanness in the sense of Definition 1.

$$\forall \pi_1. \forall \pi_2. \exists \pi_2'.\ \mathsf{StdIn}_{\pi_1} \rightarrow \Big(\mathsf{G}(\mathsf{i}_{\pi_2} = \mathsf{i}_{\pi_2'}) \wedge \big((\hat{d}_{\mathsf{Out}}(\mathsf{o}_{\pi_1}, \mathsf{o}_{\pi_2'}) \leq \kappa_\mathsf{o}) \,\mathsf{W}\, (\hat{d}_{\mathsf{In}}(\mathsf{i}_{\pi_1}, \mathsf{i}_{\pi_2'}) > \kappa_\mathsf{i}) \big) \Big) \tag{1}$$

$$\forall \pi_1. \forall \pi_2. \exists \pi_1'.\ \mathsf{StdIn}_{\pi_1} \rightarrow \Big(\mathsf{G}(\mathsf{i}_{\pi_1} = \mathsf{i}_{\pi_1'}) \wedge \big((\hat{d}_{\mathsf{Out}}(\mathsf{o}_{\pi_1'}, \mathsf{o}_{\pi_2}) \leq \kappa_\mathsf{o}) \,\mathsf{W}\, (\hat{d}_{\mathsf{In}}(\mathsf{i}_{\pi_1'}, \mathsf{i}_{\pi_2}) > \kappa_\mathsf{i}) \big) \Big) \tag{2}$$

The non-atomic propositions in the formulas above are syntactic sugar; the input and output values in system $\mathsf{S_{LTL}}$ give rise to a binary encoding into sets of atomic propositions.

Mixed-IO Model. The reactive execution model and the HyperLTL characterisation above have the strict requirement that for every input, the system produces exactly one output. Recent work [5,6] instead considers mixed-IO models, where a program $\mathsf{S_{IO}} \subseteq (\mathsf{In} \cup \mathsf{Out})^\omega$ is a subset of traces containing both inputs and outputs, but without any restriction on the order or frequency in which inputs and outputs appear in the trace. In particular, they are not required to strictly alternate (but they may, and in this way the reactive execution model can be considered a special case). A particularity of this model is the distinct output symbol δ for quiescence, i.e., the absence of an output. For example, finite behaviour can be expressed by adding infinitely many δ symbols to a finite trace. In this model, standard behaviour is captured by subset $\mathsf{Std} \subseteq \mathsf{S_{IO}}$ of traces of a system $\mathsf{S_{IO}}$. To capture the notion of robust cleanness in the mixed-IO model, every trace is projected into an input, respectively output domain. The set of input symbols contains one additional element $-_\mathsf{i}$, that indicates that in the respective steps an output was produced, but masking the concrete output. Similarly, the set of output symbols contains the additional element $-_\mathsf{o}$ to mask a concrete input symbol. *Projection on inputs* $\downarrow_\mathsf{i}\colon (\mathsf{In} \cup \mathsf{Out})^\omega \to (\mathsf{In} \cup \{-_\mathsf{i}\})^\omega$ and *projection on outputs* $\downarrow_\mathsf{o}\colon (\mathsf{In} \cup \mathsf{Out})^\omega \to (\mathsf{Out} \cup \{-_\mathsf{o}\})^\omega$ are defined for all traces $\sigma \in (\mathsf{In} \cup \mathsf{Out})^\omega$ and $k \in \mathbb{N}$ as follows: $\sigma{\downarrow_\mathsf{i}}[k] :=$ **if** $\sigma[k] \in \mathsf{In}$ **then** $\sigma[k]$ **else** $-_\mathsf{i}$ and similarly $\sigma{\downarrow_\mathsf{o}}[k] :=$ **if** $\sigma[k] \in \mathsf{Out}$ **then** $\sigma[k]$ **else** $-_\mathsf{o}$. The distance functions

$\bar{d}_{\mathsf{In}}$ and $\bar{d}_{\mathsf{Out}}$ apply on input and output symbols or their respective masks, i.e. they are pseudometrics in $(\mathsf{In} \cup \{\neg_\mathsf{i}\}) \times (\mathsf{In} \cup \{\neg_\mathsf{i}\}) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ and, respectively, $(\mathsf{Out} \cup \{\neg_\mathsf{o}\}) \times (\mathsf{Out} \cup \{\neg_\mathsf{o}\}) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. As for the reactive model, we define a contract formally as a tuple $\mathcal{C} = \langle \mathsf{Std}, \bar{d}_{\mathsf{In}}, \bar{d}_{\mathsf{Out}}, \kappa_\mathsf{i}, \kappa_\mathsf{o} \rangle$ (where StdIn is replaced by Std, d_{In} by $\bar{d}_{\mathsf{In}}$, and d_{Out} by $\bar{d}_{\mathsf{Out}}$). Its satisfaction is defined by the adapted robust cleanness definition below [6].

Definition 2. *A system* $\mathsf{S}_{\mathsf{IO}} \subseteq (\mathsf{In} \cup \mathsf{Out})^\omega$ *is* robustly clean *w.r.t. contract* $\mathcal{C} = \langle \mathsf{Std}, \bar{d}_{\mathsf{In}}, \bar{d}_{\mathsf{Out}}, \kappa_\mathsf{i}, \kappa_\mathsf{o} \rangle$ *if and only if* $\mathsf{Std} \subseteq \mathsf{S}_{\mathsf{IO}}$ *and for all* $\sigma \in \mathsf{Std}$, $\sigma' \in \mathsf{S}_{\mathsf{IO}}$ *and* $k \geq 0$ *it holds that whenever* $\bar{d}_{\mathsf{In}}(\sigma[j]{\downarrow_\mathsf{i}}, \sigma'[j]{\downarrow_\mathsf{i}}) \leq \kappa_\mathsf{i}$ *for all* $j \leq k$ *then*

1. *there exists* $\sigma'' \in \mathsf{S}_{\mathsf{IO}}$ *such that* $\sigma'{\downarrow_\mathsf{i}} = \sigma''{\downarrow_\mathsf{i}}$ *and* $\bar{d}_{\mathsf{Out}}(\sigma[k]{\downarrow_\mathsf{o}}, \sigma''[k]{\downarrow_\mathsf{o}}) \leq \kappa_\mathsf{o}$,
2. *there exists* $\sigma'' \in \mathsf{Std}$ *such that* $\sigma{\downarrow_\mathsf{i}} = \sigma''{\downarrow_\mathsf{i}}$ *and* $\bar{d}_{\mathsf{Out}}(\sigma'[k]{\downarrow_\mathsf{o}}, \sigma''[k]{\downarrow_\mathsf{o}}) \leq \kappa_\mathsf{o}$.

Def. 2 contains two requirements, numbered as *1.* and *2.*. In the following, we will sometimes explicitly address either of these conditions by referring to it as the *first*, respectively *second condition of robust cleanness*.

3 Logical characterisation for mixed IO

This section discusses how to reformulate robust cleanness to make it amenable to probabilistic falsification. For this, we translate eq. (2) into a HyperSTL formula, subsequentially remove its quantifiers by means of a highly efficient parallel composition on the level of traces and, finally, carefully adapt this quantifier-free representation to the mixed-IO model.

Hyperlogics over Continuous Domains. Previous work [21] extends STL to HyperSTL echoing the extension of LTL to HyperLTL. A major challenge of the robustness computation for HyperSTL formulas is the adequate handling of the continuous time domain when comparing two execution traces of a system. For systems that can be simulated, this can be avoided [21] by composing one or more copies of the simulation model in parallel to itself [11]. Snapshots of the composed system are effectively snapshots of the individual copies of the model at exactly the same time point. This approach is not available when interacting with (black-box) real-world cyber-physical systems (CPS). In such scenarios, a suitable logics is HyperSTL* [7], an extension of STL* [9], which enables the comparison of different time points in different traces by means of a freeze operator. We use a variant of this idea, but with a HyperSTL syntax similar to [21].

$$\begin{aligned} \psi &::= \exists \pi.\, \psi \mid \forall \pi.\, \psi \mid \phi \\ \phi &::= \top \mid f > 0 \mid \neg\phi \mid \phi \wedge \phi \mid \phi \,\mathsf{U}\, \phi \,. \end{aligned}$$

The meaning of the universal and existential quantifier is as for HyperLTL. A crucial difference to the other logics presented above is the proposition $f > 0$. In contrast to HyperLTL and to the existing definition of HyperSTL, we consider it insufficient to allow propositions to refer to only a single trace. In HyperLTL that does not cause harm, because atomic propositions of individual traces can

be compared by means of the Boolean connectives. To formulate thresholds for real values, however, we feel the need to allow real values from multiple traces to be combined in the function f, and thus to appear as arguments of f. Hence, in our semantics of HyperSTL, $f > 0$ holds if and only if the result of f, applied to all traces quantified over, is greater than 0. For this to work formally, the arity of function f is the product of the trace width n and the number m of traces quantified over at the occurrence of $f > 0$ in the formula, so $f : (\mathbb{R}^n)^m \to \mathbb{R}$.

A trace assignment [11] $\Pi : \mathcal{V} \to \mathsf{S}_{\mathsf{STL}}$ is a partial function assigning traces of $\mathsf{S}_{\mathsf{STL}}$ to variables. Let $\Pi[\pi := w]$ denote the same function as Π, except that π is mapped to trace w. The quantitative semantics of a HyperSTL formula ψ, at time point $t \in \mathcal{T}$, for a system $\mathsf{S} \subseteq (\mathcal{T} \to \mathbb{R}^n)$ and a trace assignment Π is defined inductively:

$$
\begin{aligned}
\rho(\exists \pi.\ \psi, \mathsf{S}, \Pi, t) &= \max_{w \in \mathsf{S}} \rho(\psi, \mathsf{S}, \Pi[\pi := w], t) \\
\rho(\forall \pi.\ \psi, \mathsf{S}, \Pi, t) &= \min_{w \in \mathsf{S}} \rho(\psi, \mathsf{S}, \Pi[\pi := w], t) \\
\rho(\top, \mathsf{S}, \Pi, t) &= \infty \\
\rho(f > 0, \mathsf{S}, \Pi, t) &= f(\Pi(\pi_1)(t), \ldots, \Pi(\pi_m)(t)) \\
&\quad \text{for } dom(\Pi) = \{\pi_1, \ldots, \pi_m\}^1 \\
\rho(\neg\phi, \mathsf{S}, \Pi, t) &= -\rho(\phi, \mathsf{S}, \Pi, t) \\
\rho(\phi_1 \wedge \phi_2, \mathsf{S}, \Pi, t) &= \min(\rho(\phi_1, \mathsf{S}, \Pi, t), \rho(\phi_2, \mathsf{S}, \Pi, t)) \\
\rho(\phi_1 \,\mathsf{U}\, \phi_2, \mathsf{S}, \Pi, t) &= \sup_{t' \geq t} \min\{\rho(\phi_2, \mathsf{S}, \Pi, t'), \inf_{t'' \in [t, t')} \rho(\phi_1, \mathsf{S}, \Pi, t'')\}
\end{aligned}
$$

It is an easy exercise to show that for continuous-time signals this quantitative semantics of HyperSTL is a conservative extension of the quantitative semantics of STL discussed above. For discrete-time signals it is important to understand that discrete time points often represent points in continuous time. It is widely accepted, that this can be cast into a (strictly monotonic) timing function $\tau : \mathbb{N} \to \mathbb{R}_{\geq 0}$ [3, 14]. The HyperSTL semantics given above is meaningful in a discrete-time setting if all traces share the same timing function.

HyperSTL characterisation. As discussed in Section 2.2, robust cleanness is a hyperproperty. Recent work on testing and monitoring of robust cleanness [6] explains the difficulties of monitoring such hyperproperties. In essence, it turns out that the first condition of Definition 2 cannot be refuted by observing a real system. Intuitively, this is because this condition effectively puts a constraint on the lower bound of the size of the sets of outputs that a system must be able to produce whereas the second condition enforces an upper bound. A violation of the upper-bound constraint is irrevocable, i.e., once observed, the system is for sure not robustly clean. However not having observed an output that is larger than the lower bound, does not exclude the possibility for observing such an output in the future. We therefore follow [6], and focus only on the second

[1] We admit some sloppiness; the set $dom(\Pi)$ should have a fixed order.

condition of the robust cleanness definition in our work on falsification. For the HyperLTL characterisation this means that we only work with the second formula, labelled (2).

The HyperLTL characterisation (2) assumes the system to be a subset of $(2^{\mathsf{AP}})^\omega$ and works with distances between traces by means of a Boolean encoding into atomic propositions. We will describe how to transform the HyperLTL formula (2) into a HyperSTL formula, where systems are given as subsets of $(\mathcal{T} \to \mathbb{R}^n)$ for some width $n \in \mathbb{N}$. Robust cleanness distinguishes between inputs and outputs, and we assume that the input set In and the output set Out are represented as signals of width m, respectively width l. The system space then is $\mathsf{S}_{\mathsf{STL}} \subseteq (\mathcal{T} \to \mathbb{R}^{m+l})$. Solely for the sake of clarity, we will in the sequel, unless otherwise stated, restrict to $m = l = 1$, i.e., $\mathsf{In} \subseteq \mathbb{R}$ and $\mathsf{Out} \subseteq \mathbb{R}$, and thus work with a fixed width of 2, hence $\mathsf{S}_{\mathsf{STL}} \subseteq (\mathcal{T} \to \mathsf{In} \times \mathsf{Out})$.

We can assume a set $\mathsf{Std} \subseteq \mathsf{S}_{\mathsf{STL}}$ as given, which defines all standard behaviours of the system. The HyperSTL characterisation of the HyperLTL formula (2) is then

$$\begin{aligned} &\forall \pi_1.\, \forall \pi_2.\, \exists \pi_1'.\ \mathsf{Std}_{\pi_1} > 0 \rightarrow \\ &\quad \Big(\mathsf{G}(|\mathsf{i}_{\pi_1} - \mathsf{i}_{\pi_1'}| \leq 0 \wedge \mathsf{Std}_{\pi_1'} > 0) \wedge \\ &\qquad \big((d_{\mathsf{Out}}(\mathsf{o}_{\pi_1'}, \mathsf{o}_{\pi_2}) - \kappa_{\mathsf{o}} \leq 0)\ \mathsf{W}\ (d_{\mathsf{In}}(\mathsf{i}_{\pi_1'}, \mathsf{i}_{\pi_2}) - \kappa_{\mathsf{i}} > 0) \big) \Big) \end{aligned} \tag{3}$$

The quantifiers remain unchanged relative to (2). The predicate StdIn_{π_1} that holds if and only if π_1 is a standard input, is replaced by the function Std_{π_1} which returns a positive value if π_1 is in Std, and a non-positive value otherwise. The input equality requirement of π_1 and π_1' is ensured by globally enforcing $|\mathsf{i}_{\pi_1} - \mathsf{i}_{\pi_1'}| \leq 0$.

Since we switched from the concept of standard inputs to the concept of standard traces, we must also check that π_1' is a standard trace. This echoes the setup in Definition 2, where the second requirement asks for a trace $\sigma'' \in \mathsf{Std}$ instead of a trace from S_{IO}, see [6] for an elaborate discussion. In the operands of the Weak-Until operator W, we replace the AP-encoded versions of $\hat{d}_{\mathsf{In}}$ and $\hat{d}_{\mathsf{Out}}$ by the original distance functions d_{In} and d_{Out}, and we perform simple arithmetic operations to match the syntactic requirements of HyperSTL.

We remark that for encoding Std_π, due to the absence of the Next-operator in HyperSTL, it might be necessary to add a clock signal $s(t) = t$ to traces in a preprocessing step, not considered here for the sake of avoiding cluttered notation.

Quantifier Elimination. In many practical settings—where the different standard behaviours are spelled out upfront explicitly, as in NEDC and WLTC—it can be assumed that the number of distinct standard behaviours Std is finite (while there are infinitely many possible behaviours in $\mathsf{S}_{\mathsf{STL}}$). Finiteness of Std makes it possible to remove the quantifiers by enumeration, and opens the way to work with the STL fragment of HyperSTL, after proper adjustments.

Let $\mathsf{Std} = \{w_1, \ldots, w_c\}$ be an arbitrary standard set with c unique standard traces. We will demonstrate the quantifier elimination by substituting by the placeholder $V(\pi_1, \pi_2)$ the subformula (starting with $\exists \pi_1' \ldots$) of formula (3) behind the second quantification. We can switch the order of the $\forall$-quantifiers without changing the semantics of the formula, so we are working with $\forall \pi_2. \forall \pi_1. V(\pi_1, \pi_2)$. Then, by replacing the second quantifier with the infinite conjunction [23], we get

$$\forall \pi_2. \bigwedge_{w \in \mathsf{S_{STL}}} V(w, \pi_2).$$

The latter can be split into a finite and an infinite conjunction

$$\forall \pi_2. \bigwedge_{w \in \mathsf{Std}} V(w, \pi_2) \wedge \bigwedge_{w \in \mathsf{S_{STL}} \setminus \mathsf{Std}} V(w, \pi_2). \tag{4}$$

Let $W(\pi_1, \pi_2, \pi_1')$ be the placeholder, such that $V(\pi_1, \pi_2) = \exists \pi_1'.\ \mathsf{Std}_{\pi_1} > 0 \rightarrow W(\pi_1, \pi_2, \pi_1')$. Unfolding V in the right (infinite) conjunction in formula (4) reveals

$$\bigwedge_{w \in \mathsf{S_{STL}} \setminus \mathsf{Std}} \exists \pi_1'.\ \mathsf{Std}_w > 0 \rightarrow W(w, \pi_2, \pi_1').$$

It follows directly from the definition of Std_π that for all $w \notin \mathsf{Std}$, Std_w is non-positive. Hence that fragment of the formula is trivially fulfilled, and formula (4) is equivalent to

$$\forall \pi_2. \bigwedge_{w \in \mathsf{Std}} V(w, \pi_2).$$

Combined with similar reasoning for the $\exists$-operator and disjunctions we can altogether rewrite formula (3) into

$$\bigwedge_{w \in \mathsf{Std}} \bigvee_{w' \in \mathsf{Std}} \Big(\mathsf{G}(|\mathsf{i}_w - \mathsf{i}_{w'}| \leq 0) \wedge \big((d_{\mathsf{Out}}(\mathsf{o}_{w'}, \mathsf{o}) - \kappa_\mathsf{o} \leq 0)\ \mathsf{W}\ (d_{\mathsf{In}}(\mathsf{i}_{w'}, \mathsf{i}) - \kappa_\mathsf{i} > 0) \big) \Big), \tag{5}$$

where the $\forall$-quantification over π_1 is replaced by the conjunction over standard traces w, the $\exists$-quantification of π_1' by the disjunction over standard traces w', and the remaining $\forall$-quantification of π_2 is eliminated by rewriting into a trace formula and removing the trace indices from i_{π_2} and o_{π_2}.

Self-composition in logic. Formula (5) is not yet an STL formula, because the distance function d_{In} needs to compare the trace input with inputs of constant traces from the set Std. A popular technique to analyse hyperproperties is self-composition of a system [4,15]. We use a syntactic variant of parallel self-composition as follows. For a trace width n, we compose the signals of the trace under investigation $w = (s_1, \ldots, s_n)$ and the signals of each of the, say, c standard traces $\{(s_{11}, \ldots, s_{1n}), \ldots, (s_{c1}, \ldots, s_{cn})\} = \mathsf{Std}$. The composed trace then is of width $n + nc$, it is $w' = (s_1, \ldots, s_n, s_{11}, \ldots, s_{n1}, \ldots, s_{1c}, \ldots, s_{nc})$. For

the restricted case considered here (one-dimensional input and output signals), $w' = (\mathsf{i}, \mathsf{o}, \mathsf{i}_1, \mathsf{o}_1, \ldots, \mathsf{i}_c, \mathsf{o}_c)$ is of trace width $2 + 2c$. The resulting STL formula for monitoring robust cleanness is

$$\bigwedge_{1 \le a \le c} \bigvee_{1 \le b \le c} \Big(\mathsf{G}(|\mathsf{i}_a - \mathsf{i}_b| \le 0) \wedge \big((d_{\mathsf{Out}}(\mathsf{o}_b, \mathsf{o}) - \kappa_{\mathsf{o}} \le 0) \,\mathsf{W}\, (d_{\mathsf{In}}(\mathsf{i}_b, \mathsf{i}) - \kappa_{\mathsf{i}} > 0) \big) \Big). \tag{6}$$

Recall that a discrete time interpretation of such a formula requires all system traces to share the same timing function τ.

Embedding into the mixed-IO model. The STL formula (6) still is bound to inputs and outputs forming pairs synchronized in time. A more realistic scenario is that of inputs and outputs occurring independently of each other. In particular, when testing a real-world CPS, the testing interface can either pass an input to the system under test or receive an output, but not both at the same time. Furthermore, certain tests require to pass a series of inputs before receiving an output at all [6]. The mixed-IO model supports such real-world testing scenarios. Mixed-IO signals are always defined in the discrete time domain. A mixed-IO signal $s \in (\mathsf{In} \cup \mathsf{Out})^\omega$ (or, equivalently, $s : \mathbb{N} \to \mathsf{In} \cup \mathsf{Out}$) is similar to a real-valued discrete-time signal but the value domain $\mathbb{R}$ is replaced by the domain $\mathsf{In} \cup \mathsf{Out}$. A discrete-time mixed-IO trace $w = (s_1, \ldots, s_n) \in ((\mathsf{In} \cup \mathsf{Out})^\omega)^n$ is a tuple of n mixed-IO signals. Accordingly, predicates of the form $f > 0$ must use functions f that produce real values for mixed-IO signals. Formula (6) requires that all traces share the same timing function. For continuous-time signals, we ensure that this condition is met by transforming all traces into traces with a common value frequency (say, 1 Hz) by averaging the values observed in a time unit (of one second). Let $w = (s_1, \ldots, s_n) \in (\mathbb{N} \to \mathsf{In} \cup \mathsf{Out})^n$ be a recorded trace with some timing function $\tau : \mathbb{N} \to \mathbb{R}_{\ge 0}$, that is sampled with at least one value per time unit, i.e., $\tau(i+1) - \tau(i) \le 1$ for all $i \in \mathbb{N}$. This trace is condensed to a new trace $w' = (s'_1, \ldots, s'_n)$ with timing function $\tau'(i) = i$, and $s'_j(t) \coloneqq \mathsf{average}\left(\cup_{\tau(i) \in [t, t+1)} s_j(i)\right)$ for $1 \le j \le n$, i.e., each signal s'_j is piecewise constant: for each unit time interval $[t, t+1)$ the signal value is set to the average of all signal values originally recorded in that unit time interval.

For adjusting formula (6), let $\mathsf{Std} = \{s_1, \ldots, s_c\} \subseteq (\mathsf{In} \cup \mathsf{Out})^\omega$ be a set of standard traces, each in the form of a single mixed-IO signal. Following the syntactic self-composition idea from above, the composition of a trace w under investigation with Std is the trace $w' = (w, s_1, \ldots, s_c) \in ((\mathsf{In} \cup \mathsf{Out})^\omega)^{c+1}$. This needs two subtle adjustments of the formula. First, the distances d_{In} and d_{Out} are replaced by their mixed-IO counterparts $\bar{d}_{\mathsf{In}}$ and $\bar{d}_{\mathsf{Out}}$, and instead of directly accessing inputs and outputs, the current value is projected to the input and output domain, respectively. Second, since the set In and Out is opaque, the expression $|i_a - i_b|$ is not evaluable any more, it is replaced by the distance $\bar{d}$

based on $\bar{d}_{\mathsf{In}}$ and $\bar{d}_{\mathsf{Out}}$. The resulting formula is

$$\bigwedge_{1\le a\le c}\ \bigvee_{1\le b\le c}\Big(\mathsf{G}(\bar{d}(s_a{\downarrow_{\mathsf{i}}}, s_b{\downarrow_{\mathsf{i}}}) \le 0) \wedge \big((\bar{d}_{\mathsf{Out}}(s_b{\downarrow_{\mathsf{o}}}, s{\downarrow_{\mathsf{o}}}) - \kappa_{\mathsf{o}} \le 0)\ \mathsf{W}\ (\bar{d}_{\mathsf{In}}(s_b{\downarrow_{\mathsf{i}}}, s{\downarrow_{\mathsf{i}}}) - \kappa_{\mathsf{i}} > 0)\big)\Big), \tag{7}$$

where $\bar{d}$ is defined for some $\varepsilon > 0$ as

$$\bar{d}(x,y) \coloneqq \begin{cases} 0, & \text{if } x = y \\ \bar{d}_{\mathsf{In}}(x,y) + \varepsilon, & \text{if } x \neq y \wedge x, y \in \mathsf{In} \\ \bar{d}_{\mathsf{Out}}(x,y) + \varepsilon, & \text{if } x \neq y \wedge x, y \in \mathsf{Out} \\ \infty, & \text{otherwise.} \end{cases}$$

In the second and third clause of the above definition we add some positive value ε to the result of $\bar{d}_{\mathsf{In}}$ and $\bar{d}_{\mathsf{Out}}$, because they are pseudometrics, and $\bar{d}_{\mathsf{In}}(\mathsf{i}_1, \mathsf{i}_2)$ could be 0 even if $\mathsf{i}_1 \neq \mathsf{i}_2$. For the correctness of formula (7), however, it is crucial that $\bar{d}(x,y) = 0$ if and only if $x = y$. For a good performance of the falsification algorithm, we will nevertheless want to make use of $\bar{d}_{\mathsf{In}}$ and $\bar{d}_{\mathsf{Out}}$ if $\mathsf{i}_1 \neq \mathsf{i}_2$. We remark that $\bar{d}$ is not a metric, because the triangle inequality requirement now is violated.

The discussion above has assembled all the details to formally back the following theorem, stating that a system satisfies formula (7) if and only if it satisfies the second condition of robust cleanness in Definition 2.

Theorem 1. *Let* $\mathcal{C} = \langle \mathsf{Std}, \bar{d}_{\mathsf{In}}, \bar{d}_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$ *be a contract for some system* $\mathsf{S}_{\mathsf{IO}} \subseteq (\mathsf{In} \cup \mathsf{Out})^\omega$ *with* $\mathsf{Std} = \{\sigma_1, \ldots, \sigma_c\} \subseteq \mathsf{S}_{\mathsf{IO}}$*, and let* ϕ *denote formula (7). Then, for all* $\sigma' \in \mathsf{S}_{\mathsf{IO}}$*, it holds* $(\sigma', \sigma_1, \ldots, \sigma_c), 0 \models \phi$ *if and only if for all* $\sigma \in \mathsf{Std}$ *and* $k \ge 0$ *such that* $\bar{d}_{\mathsf{In}}(\sigma[j]{\downarrow_{\mathsf{i}}}, \sigma'[j]{\downarrow_{\mathsf{i}}}) \le \kappa_{\mathsf{i}}$ *holds for all* $j \le k$*, there exists* $\sigma'' \in \mathsf{Std}$ *such that* $\sigma{\downarrow_{\mathsf{i}}} = \sigma''{\downarrow_{\mathsf{i}}}$ *and* $\bar{d}_{\mathsf{Out}}(\sigma'[k]{\downarrow_{\mathsf{o}}}, \sigma''[k]{\downarrow_{\mathsf{o}}}) \le \kappa_{\mathsf{o}}$.

Example 2. We consider $\mathcal{C} = \langle \mathsf{Std}, \bar{d}_{\mathsf{In}}, \bar{d}_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$ where $\mathsf{Std} = \{w_1, w_2\}$ contains the two standard traces $w_1 = 1_{\mathsf{i}}\, 2_{\mathsf{i}}\, 3_{\mathsf{i}}\, 7_{\mathsf{o}}\, 0_{\mathsf{i}}\, \delta^\omega$ and $w_2 = 0_{\mathsf{i}}\, 1_{\mathsf{i}}\, 2_{\mathsf{i}}\, 3_{\mathsf{i}}\, 6_{\mathsf{o}}\, \delta^\omega$. We here decorate inputs with index i and outputs with index o, i.e., w_1 describes a system receiving the three inputs 1, 2, and 3, then producing the output 7, and finally receiving input 0 before entering quiescence. We take

$$\bar{d}_{\mathsf{Out}}(\mathsf{o}_1, \mathsf{o}_2) = \begin{cases} |\mathsf{o}_1 - \mathsf{o}_2|, & \text{if } \mathsf{o}_1, \mathsf{o}_2 \in \mathsf{Out}\backslash\{\delta\} \\ 0, & \text{if } \mathsf{o}_1 = \mathsf{o}_2 = -_{\mathsf{o}} \text{ or } \mathsf{o}_1 = \mathsf{o}_2 = \delta \\ \infty, & \text{otherwise,} \end{cases}$$

and similarly for $\bar{d}_{\mathsf{In}}$. The contractual value thresholds are assumed to be $\kappa_{\mathsf{i}} = 1$ and $\kappa_{\mathsf{o}} = 6$.

Assume we are observing the trace $w = 0_{\mathsf{i}}\, 1_{\mathsf{i}}\, 2_{\mathsf{i}}\, 6_{\mathsf{o}}\, 0_{\mathsf{i}}\, \delta^\omega$ to be monitored with formula (7). First notice, that for combinations of a and b in (7), where $a \neq b$, the subformula $\mathsf{G}(\bar{d}(s_a{\downarrow_{\mathsf{i}}}, s_b{\downarrow_{\mathsf{i}}}) \le 0)$ is always false, because s_1 and s_2 (i.e., the

combination of w_1 and w_2) have different values at time point 0. Hence, it remains to show that

$$(\bar{d}_{\mathsf{Out}}(w_1{\downarrow_{\mathsf{o}}}, w{\downarrow_{\mathsf{o}}}) - \kappa_{\mathsf{o}} \leq 0)\ \mathsf{W}\ (\bar{d}_{\mathsf{In}}(w_1{\downarrow_{\mathsf{i}}}, w{\downarrow_{\mathsf{i}}}) - \kappa_{\mathsf{i}} > 0)\ \wedge$$

$$(\bar{d}_{\mathsf{Out}}(w_2{\downarrow_{\mathsf{o}}}, w{\downarrow_{\mathsf{o}}}) - \kappa_{\mathsf{o}} \leq 0)\ \mathsf{W}\ (\bar{d}_{\mathsf{In}}(w_2{\downarrow_{\mathsf{i}}}, w{\downarrow_{\mathsf{i}}}) - \kappa_{\mathsf{i}} > 0).$$

For the first part, the input distance between inputs in w and w_1 is always 1 at positions 1 to 3, it is 0 at position 4 (because $-_{\mathsf{i}}$ is compared to $-_{\mathsf{i}}$) and in position 5 and beyond. Thus, $\bar{d}_{\mathsf{In}}(w_1{\downarrow_{\mathsf{i}}}, w{\downarrow_{\mathsf{i}}}) - \kappa_{\mathsf{i}}$ is always at most 0, and the right hand-side of the W operator is always false. Consequently, by definition of W, the left operand of W must always hold, i.e., $\bar{d}_{\mathsf{Out}}(w_1{\downarrow_{\mathsf{o}}}, w{\downarrow_{\mathsf{o}}})$ must always be less or equal to 6. This is the case for w_1 and w: at all positions except for 4, $-_{\mathsf{o}}$ is compared to $-_{\mathsf{o}}$ (or δ to δ), so the difference is 0, and at position 4, the distance of 6 and 7 is 1.

For the second W-formula, w is compared to w_2. These two traces are comparable only to a limited extent: the order of input and output is altered at the last two positions of the signals before quiescence. Hence, the right operand of W is true at position 4, and the formula holds for the remaining trace. For positions 1 to 3, the input distances are 0, because the input values are identical. At these positions, the left operand must hold. The values are input values, so $-_{\mathsf{o}}$ is compared to $-_{\mathsf{o}}$ at each position. This distance is defined to be 0, so it holds that $-6 \leq 0$, and the formula is satisfied. Since both formulas hold, the conjunction of both holds, too, and trace w is qualified as robustly clean. There could however be other system traces not considered in this example, that overall could violate robust cleanness of the system.

Restriction of input space. Robust cleanness puts semantic requirements on fragments of a system's input space, outside of which the system's behaviour remains unspecified. Typically, the fragment of the input space covered is rather small. To falsify the STL formula (7), the falsifier has two challenging tasks. First, it has to find a way to stay in the relevant input space, i.e., select inputs with a distance of at most κ_{i} from the standard behaviour. Only if this is assured it can search for an output large enough to violate the κ_{o} requirement. In this, a large robustness estimate provided by the quantitative semantics of STL cannot serve as an indicator for deciding whether an input is too far off or whether an output stays too close to the standard behaviour.

The general strength of the falsification technique is its proven ability to discover outputs of a black-box system violating a property. That is why the technique is considered suitable for real-world robust cleanness tests. We can improve its efficiency significantly by narrowing upfront the input space the falsifier uses.

In practice, test execution traces will always be finite. In previous real-life doping tests, test execution lengths have been bounded by some constant $B \in \mathbb{N}$ [6], i.e., systems are represented as sets of finite traces $\mathsf{S} \subseteq (\mathsf{In} \cup \mathsf{Out})^B$ (which for formality reasons each can be considered suffixed with δ^ω). In this bounded horizon, we can provide a predicate discriminating between relevant

and irrelevant input sequences. Formally, the restriction to the relevant input space fragment of a system $\mathsf{S} \subseteq (\mathsf{In} \cup \mathsf{Out})^B$ is given by the set $\mathsf{In}_{\mathsf{Std},\kappa_\mathsf{i}} = \{w \in \mathsf{S} \mid \exists w' \in \mathsf{Std}.\ \bigwedge_{k=0}^{B-1}(\bar{d}_{\mathsf{In}}(w(k)\downarrow_\mathsf{i}, w'(k)\downarrow_\mathsf{i}) \leq \kappa_\mathsf{i})\}$. Since Std and B are finite, membership is computable.

There are rare cases in which this optimisation may prevent the falsifier from finding a counterexample. This is only the case if there is an input prefix leading to a violation of the formula for which there is no suffix such that the whole trace satisfies the κ_i constraint. Below is a pathological example in which this could make a difference.

Example 3. Apart from NO_x emissions, NEDC (and WLTC) tests are used to measure fuel consumption. Consider a contract similar to the contracts above, but with fuel rate as the output quantity. Assuming a "normal" fuel rate behaviour during the standard test, there might be a test within a reasonable κ_i distance, where the fuel is wasted insanely. Then, the fuel tank might run empty before the intended end of the test, which therefore could not be finished within the κ_i distance, because speed would be constantly 0 at the end. The actually driven test is not in set $\mathsf{In}_{\mathsf{Std},\kappa_\mathsf{i}}$, but there is a prefix within κ_i distance that violates the robust cleanness property.

4 Diesel Emissions

This section discusses how to tailor the generic probabilistic falsification approach for STL based on Algorithm 1 to the particular case of diesel emissions, and reports on empirical observations when putting the approach into practice.

Robustness. In the case of diesel emissions doping, the only standard behaviour is either the NEDC or the WLTC. Assuming, for example, NEDC, let $\mathcal{C} = \langle\{\mathsf{NEDC} \circ \mathsf{o}\}, \kappa_\mathsf{i}, \kappa_\mathsf{o}, d_{\mathsf{In}}, d_{\mathsf{Out}}\rangle$ be a diesel emissions specific contract, where NEDC is the sequence of 1180 inputs with the kth input defining the speed of the car after k seconds from the beginning of the test. Here, the output o suffixed to NEDC is the (average) amount of emitted NO_x during the NEDC drive. By restricting the input space to $\mathsf{In}_{\{\mathsf{NEDC}\circ\mathsf{o}\},\kappa_\mathsf{i}}$ as explained in Section 3, formula (7) can be simplified to

$$\mathsf{G}(\bar{d}_{\mathsf{Out}}((\mathsf{NEDC} \circ \mathsf{o})\downarrow_\mathsf{o}, s\downarrow_\mathsf{o}) - \kappa_\mathsf{o} \leq 0). \tag{8}$$

This is because the conjunction and disjunction over standard traces becomes obsolete for only a single standard trace. For the same reason, the requirement $\mathsf{G}(\bar{d}(s_a\downarrow_\mathsf{i}, s_b\downarrow_\mathsf{i}) \leq 0)$ becomes obsolete, as the compared traces are always identical. In the W subformula, the right proposition is always false, because of the restricted input space: the proposition collapses to $\bar{d}_{\mathsf{In}}(\mathsf{NEDC}\circ\mathsf{o}\downarrow_\mathsf{i}, s\downarrow_\mathsf{i}) - \kappa_\mathsf{i} > 0)$ and the input domain $\mathsf{In}_{\{\mathsf{NEDC}\circ\mathsf{o}\},\kappa_\mathsf{i}}$ is $\{(s) \in \mathsf{S} \mid \forall k \in [0, 1180].\ \bar{d}_{\mathsf{In}}\big(s(k)\downarrow_\mathsf{i}, (\mathsf{NEDC}\circ\mathsf{o})(k)\downarrow_\mathsf{i}\big) \leq \kappa_\mathsf{i}\}$. And thus, by the definition of W and U, the W subformula is equivalent to formula (8). We implemented Algorithm 1 for the robustness computation according to formula (8).

Emissions Approximation. In practice, running tests like NEDC with real cars is a time consuming and expensive endeavour. Furthermore, tests on chassis dynamometers are usually prohibited to be carried out with rented cars by the rental companies. On the other hand, car emission models for simulation are not available to the public—and models provided by the manufacturer cannot be considered trustworthy. To carry out our experiments, we instead use an approximation technique that estimates the amount of NO_x emissions of a car along a certain trajectory based on data recorded during previous trips with the same car, sampled at a frequency of 1 Hz (one sample per second). Notably, these trips do not need to have much in common with the trajectory to be approximated. A trip is represented as a finite sequence $\mathcal{T} \in (\mathbb{R} \times \mathbb{R} \times \mathbb{R})^*$ of triples, where each such triple (v, a, n) represents the speed, the acceleration and the absolute amount of NO_x emitted at a particular time instant in the sample. Speed and acceleration can be considered as the main parameters influencing the instant emission of NO_x. This is, for instance, reflected in the RDE regulation [16,24] where the decisive quantities to validate the test route and driving behaviour during RDE tests are speed and acceleration.

A recording $\mathcal{D}$ is the union of finitely many trips $\mathcal{T}$. We can turn such a recording into a predictor $\mathcal{P}$ of the NO_x values given pairs of speed and acceleration as follows:

$$\mathcal{P}(v, a) = \mathsf{average}[n \mid (\exists v', a'.\ (|v - v'| \leq 2\ \wedge |a - a'| \leq 2\ \wedge\ (v', a', n) \in \mathcal{D}))].$$

The amount of NO_x assigned to a pair (v, a) here is the average of all NO_x values seen in the recording $\mathcal{D}$ for $v \pm \ell$ and $a \pm \ell$, with $0 \leq \ell \leq 2$. To overcome measurement inaccuracies and to increase the robustness of the approximated emissions, the speed and acceleration may deviate up to 2 km/h, and $2\,\mathrm{m/s^2}$, respectively. This tolerance is adopted from the official NEDC regulation [26], which allows up to 2 km/h of deviations while driving the NEDC.

Experiment setup. To demonstrate the practical applicability of our implementation of Algorithm 1 and our NO_x approximation, we report here on two experiments. For the first experiment, we use recordings from Biewer et al. [8]. They used the app LolaDrives to perform low-cost RDE tests and recorded the data received from a car's diagnosis port. Using the two RDE recordings that appear in their work, the above predictor can be used to estimate the NO_x emission during NEDC to be 86 mg/km. Their car was an Audi A6 Avant Diesel, admitted in June 2020. We rented the successor of this car model, admitted in 2021, and recorded three low-cost RDE trips with the help of LolaDrives. The new version of this car turned out to have a significantly better emission cleaning system: the estimated amount of NO_x emitted during the NEDC is 9 mg/km. In the sequel, we will refer to the first car as *A20* and to the second as *A21*. Car A20 has previously been falsified w.r.t. the RDE specification. Neither A20 nor A21 has been falsified w.r.t. robust cleanness.

Contracts. Before turning to falsification, we need to spell out meaningful contracts. The input domain $\mathsf{In} \subseteq \mathbb{R}^*$ is the set of finite speed trajectories, and

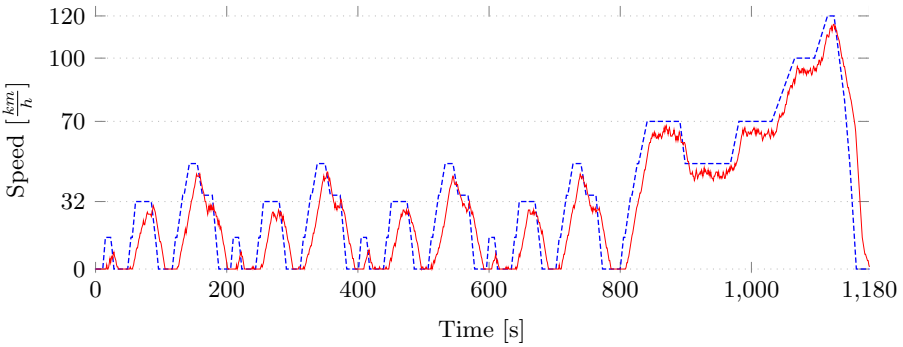


Fig. 3: NEDC speed profile (blue, dashed) and input falsifying $\mathcal{C}$ for $\kappa_o = 88$ mg/km (red) with 182 mg/km of emitted NO_x.

the set $\mathsf{Out} \subseteq \mathbb{R}$ represents the average amount of NO_x emitted during the test. d_{In} must be past-forgetful, hence only the last speed value in each trace must be considered. A natural distance function for inputs is $d_{\mathsf{In}}(v_1, v_2) = |v_1 - v_2|$. Similarly, a measurement for the distance of outputs is $d_{\mathsf{Out}}(o_1, o_2) = |o_1 - o_2|$. Adding the necessary technicalities for the mixed-IO setting, we get $\bar{d}_{\mathsf{In}}$ and $\bar{d}_{\mathsf{Out}}$ as defined in Example 2. For κ_i, it turned out that $\kappa_i = 15$ km/h is a reasonable choice, as it leaves enough flexibility for human-caused driving mistakes and intended deviations [6]. The threshold for NO_x emissions under lab conditions is 80 mg/km. The emission limits for RDE tests depend on the admission date of the car. Cars admitted in 2020 or earlier, must emit 168 mg/km at most, and cars admitted later must adhere to the limit of 120 mg/km. For our experiments, we use $\kappa_o = 88$ mg/km for A20 and $\kappa_o = 40$ mg/km for A21 to have the same tolerances as for RDE tests. Effectively, the upper threshold for A20 is $84 + 88 = 172$ mg/km, and for A21 the limit is $9 + 40 = 49$ mg/km. Notice that for software doping analysis, the output observed for a certain standard behaviour and the constant κ_o define the effective threshold; this threshold is typically different from the threshold defined by the regulation.

Evaluation. We modified Algorithm 1 by adding a timeout condition, i.e., if the algorithm is not able to find a falsifying counterexample within 3,000 iterations, it terminates and returns both the trace for which the smallest robustness has been observed and its corresponding robustness value. Hence, if falsification of robust cleanness for a system is not possible, the algorithm outputs an upper bound on how robust the system satisfies robust cleanness.

For the concrete case of the diesel emissions, the robustness value during the first 1180 inputs (sampled from the restricted input space $\mathsf{In}_{\mathsf{Std},\kappa_i}$) is always κ_o. When the NEDC output o_{NEDC} and the non-standard output o are compared, the robustness value is $\kappa_o - |o_{\mathsf{NEDC}} - o|$ (cf., eq. (8), the quantitative semantics of STL, and definition of $\bar{d}_{\mathsf{Out}}$). Hence, for test cycles with small robustness values,

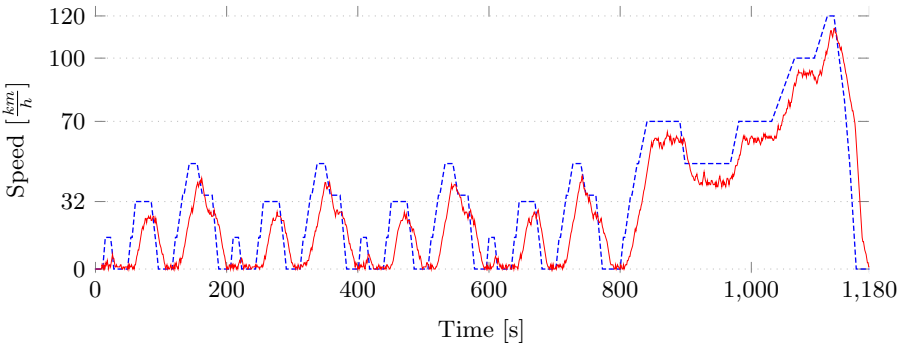


Fig. 4: NEDC speed profile (blue, dashed) and input maximising NO_x emissions to 11 mg/km (red).

we get NO_x emissions o that are either very small or very large compared to o_{NEDC}. We ran the modified Algorithm 1 on A20 and A21 for the contracts defined above. For A20, it found a robustness value of -8, i.e., it was able to falsify robust cleanness relative to the assumed contract and found a test cycle for which NO_x emissions of 182 mg/km are predicted. The test cycle is shown in Fig. 3. For A21, the smallest robustness estimate found—even after 100 independent executions of the algorithm—was 38, i.e., A21 is predicted to satisfy robust cleanness with a very high robustness upper bound. The corresponding test cycle is shown in Fig. 4.

5 Conclusion & Future Work

This paper marks an important milestone in making software doping tests of real-world CPS practically feasible. Regarding test execution effort, real-world testing of CPS is not scalable; the number of tests realistically executable is usually very limited. Probabilistic falsification has its strength in repetitive testing of a system model in a strategic way. We improved this approach by embedding it into a very natural problem solving strategy: Patiently observing the system in-the-wild for the purpose of eventually conducting a small set of doping tests in an even more strategic way. With this paper, we have laid the formal foundations, and we have carved out the aspects that dominate practical applicability. For the latter we focussed on the automotive emissions context. In that context, we are currently spending considerable effort on the acquisition of more high-quality training data. We are building a car data platform (CDP) as a central place for automotive data, which, most importantly, includes the app LolaDrives for convenient recording, uploading and crowd-sourcing of data. With increasing amounts of data collected we hope to be able to roll out predictions that are more and more precise. Finally, we will extend the approach to broader application contexts, to make software doping tests available across the wider CPS domain.

References

1. Abbas, H., Fainekos, G.E., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Probabilistic temporal logic falsification of cyber-physical systems. ACM Trans. Embed. Comput. Syst. **12**(2s), 95:1–95:30 (2013). https://doi.org/10.1145/2465787.2465797
2. Adroit, A.: Software-defined everything (SDE) market perspective (2021-2027): Cisco Systems Inc, Dell Inc, EMC Corp, Extreme Networks, Fujitsu Ltd, Hewlett Packard Enterprise. New Mexico Tribune (2021), https://nmtribune.com/uncategorized/199383/software-defined-everything-sde-market-perspective-2021-2027-cisco-systems-inc-dell-inc-emc-corp-extreme-networks-fujitsu-ltd-hewlett-packard-enterprise/, Online; accessed: 2021-07-13
3. Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990. pp. 390–401. IEEE Computer Society (1990). https://doi.org/10.1109/LICS.1990.113764
4. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Mathematical Structures in Computer Science **21**(6), 1207–1252 (2011). https://doi.org/10.1017/S0960129511000193
5. Biewer, S., D'Argenio, P., Hermanns, H.: Doping tests for cyber-physical systems. In: Parker, D., Wolf, V. (eds.) Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11785, pp. 313–331. Springer (2019). https://doi.org/10.1007/978-3-030-30281-8_18
6. Biewer, S., D'Argenio, P.R., Hermanns, H.: Doping tests for cyber-physical systems. ACM Trans. Model. Comput. Simul. **31**(3), 16:1–16:27 (2021). https://doi.org/10.1145/3449354
7. Biewer, S., Dimitrova, R., Fries, M., Gazda, M., Heinze, T., Hermanns, H., Mousavi, M.R.: Conformance Relations and Hyperproperties for Doping Detection in Time and Space. Logical Methods in Computer Science **18**(1), 14:1–14:39 (2022). https://doi.org/10.46298/lmcs-18(1:14)2022
8. Biewer, S., Finkbeiner, B., Hermanns, H., Köhl, M.A., Schnitzer, Y., Schwenger, M.: RTLola on board: Testing real driving emissions on your phone. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 365–372. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_20
9. Brim, L., Dluhos, P., Safranek, D., Vejpustek, T.: STL*: Extending signal temporal logic with signal-value freezing operator. Inf. Comput. **236**, 52–67 (2014). https://doi.org/10.1016/j.ic.2014.01.012
10. Chib, S., Greenberg, E.: Understanding the metropolis-hastings algorithm. The american statistician **49**(4), 327–335 (1995). https://doi.org/10.1080/00031305.1995.10476177
11. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15
12. D'Argenio, P.R., Barthe, G., Biewer, S., Finkbeiner, B., Hermanns, H.: Is your software on dope? - Formal analysis of surreptitiously "enhanced" programs. In:

Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Proceedings. LNCS, vol. 10201, pp. 83–110. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_4
13. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 264–279. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_19
14. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theor. Comput. Sci. **410**(42), 4262–4291 (2009). https://doi.org/10.1016/j.tcs.2009.06.021
15. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Pasareanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_3
16. Köhl, M.A., Hermanns, H., Biewer, S.: Efficient monitoring of real driving emissions. In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 299–315. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_17
17. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3253, pp. 152–166. Springer (2004). https://doi.org/10.1007/978-3-540-30206-3_12
18. Mathews, M.: Are You Ready for Software-Defined Everything? Wired, https://www.wired.com/insights/2013/05/are-you-ready-for-software-defined-everything/, Online; accessed: 2021-07-13
19. Meinke, K., Sindhu, M.A.: Incremental learning-based testing for reactive systems. In: Gogolla, M., Wolff, B. (eds.) Tests and Proofs - 5th International Conference, TAP@TOOLS 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6706, pp. 134–151. Springer (2011). https://doi.org/10.1007/978-3-642-21768-5_11
20. Nghiem, T., Sankaranarayanan, S., Fainekos, G.E., Ivancic, F., Gupta, A., Pappas, G.J.: Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In: Johansson, K.H., Yi, W. (eds.) Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010. pp. 211–220. ACM (2010). https://doi.org/10.1145/1755952.1755983
21. Nguyen, L.V., Kapinski, J., Jin, X., Deshmukh, J.V., Johnson, T.T.: Hyperproperties of real-valued signals. In: Talpin, J., Derler, P., Schneider, K. (eds.) Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017. pp. 104–113. ACM (2017). https://doi.org/10.1145/3127041.3127058
22. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
23. Rosen, K.H., Krithivasan, K.: Discrete mathematics and its applications: with combinatorics and graph theory. Tata McGraw-Hill Education (2012)

24. The European Parliament and the Council of the European Union: Commission Regulation (EU) 2017/1151 (June 2017), http://data.europa.eu/eli/reg/2017/1151/oj
25. Tutuianu, M., Bonnel, P., Ciuffo, B., Haniu, T., Ichikawa, N., Marotta, A., Pavlovic, J., Steven, H.: Development of the world-wide harmonized light duty test cycle (wltc) and a possible pathway for its introduction in the european legislation. Transportation Research Part D: Transport and Environment **40**(Supplement C), 61 – 75 (2015). https://doi.org/10.1016/j.trd.2015.07.011
26. United Nations: UN Vehicle Regulations - 1958 Agreement, Revision 2, Addendum 100, Regulation No. 101, Revision 3 — E/ECE/324/Rev.2/Add.100/Rev.3 (2013), http://www.unece.org/trans/main/wp29/wp29regs101-120.html
27. Volpato, M., Tretmans, J.: Approximate active learning of nondeterministic input output transition systems. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **72** (2015). https://doi.org/10.14279/tuj.eceasst.72.1008

Estimating Worst-case Resource Usage by Resource-usage-aware Fuzzing*

Liqian Chen[1](✉), Renjie Huang[1,2], Dan Luo[1], Chenghu Ma[1,2], Dengping Wei[1](✉), and Ji Wang[1,2]

[1] College of Computer Science, National University of Defense Technology, Changhsha, China
{lqchen,renjiehuang,luodan,machenghu,dpwei,wj}@nudt.edu.cn
[2] State Key Laboratory of High Performance Computing, Changhsha, China

Abstract. Worst-case resource usage provides a useful guidance in the design, configuration and deployment of software, especially when it runs under a context with limited amount of resources. Static resource-bound analysis can provide sound upper bounds of worst-case resource usage but may provide too conservative, even unbounded, results. In this paper, we present a resource-usage-aware fuzzing approach to estimate worst-case resource usage. The key idea is to guide the fuzzing process using resource-usage amount together with resource-usage relevant coverage. Moreover, we leverage semantic patch to make use of static analysis information (including control-flow, function-call, etc.) to instrument the original program, for the sake of aiding the subsequent fuzzing. We have conducted experiments to estimate worst-case resource usage of various resources in real-world programs, including heap memory, stack depths, sockets, user-defined resources, etc. The preliminary experimental results show the promising ability of our approach in estimating worst-case resource usage in real-world programs, compared with two state-of-the-art fuzzing tools (AFL and MemLock).

Keywords: Fuzzing · Resource Usage · Static Analysis

1 Introduction

Resources refer to any abstractions offered to a process by system calls, apart from the process itself. Typical resources in practice include heap/stack memory, sockets, file descriptors, threads, database connections, gas consumed in Solidity smart contracts, etc. In addition, there exist a variety of user-defined application-dependent resources in applications, such as buffers, memory pools, number of licenses consumed, etc. Worst-case resource usage provides a useful guidance in the design, configuration and deployment of software, especially when the software runs with limited amount of resources, e.g., under the context of modern

* This work is supported by the National Key R&D Program of China (No. 2017YFB1001802), and the NSFC (Nos. 61872445, 62032024).

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 92–101, 2022.
https://doi.org/10.1007/978-3-030-99429-7_5

cyber-physical systems, mobile systems and IoT devices, etc. Unexpected or uncontrolled resource usage may degrade program performance, or even leads to CWE (Common Weakness Enumeration) vulnerabilities (such as *uncontrolled-resource-consumption*, *file-descriptor-exhaustion*, etc.).

Static resource-bound analysis can provide sound upper bounds of worst-case resource usage but may provide too conservative, even unbounded, results. Moreover, most of existing static resource-bound analysis techniques [1, 2, 4, 5, 8, 9, 14] focus on deriving the upper-bound number of accesses to a given control location or simply the bound of iterations of a loop (or recursion). The programs under analysis are often of small-scale, and complex syntactic constructs are usually being abstracted away for simplicity.

In real-world programs, resources are often manipulated via specific APIs which may involve complex structures. Moreover, the usage amount of resources often depends on not only such parameters, but also the running system environment. For example, considering *malloc(n)* in C programs, its actual allocation amount of heap memory depends on the running environment (due to factors such as alignment, the current first available free slot, etc.) and is somehow non-deterministic before execution. The allocation may fail or may allocate memory with size larger than n (e.g., due to alignment). In such cases, dynamic analysis methods are highly desired.

In this paper, we present a resource-usage-aware fuzzing approach to estimate worst-case resource usage. We use resource-usage amount together with resource-usage relevant coverage to guide the fuzzing process, so as to generate inputs triggering large resource-usage amount. More clearly, we use a different definition of branch coverage and additionally add resource-usage amount to guide the fuzzing process. Moreover, we also leverage semantic patch [11] to make use of static analysis information (including control-flow, function-call, etc.) to instrument the original program. Such information is helpful in aiding the subsequent fuzzing during runtime. We have conducted experiments to estimate worst-case resource usage of various resources in real-world programs, including heap memory, stack depths, sockets, user-defined resources, etc. Preliminary experimental results show the promising ability of our approach in estimating worst-case resource usage in real-world programs, compared with two state-of-the-art fuzzing tools (AFL and MemLock).

2 Approach

In this section, we describe the basic process of our approach (shown in Fig. 1).

2.1 Static analysis and instrumentation

For the target program, we first identify all program locations (i.e., program points) of the calls to resource-usage operations in the program. Such resource-usage operations can be APIs provided by systems or libraries, as well as application programmer-defined APIs. From the point of view of increasing/decreasing

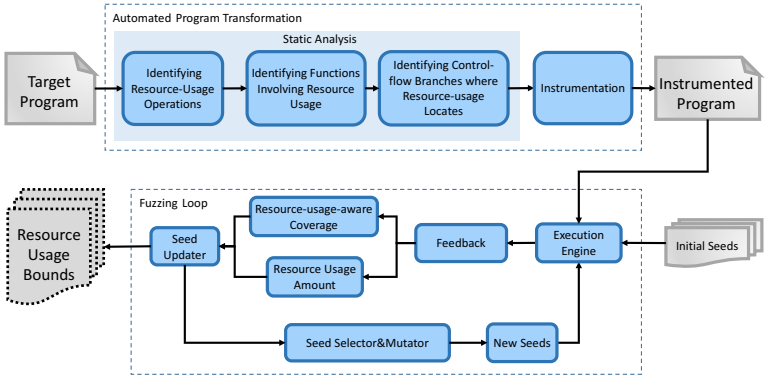


Fig. 1. Workflow of resource-usage-aware fuzzing

resource-usage amount, all operations changing resource-usage can essentially be reduced into allocation (i.e., increasing) and deallocation (i.e., decreasing) operations. To this end, we define two basic modeling functions

- $\_\_RAlloc(int\ n)$, to model allocating n number of resources, and
- $\_\_RDealloc(int\ n)$, to model deallocating n number of resources.

We will instrument invocations of these two basic modeling functions to explicitly model the resource usage for each resource-usage operation in the original program, according to its semantics. For example, to model $pFile = fopen(\ldots)$, we will instrument (afterwards) $\_\_RAlloc(pFile\ !=\ NULL?1:0)$. To model $free(p)$, we will instrument (beforehand) $\_\_RAlloc(malloc\_usable\_size(p))$, wherein the $malloc\_usable\_size(p)$ function (which is a C library function) returns the number of usable bytes in the block pointed to by p. To model the change of call-stack depths, we instrument $\_\_RAlloc(1)$ and $\_\_RDealloc(1)$, respectively at the entry and exit (before *return* statement) of each function. Note that each time of resource-usage fuzzing, we consider only one type of resources. The fuzzing engine will track the invocations of $\_\_RAlloc(int\ n)$ and $\_\_RDealloc(int\ n)$ and capture their parameters to maintain the current amount and the historical peak amount of resource usage at runtime.

On the other hand, many functions and basic blocks in the program are useful for implementing functionality of the program but not relevant to resource usage. Based on this insight, we propose to guide the fuzzing process to cover functions and basic blocks that are relevant to resource usage.

– First, we make use of the call graph of the target program to identify the list of all functions that directly or indirectly invoke resource-usage operations.[3]

[3] Specially, to track stack depth, we first collect a set *FSet* of functions that directly or indirectly call recursive functions. For other functions, we calculate for each function the depth from the *main*() function to that function according to the call graph, and add into *FSet* the top-K percent (e.g., top 30%) functions with large depths.

Then we instrument coverage-label function *__covl()* before the invocation of these functions. We use *__covl()* to identify basic blocks that involve resource usage, which will be further used to define resource-usage-aware coverage.

– Second, for each program block containing invocations of resource-usage modeling functions (i.e., *__RAlloc()*, *__RDealloc()*), label function *__covl()* or exit function *exit()* (as well as similar functions such as *raise()*), we instrument label function *__covl()* before the control-flow branch where this block locates in (e.g., in the *then* branch) and also at the beginning of the block in the other branch (e.g., the *else* branch). We conduct instrumentations of *__covl()* in a bottom-up manner, i.e., from inside to outside blocks.

We leverage program transformation tool Coccinelle [12], to automatically instrument statements invoking resource-usage modeling functions as well as coverage-label function *__covl()* into the original program. Coccinelle is a program matching and transformation engine which allows us to write so-called *semantic patches* [11] for specifying desired code matches and transformations. Particularly, the transformation engine of Coccinelle is defined in terms of control flow, and thus it fits well to instrument coverage-label functions for desired control-flow branches where resource-usage locates.

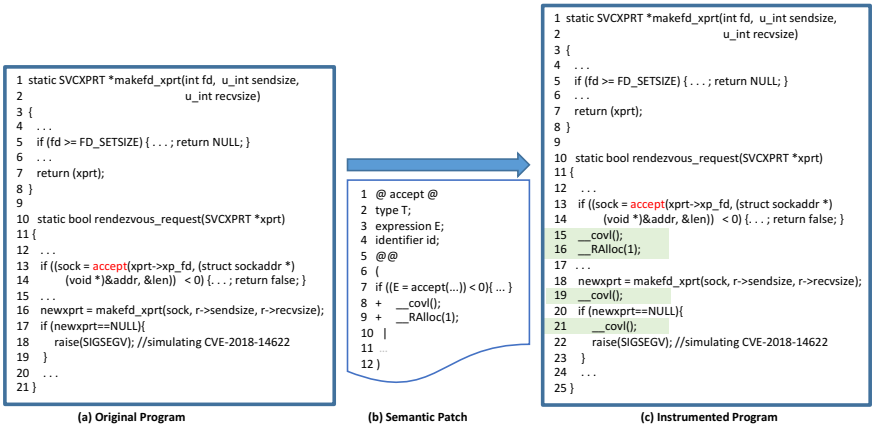


Fig. 2. Example illustration

Example illustration Fig. 2 illustrates the above process via an example (named libtirpc_slice) extracted from an old version of *libtirpc* (that is a Transport-Independent RPC library for Linux) which contains a known CVE vulnerability [4]. The cause of this CVE is that the return value of *makefd_xprt()* was not checked in all instances, which could lead to a crash when the server exhausted the maximum number of available file descriptors. Fig. 2(a) shows the slice extracted from the original code of *libtirpc*. Fig. 2(b) shows part of the semantic patch applied for instrumentation. The instrumented program is shown

[4] https://ubuntu.com/security/CVE-2018-14622

in Fig. 2(c). This program consumes socket connections, e.g., by calling *accept*() as shown on Line 13 in Fig. 2(a). We use semantic patch shown in Fig. 2(b), to instrument resource-usage modeling function *__RAlloc(1)* as well as coverage-label function *__covl*() at the program location when a connection is established successfully. The instrumented code is highlighted in Fig. 2(c).

2.2 Fuzzing loop

Algorithm 1 Resource-usage Aware Fuzzing

Require: an instrumented program P, and a set of initial seeds I_0
Ensure: $(max\_res, BuggyS)$ where max_res is the found largest resource usage amount, and $BuggyS$ is a set of test cases triggering resource-usage bugs

```
1: max_res ← 0
2: BuggyS ← ∅
3: SeedQueue ← I_0
4: while time not expire do
5:     s ← select(SeedQueue)
6:     s' ← mutate(s)
7:     trace ← execute(s')
8:     n_res ← resPeak(trace)
9:     if n_res > max_res then
10:        max_res ← n_res
11:        SeedQueue ← SeedQueue ∪ s'
12:    else
13:        if find_new_path(trace) then
14:            SeedQueue ← SeedQueue ∪ s'
15:        end if
16:    end if
17:    if trigger_crash(trace) then
18:        BuggyS ← BuggyS ∪ s'
19:    end if
20: end while
21: return (max_res, BuggyS)
```

Algorithm. 1 shows the main procedure of our resource-usage aware fuzzing. The algorithm first selects an input s from the seed pool *SeedQueue*, mutates it and generates a mutant s'. Then, the fuzzer runs the mutant input and monitors its execution. If the mutant input consumes more resources or leads to new resource-usage-aware coverage, it will be added to the seed pool as an interesting input. This process is similar to the process of traditional coverage-based grey-box fuzzers (e.g., AFL). The main difference lies in that resource-usage aware fuzzer uses a different definition of branch coverage and adds resource consumption guidance to retain interesting inputs. Now we give the details.

Resource-usage aware coverage Traditional coverage-based grey-box fuzzers use instrumentation to capture basic block transitions, and log edge coverage information during runtime. For example, AFL uses a random number to represent each basic block, and each transition from one basic block to another is marked by the Exclusive-OR (and right shift) result of the two random values. The identifier of each transition is considered as an address and each time of triggering will increment the count of hits at that address. During runtime, AFL records edge coverage information, including whether the edge has been visited, and the count of hits.

In this paper, we concentrate only on resource usage in a program, while many basic blocks in the program are useful for implementing functionality of the program but not relevant to resource usage. Based on this insight, we log only transitions between those basic blocks that contain resource-usage modeling functions (i.e., *__RAlloc*(), *__RDealloc*()), coverage-label function *__covl*() and *exit*() function. E.g., consider an execution trace $B_1^r, B_2, \ldots, B_{n-1}, B_n^r$ wherein only B_1^r, B_n^r contain aforementioned resource-usage relevant functions. We will log it as a transition from B_1^r to B_n^r, and increase the count of hits of this transition. Resource-usage-aware edge coverage is more delicate and sensitive than traditional edge coverage in identifying different resource usage.

Resource-usage amount guidance When resource-usage aware fuzzer runs an input on the instrumented program, it collects not only the resource-usage aware coverage information, but also resource-usage amount. The fuzzing engine maintains two variables, *resc_cur* and *resc_peak*, to track respectively the current amount and the historical peak amount of resource usage. It captures the parameters of *__RAlloc*(n) and *__RDealloc*(n), and updates the current amount as well as the historical peak amount of resource usage.

Overall guidance mechanism As shown in Algorithm. 1, after execution over an input s', we collect the peak resource usage amount of the running trace through *resPeak*(*trace*) (Line 8). If this input leads to more resource usage, it is added into the seed pool for further mutation (Lines 9-11). Besides, if it leads to new resource-usage aware coverage, it is also added into the seed pool for further mutation (Lines 13-14). In addition, if the input triggers a crash, it is added into *BuggyS* which collects the set of test cases triggering resource-usage bugs.

3 Experiments

We have implemented our approach in a prototype fuzzer named ResFuz [5], based on MemLock [16] which is built on top of AFL [17]. We employ Coccinelle [12] to conduct program instrumentation.

We conduct preliminary experiments on several open-source software, including jasper, openjpeg and yara, which are also part of the benchmark used in [16], as well as the small example libtirpc_slice explained in Fig. 2. More specifically, jasper and openjpeg contain many heap resource operations, while yara contains recursive functions. Moreover, jasper and openjpeg contain many user-defined application-specific resource-usage operations. E.g., jasper uses operations like *jas_malloc*(), *jas_free*() to manage a heap memory pool with a user-configurable size. Similarly, openjpeg uses operations like *opj_malloc*(), *opj_free*() to manage a specific type of heap memory. The small program libtirpc_slice contains socket operations, as explained in Sect. 2.1. We compare ResFuz against other two state-of-the-art fuzzers, namely AFL and MemLock [16]. All our experiments

[5] The artifact is available at https://doi.org/10.5281/zenodo.5894821.

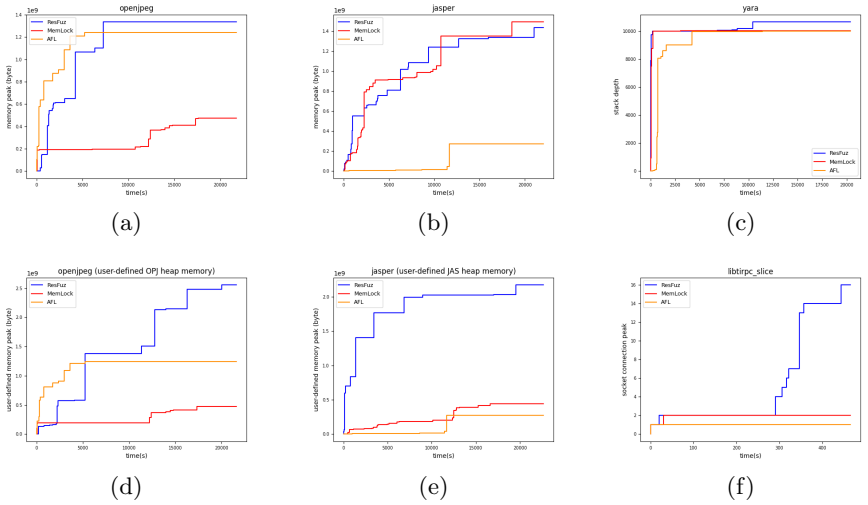


Fig. 3. The growth trend of resource usage

have been performed on machines with an Intel (R) Core (TM) i9-10940X CPU (3.30GHz) and 32GB of RAM under 64-bit Ubuntu LTS 20.04. We run each fuzzer for 6 hours (except 10 minutes for libtirpc_slice) each time, perform each experiment for 3 times, and use their average statistical performance as result.

Fig. 3 depicts the growth trend of the found resource peaks over time through the plots. The vertical axis shows the amount of the peak resource consumed (heaps for jasper and openjpeg, stack depths for yara, sockets for libtirpc_slice). Fig. 3 shows that ResFuz outperforms the two baseline fuzzers in finding large resource consumption for almost all the cases (except for japser shown in Fig. 3(b), for which MemLock performs a little bit better than ResFuz). In particular, as shown in Figs. 3(d-f), for user-defined resources in openjpeg and jasper as well as sockets in libtirpc_slice, ResFuz provides much better results than the other two tools. This is because the guidance mechanism in ResFuz is based on resource-usage amount and resource-usage aware coverage information, which accelerates the process of adding inputs triggering large resource usage into the seed pool. Note that for these user-defined resources and sockets, MemLock uses the consumption of the general heap to guide the fuzzing process, while ResFuz uses respectively the consumption of the specific OPJ heap (in openjpeg), JAS heap (in japser), sockets (in libtirpc_slice) to guide the fuzzing process.

4 Related Work

Using dynamic analysis or fuzzing to find resource-usage relevant bugs has received much attention in recent years. PREDATOR [3] is an automated black box testing tool for detection and identification of local resource-exhaustion vul-

nerabilities in network servers, which computes resource usage profiles for predicting the utilization of every monitored resource for test inputs. Radmin [7] confines the resource usage of a target program from its benign executions to the learned automata and then uses it to detect resource usage anomalies. Both PREDATOR and Radmin do not use fuzzing. MemFuzz [6] uses memory access (rather than memory consumption) instrumentation as addition to branch coverage to guide evolutionary fuzzing. Recently, researchers have drawn attention to the algorithmic complexity vulnerabilities such as SlowFuzz [13], Singularity [15] and PerfFuzz [10]. The basic idea behind is to use the number of executed instructions as the guidance for fuzzing. However, all these works consider time complexity issues.

The most relevant work to our technique is MemLock [16], which uses memory usage guided fuzzing to generate the excessive memory consumption inputs and trigger uncontrolled memory consumption bugs. MemLock also uses memory consumption information to guide the fuzzing process and considers two kinds of memory resources, i.e., stack memory and heap memory. Compared with MemLock, we consider the usage of general resources, including memory, file descriptors, socket connections, user-defined resources, etc. Moreover, MemLock uses default branch coverage of AFL (which considers transitions of all basic blocks) to guide the fuzzing process, while our approach adopts resource-usage-aware coverage (which considers transitions between basic blocks that are relevant to resource usage). In addition, we employ semantic patch to make use of resource-usage relevant call graph and control-flow graph to conduct instrumentation at source code level, while MemLock uses control-flow graph in the same way as AFL (to define branch coverage) and uses call graph only to determine stack memory usage (by instrumenting at the entry and exit of functions).

5 Conclusion and Future Work

In this paper, we present a resource-usage-aware fuzzing approach to estimate worst-case resource usage. It employs resource-usage amount and resource-usage-aware coverage to guide the fuzzing process, for the sake of generating inputs to triggering massive resource usage. Moreover, we employ semantic patches to make use of resource-usage relevant call graph and control-flow graph information to conduct instrumentation, for the sake of aiding the subsequent fuzzing process. We have conducted experiments to estimate worst-case resource usage of various resources in real-world programs, including heap memory, stack depths, sockets, user-defined resources, etc. Preliminary experimental results show its promising ability to estimate worst-case resource usage in real-world programs, compared with two state-of-the-art fuzzing tools.

For future work, we plan to conduct experiments on more real-world programs and over more kinds of resources. We also plan to conduct evaluation comparison with more state-of-the-art fuzzing tools. Furthermore, we will evaluate our approach in detecting resource-usage bugs and security-critical vulnerabilities in real-world programs.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012)
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Proceedings of the 17th International Static Analysis Symposium (SAS). pp. 117–133. Lecture Notes in Computer Science, Springer (2010)
3. Antunes, J., Neves, N.F., Veríssimo, P.J.: Detection and prediction of resource-exhaustion vulnerabilities. In: Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE). pp. 87–96. IEEE (2008)
4. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **38**(4), 1–50 (2016)
5. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 467–478. ACM (2015)
6. Coppik, N., Schwahn, O., Suri, N.: Memfuzz: Using memory accesses to guide fuzzing. In: Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 48–58. IEEE (2019)
7. Elsabagh, M., Barbará, D., Fleck, D., Stavrou, A.: On early detection of application-level resource exhaustion and starvation. Journal of Systems and Software **137**, 430–447 (2018)
8. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Proceedings of the 12th Asian Symposium on Programming Languages and Systems (APLAS). pp. 275–295. Lecture Notes in Computer Science, Springer (2014)
9. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 127–139. ACM (2009)
10. Lemieux, C., Padhye, R., Sen, K., Song, D.: Perffuzz: Automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). pp. 254–265. ACM (2018)
11. Muller, G., Padioleau, Y., Lawall, J.L., Hansen, R.R.: Semantic patches considered helpful. ACM SIGOPS Oper. Syst. Rev. **40**(3), 90–92 (2006)
12. Padioleau, Y., Lawall, J.L., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in linux device drivers. In: Proceedings of the 2008 EuroSys Conference (EuroSys). pp. 247–260. ACM (2008)
13. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 2155–2168. ACM (2017)
14. Sinn, M., Zuleger, F., Veith, H.: Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In: Proceedings of the 2015 Formal Methods in Computer-Aided Design (FMCAD). pp. 144–151. IEEE (2015)
15. Wei, J., Chen, J., Feng, Y., Ferles, K., Dillig, I.: Singularity: pattern fuzzing for worst case complexity. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE). pp. 213–223. ACM (2018)

16. Wen, C., Wang, H., Li, Y., Qin, S., Liu, Y., Xu, Z., Chen, H., Xie, X., Pu, G., Liu, T.: Memlock: Memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE). pp. 765–777 (2020)
17. Zalewski, M.: American fuzzy lop 2.52b. http://lcamtuf. coredump. cx/afl (2017)

Quantitative Program Sketching using Lifted Static Analysis

Aleksandar S. Dimovski[1] (✉)

Mother Teresa University, st. Mirche Acev nr. 4, 1000 Skopje, North Macedonia
aleksandar.dimovski@unt.edu.mk

Abstract. We present a novel approach for resolving numerical program sketches under Boolean and quantitative objectives. The input is a program sketch, which represents a partial program with missing numerical parameters (holes). The aim is to automatically synthesize values for the parameters, such that the resulting complete program satisfies: a *Boolean (qualitative) specification* given in the form of assertions; and a *quantitative specification* that estimates the number of execution steps to termination and which the synthesizer is expected to optimize.
To address the above quantitative sketching problem, we encode a program sketch as a program family (a.k.a. software product line) and analyze it by the specifically designed lifted analysis algorithms based on abstract interpretation. In particular, we use a combination of forward (numerical) and backward (termination) lifted analysis of program families to find the variants (family members) that satisfy all assertions, and moreover are optimal with respect to the given quantitative objective. Such obtained variants represent "correct & optimal" sketch realizations. We present a prototype implementation of our approach within the FamilySketcher tool for resolving C sketches with numerical types. We have evaluated our approach on a set of benchmarks, and experimental results confirm the effectiveness of our approach.

Keywords: Quantitative program sketching · Software Product Lines · Abstract Interpretation

1 Introduction

A *sketch* [29,30] is a partial program with missing numerical expressions called *holes* to be discovered by the synthesizer. Previous approaches for program sketching [29,30,17] automatically synthesize integer constant values for the holes so that the resulting complete program satisfies *Boolean (qualitative) properties* in the form of assertions. However, the need for considering combined Boolean and quantitative properties is prominent in many applications. Still, quantitative properties have been largely missing from previous approaches for program sketching. In particular, there has been no possibility for measuring the "goodness" of solutions. Boolean properties are used to define minimal requirements for the synthesized complete programs. Still, there are usually many different

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 102–122, 2022.
https://doi.org/10.1007/978-3-030-99429-7_6

complete programs that satisfy the Boolean properties, and some of them may be preferred over the others. Therefore, it is important to define synthesis algorithms, which construct complete programs (solutions) that not only meet the Boolean properties, but are also optimal with respect to a given quantitative objective [2,6]. This is so-called *quantitative sketching problem*.

In this paper, we use *lifted static analysis* based on abstract interpretation [25] for program families (a.k.a. software product lines) [8] to solve this quantitative sketching problem. The key observation is that all possible sketch realizations constitute a program family, where each numerical hole is represented as a numerical feature. A *program family* describes a set of similar programs as *variants* of some common code base [8]. At compile-time, a variant of a program family is derived by assigning concrete values to a set of *features* (configuration options) relevant for it, and only then is this variant compiled or interpreted. Program families (often in C) enriched with compile-time configurability by the C preprocessor `CPP` [8,21] are today widely used in open-source projects and industry [21]. By using the proposed transformation from program sketches to program families, we reduce the quantitative sketching problem to selecting those variants (family members) from the corresponding program family that satisfy all assertions and are optimal with respect to the given quantitative objective. As a *quantitative objective* we consider here the sufficient preconditions inferred by a *quantitative termination analysis* that estimates the efficiency of a program by counting upper-bounds on the number of execution steps to termination. More specifically, we use a combination of forward and backward lifted analysis to solve this problem. The forward numerical lifted analysis infers numerical invariants for all members of a program family, thus finding the "correct" variants that satisfy all assertions. Subsequently, the backward termination lifted analysis is performed on a sub-family of "correct" variants to infer piecewise-defined ranking functions, which provide upper-bounds on the number of execution steps to termination. The variants with minimal ranking function are reported as optimal complete programs that solve the original quantitative sketching problem.

To find the required variants (i.e., the solution to the quantitative sketching problem), we use the specifically designed lifted static analysis algorithms, which efficiently analyze all variants of the program family simultaneously, without generating any of them explicitly [3,24,22,28,19,11,20,16]. Lifted analysis processes the common code base of a program family directly, exploiting the similarities among individual variants to reduce analysis effort. It reports precise analysis results for all variants of the family. In particular, we use an efficient, abstract interpretation-based lifted analysis of program families with numerical features [16], where *sharing* is explicitly possible between equivalent analysis elements corresponding to different variants. This is achieved by using a specialized *decision tree lifted domain* [16] that provides a symbolic and compact representation of the lifted analysis elements. More precisely, the elements of the lifted domain are *decision trees*, in which decision nodes are labelled with linear constraints over features, while leaf nodes belong to an existing single-program analysis domain (e.g., some numerical domain [25] or the termination domain [31,32]). The

decision trees recursively partition the space of all variants (i.e., the space of possible combinations of feature's values), whereas the program properties at the leaves provide analysis information corresponding to each partition (i.e., to those variants that satisfy the constraints along the path to the given leaf node). This way, the forward (numerical) lifted analysis partitions the given family into: "correct", "incorrect", and "I don't know" (inconclusive) sub-families (sets of variants) with respect to the given assertions. The backward (termination) lifted analysis additionally partitions the "correct" sub-family with respect to the estimated number of execution steps to termination. Because of its special structure and possibilities for sharing of equivalent analysis results, the decision tree-based lifted analyses are able to converge to a solution very fast even for program families (sketches) that contain numerical features (holes) with large domains, thus giving rise to astronomical search spaces. This is particularly true for sketches in which holes appear in (linear) expressions that can be exactly represented in the underlying numerical domains used in the decision trees (e.g., polyhedra). In those cases, we can design very efficient lifted analysis with extended (improved) transfer functions for assignments and tests.

We have implemented our approach in a prototype program synthesizer, called FamilySketcher [17]. The numerical abstract domains (e.g., intervals, octagons, polyhedra) from the APRON library [23] are used as parameters of the underlying decision trees. FamilySketcher calls the Z3 SMT solver [26] to solve the optimization problem that represents the given quantitative objective. We illustrate this approach for automatic completion of various numerical C sketches from the `Sketch` project [29,30], SV-COMP (https://sv-comp.sosy-lab.org/), and the SyGuS-Competition (https://sygus.org/) [1]. We compare performances of our approach against the most popular sketching tool `Sketch` [29,30] and `Brute-Force` enumeration approach that checks for correctness and optimality all sketch realizations one by one.

In summary, this work makes the following contributions: (1) We combine forward and backward lifted analyses to resolve numerical program sketches with respect to both Boolean and quantitative specifications; (2) We implement our approach in the FamilySketcher tool, which uses numerical domains from the APRON library as parameters and the Z3 tool for solving the underlying (linear) optimization problem; (3) We evaluate our approach and compare its performances with the `Sketch` tool and `Brute-Force` enumeration approach.

2 Motivating Examples

Let us consider the Loop1a sketch taken from SyGuS-Competition [1]:

```
void main() {
1    int x := ??1, y := 0;
2    while h (x > ??2) {
3      x := x-1;
4      y := y+1; }
5    assert (y > 2); //assert (y < 8); }
```

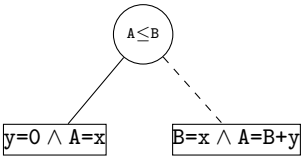


Fig. 1: Lifted numerical invariant at location ⑤ of LOOP1A (solid edges = true, dashed edges = false).

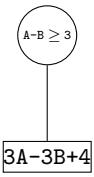


Fig. 2: Lifted ranking function at location ① of LOOP1A.

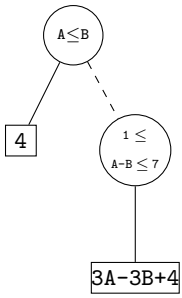


Fig. 3: Lifted ranking function at location ① of LOOP1B.

which contains two numerical holes, denoted by $??_1$ and $??_2$. The synthesizer should replace the holes with constants from $\mathbb{Z}$, such that the synthesized program satisfies the assertion at location ⑤ under all possible inputs. Moreover, we want to select the most efficient correct program, i.e. the one that terminates in the minimum number of execution steps.

We transform the LOOP1A sketch to a program family, which contains two numerical features `A` and `B` with domains $[\texttt{Min}, \texttt{Max}] \subseteq \mathbb{Z}$. [1] Since both holes in the LOOP1A sketch occur in (linear) expressions that can be exactly represented in numerical domains (e.g. intervals), the LOOP1A program family is obtained by replacing the two holes $??_1$ and $??_2$ with the features `A` and `B`. The total number of variants that can be generated from this family is $(\texttt{Max} - \texttt{Min} + 1)^2$, so that each variant corresponds to one possible sketch realization. We perform a *forward numerical lifted analysis* based on decision trees [16] of the LOOP1A program family. The decision tree (lifted numerical invariant) inferred at the location ⑤ is shown in Fig. 1. Notice that the inner nodes of the decision tree in Fig. 1 are labeled with *polyhedral* linear constraints defined over feature variables `A` and `B`, while the leaves are labeled with *polyhedral* linear constraints defined over program and feature variables `x`, `y`, `A` and `B`. The edges of decision trees are labeled with the truth value of the decision on the parent node: we use solid edges for true (i.e., the constraint in the parent node is satisfied) and dashed edges for false (i.e., the negation of the constraint in the parent node is satisfied). Note that linear constraints in decision nodes implicitly take domains of features into account. For example, the decision node $(\texttt{A} \leq \texttt{B})$ is satisfied when $(\texttt{A} \leq \texttt{B}) \wedge (\texttt{Min} \leq \texttt{A} \leq \texttt{Max}) \wedge (\texttt{Min} \leq \texttt{B} \leq \texttt{Max})$. From the invariant inferred at location ⑤ shown in Fig. 1, we can see that the given assertion $(\texttt{y} > 2)$ may be valid in the leaf node that can be reached along the path satisfying the constraint $\neg(\texttt{A} \leq \texttt{B})$, i.e. $(\texttt{A-B} \geq 1)$. In fact, $(\texttt{y} > 2)$ holds when the stronger constraint $(\texttt{A-B} \geq 3)$ is satisfied. Thus, any variant that satisfies the above constraint $(\texttt{A-B} \geq 3)$ represents a "correct" solution to the LOOP1A sketch. To find a "correct & optimal" solution,

[1] Note that `Min` and `Max` represent some minimal and maximal representable integers. E.g., we may take $\texttt{Min} = 0$ and $\texttt{Max} = 31$ for 5-bit sizes of holes.

we perform a *backward termination lifted analysis* based on decision trees [13] of the Loop1a sub-family satisfying (A-B $\geq$ 3). The decision tree representing the lifted ranking function of the above sub-family at initial location ① is shown in Fig. 2. [2] Notice that the leaf nodes represent affine functions defined over feature and program variables. We can see that the ranking function is: 3A-3B+4. We call the Z3 solver [26] to solve the following linear optimization problem: find values for A and B that *minimizes* the value of ranking function 3A-3B+4 over the constraint (A-B $\geq$ 3) $\wedge$ (3A-3B+4 $>$ 0). Minimizing this function gives us values for A and B that are desirable according to the quantitative criterion while satisfying the given assertion. The solution produced by Z3 is: A=3 and B=0 with the minimal objective 13. Therefore, the synthesizer reports this variant, i.e. program where $??_1$=3 and $??_2$=0, as a "correct & optimal" solution.

We consider an alternative sketch of Loop1a, denoted by Loop1b, in which the assertion in location ⑤ is (y $<$ 8). The numerical invariant inferred in location ⑤ is the same as for Loop1a as shown in Fig. 1. However, there are now two solutions to the assertion (y $<$ 8): (A $\leq$ B) when the left leaf node is reached, and (1 $\leq$ A-B $\leq$ 7) when the right leaf node is reached. We perform two backward termination lifted analysis to find optimal solutions for both correct sub-families: (A $\leq$ B) and (1 $\leq$ A-B $\leq$ 7). The lifted ranking function inferred at the initial location is given in Fig. 3. The solutions to the given optimization problem produced by Z3 solver are: A=0, B=0 with the minimal objective 4 for the case (A $\leq$ B); and A=1, B=0 with the minimal objective 7 for the case (1 $\leq$ A-B $\leq$ 7).

Let us consider the Loop2a sketch in Fig. 10. The lifted numerical invariant inferred at location ⑥ is shown in Fig. 4. We can see that the assertion (y $>$ 2) is valid for variants satisfying: (A-B $\geq$ 1) $\wedge$ (3 $\leq$ A $\leq$ Max). The lifted ranking function inferred for this sub-family is shown in Fig. 5. It represents a piecewise-defined ranking function since it depends on the value of the input variable x. To represent graphically piecewise-defined ranking functions in decision trees, we use rounded rectangles to represent second-level decision nodes that are labelled with linear constraints defined over both feature and program variables. Thus, they partition the configuration and memory space, i.e. the possible values of feature and program variables (see Fig. 5). The obtained "correct & optimal" solution is: A=3 and B=0 with the minimal objective 3 when (x $>$ 10) and -3x+36 when (x $\leq$ 10). Similarly, we can resolve the Loop2b sketch, where the assertion (y $<$ 8) is considered. The "correct" variants satisfy: (A-B $\geq$ 1) $\wedge$ (Min $\leq$ A $\leq$ 7), and the "correct & optimal" solution is: A=1 and B=0 with the minimal objective 3 when (x $>$ 10) and -3x+36 when (x $\leq$ 10). Note that, the inferred ranking functions for "correct" sub-families of Loop2a and Loop2b in Figs. 5 and 6 do not depend on feature variables, so any "correct" solution is "optimal" as well.

From the decision trees inferred by performing lifted analyses of our motivating examples, we can see that the decision tree-based representation uses only one or two leaf nodes, although there are many variants in total. This possibility for sharing of analysis equivalent information corresponding to different variants confirms that decision trees are symbolic and compact representation of lifted

[2] Termination analysis is backward, so the final result is reported in the initial location.

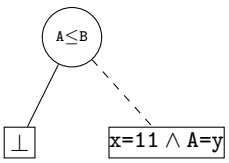


Fig. 4: Lifted numerical invariant before assertion in Loop2a.

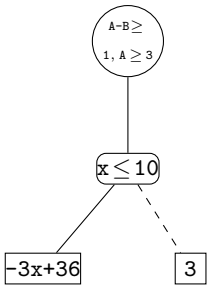


Fig. 5: Lifted ranking function at initial location of Loop2a.

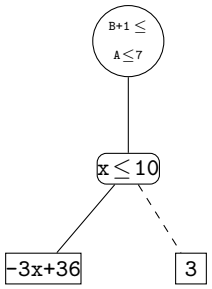


Fig. 6: Lifted ranking function at initial location of Loop2b.

analysis elements. This is the key for obtaining efficient lifted analyses of program families with large configuration spaces, and thus for efficiently solving the quantitative sketching problem.

3 Transforming Sketches to Program Families

We now introduce the IMP language that we use to illustrate our work. We describe two extensions of IMP: $\text{IMP}_{??}$ for writing program sketches, and $\overline{\text{IMP}}$ for writing program families. Finally, we define the transformation of sketches to program families and show its correctness.

IMP. We use a simple imperative language, called IMP [27,25], for writing general-purpose single-programs. Program variables *Var* are statically allocated, and the only data type is the set $\mathbb{Z}$ of mathematical integers. Syntax is:

$$s ::= \texttt{skip} \mid \texttt{x:=}ae \mid s; s \mid \texttt{if}\ (be)\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ (be)\ \texttt{do}\ s \mid \texttt{assert}\ (be),$$
$$ae ::= n \mid [n, n'] \mid \texttt{x} \mid ae \oplus ae, \qquad be ::= ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be$$

where n ranges over integers $\mathbb{Z}$, $[n, n']$ over integer intervals, $\texttt{x}$ over program variables *Var*, $\oplus \in \{+, -, *, /\}$, and $\bowtie \in \{<, \leq, =, \neq\}$. Intervals $[n, n']$ denote a random choice of an integer in the interval. The set of all statements s is denoted by *Stm*; the set of all arithmetic expressions ae is denoted by *AExp*; the set of all boolean expressions be is denoted by *BExp*.

A *program state* $\sigma : \Sigma = Var \to \mathbb{Z}$ is a mapping from program variables to values. The meaning of boolean expressions $[\![be]\!] : \Sigma \to \mathcal{P}(\{\text{true}, \text{false}\})$, arithmetic expressions $[\![ae]\!] : \Sigma \to \mathcal{P}(\mathbb{Z})$, and statements $[\![s]\!] : \Sigma \to \mathcal{P}(\Sigma)$, are defined by induction on their structure [27,25]. For example, the meaning of an arithmetic expression ae is a function from a state to a set of values:

$$[\![n]\!]\sigma = \{n\},\ [\![[n, n']]\!]\sigma = \{n, \ldots, n'\},\ [\![\texttt{x}]\!]\sigma = \{\sigma(\texttt{x})\},$$
$$[\![ae_0 \oplus ae_1]\!]\sigma = \{n_0 \oplus n_1 \mid n_0 \in [\![ae_0]\!]\sigma, n_1 \in [\![ae_1]\!]\sigma\}$$

We write $[\![s]\!]$ for the set of final states that can be derived by executing s from some initial input state [27,25].

IMP$_{??}$. The language for sketches IMP$_{??}$ is obtained by extending IMP with a basic hole construct, denoted by `??`. The numerical hole `??` is a placeholder that the synthesizer must replace with a suitable integer constant.

$$ae ::= \ldots \mid \texttt{??}$$

Each hole occurrence in a program sketch is assumed to be uniquely labelled as $\texttt{??}_i$ and has a bounded integer domain $[n, n']$. We will sometimes write $\texttt{??}_i^{[n,n']}$ to make explicit the domain of a given hole.

Let H be a set of holes in a program sketch. We define a *control function* $\phi : \Phi = H \to \mathbb{Z}$ to describe the value of each hole in the sketch. Thus, ϕ fully describes a candidate solution to the sketch. We write s^{ϕ} to describe a candidate solution to the sketch s fully defined by control function ϕ.

$\overline{\text{IMP}}$. Let $\mathcal{F} = \{A_1, \ldots, A_n\}$ be a finite and totaly ordered set of *numerical features* available in a program family. For each feature $A \in \mathcal{F}$, $\text{dom}(A) \subseteq \mathbb{Z}$ denotes the set of possible values that can be assigned to A. A valid combination of feature's values represents a *configuration* k, which specifies one *variant* of a program family. It is given as a *valuation function* $k : \mathcal{F} \to \mathbb{Z}$, which is a mapping that assigns a value from $\text{dom}(A)$ to each feature $A \in \mathcal{F}$. We assume that only a subset $\mathcal{K}$ of all possible configurations are *valid*. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathcal{K}$ can also be represented by a propositional formula: $(A_1 = k(A_1)) \wedge \ldots \wedge (A_n = k(A_n))$. The set of configurations $\mathcal{K}$ can be also represented as a formula: $\vee_{k \in \mathcal{K}} k$. We define *feature expressions*, denoted *FeatExp*$(\mathcal{F})$, as the set of propositional logic formulas over constraints of $\mathcal{F}$ generated by:

$$\theta ::= \text{true} \mid e_{\mathcal{F}} \bowtie e_{\mathcal{F}} \mid \neg\theta \mid \theta_1 \wedge \theta_2 \mid \theta_1 \vee \theta_2, \qquad e_{\mathcal{F}} ::= n \in \mathbb{Z} \mid A \in \mathcal{F} \mid e_{\mathcal{F}} \oplus e_{\mathcal{F}}$$

When a configuration $k \in \mathcal{K}$ satisfies a feature expression $\theta \in \textit{FeatExp}(\mathcal{F})$, we write $k \models \theta$, where $\models$ is the standard satisfaction relation. We write $[\![\theta]\!]$ to denote the set of configurations from $\mathcal{K}$ that satisfy θ, that is, $k \in [\![\theta]\!]$ iff $k \models \theta$.

The language for program families $\overline{\text{IMP}}$ is obtained by extending IMP with a new compile-time conditional statement for encoding multiple variants and a new arithmetic expression that represents a feature variable. The new statement "`#if` (θ) s `#endif`" contains a feature expression $\theta \in \textit{FeatExp}(\mathcal{F})$ as a *presence condition*, such that only if θ is satisfied by a configuration $k \in \mathcal{K}$ the statement s will be included in the variant corresponding to k. The syntax is:

$$s ::= \ldots \mid \texttt{\#if}\ (\theta)\ s\ \texttt{\#endif}, \qquad ae ::= \ldots \mid \texttt{A} \in \mathcal{F}$$

Any other preprocessor conditional constructs can be desugared and represented only by `#if` construct. For example, `#if` (θ) s_0 `#elif` (θ') s_1 `#endif` is translated into the following: `#if` (θ) s_0 `#endif ; #if` $(\neg\theta \wedge \theta')$ s_1 `#endif`. Note that feature variables $\texttt{A} \in \mathcal{F}$ can occur in arbitrary expressions in $\overline{\text{IMP}}$, not only in presence conditions of `#if`-s as in traditional program families [21,24].

The semantics of $\overline{\text{IMP}}$ has two stages: first, given a configuration $k \in \mathcal{K}$ compute an IMP single-program without `#if`-s and $\texttt{A} \in \mathcal{F}$; second, the obtained

program is evaluated using the standard IMP semantics [24]. The first stage is specified by the projection function π_k, which recursively pre-processes all sub-statements and sub-expressions of statements. Hence, $\pi_k(\texttt{skip}) = \texttt{skip}$, $\pi_k(\texttt{x:=}ae) = \texttt{x:=}\pi_k(ae)$, $\pi_k(s;s') = \pi_k(s);\pi_k(s')$, $\pi_k(ae \oplus ae') = \pi_k(ae) \oplus \pi_k(ae')$, and $\pi_k(ae \bowtie ae') = \pi_k(ae) \bowtie \pi_k(ae')$. For "`#if` (θ) s `#endif`", statement s is included in the variant if $k \models \theta$, otherwise, if $k \not\models \theta$ statement s is removed: [3]

$$\pi_k(\texttt{\#if } (\theta)\ s\ \texttt{\#endif}) = \begin{cases} \pi_k(s) & \text{if } k \models \theta \\ \texttt{skip} & \text{if } k \not\models \theta \end{cases}.$$

For a feature $A \in \mathcal{F}$, the projection function π_k replaces A with the value $k(A) \in \mathbb{Z}$, that is $\pi_k(A) = k(A)$.

Transformation. We want to transform an input sketch $\hat{s}$ with a set of m holes $\texttt{??}_1^{[n_1,n_1']}, \ldots, \texttt{??}_m^{[n_m,n_m']}$ into an output program family $\overline{s}$ with a set of features $A_1, \ldots, A_m$ with domains $[n_1, n_1'], \ldots, [n_m, n_m']$, respectively. The set of configurations $\mathcal{K}$ in $\overline{s}$ includes all possible combinations of feature's values.

If a hole occurs in a (linear) expression that can be exactly represented in the underlying numerical abstract domain $\mathbb{D}$, then we can handle the hole in a more efficient symbolic way by an extended lifted analysis. Given the polyhedra domain P, we say that a hole `??` can be *exactly represented* in P, if it occurs in an expression of the form: $\alpha_1 x_1 + \ldots \alpha_i \texttt{??} + \ldots \alpha_n x_n + \beta$, where $\alpha_1, \ldots, \alpha_n, \beta \in \mathbb{Z}$ and $x_1, \ldots x_n$ are program variables or other hole occurrences. Similarly, we define that a hole can be *exactly represented* in the interval I and the octagon O domains, if it occurs in expressions of the form: $\pm\texttt{??} + \beta$ and $\pm\texttt{x} \pm \texttt{??} + \beta$, (where $\beta \in \mathbb{Z}$, `x` is a program variable or other hole occurrence), respectively.

We now define rewrite rules for eliminating holes `??` from a program sketch $\hat{s}$. Let $s[\texttt{??}^{[n,n']}]$ be a basic (non-compound) statement in which the hole $\texttt{??}^{[n,n']}$ occurs as a sub-expression. When the hole $\texttt{??}^{[n,n']}$ occurs in an expression that can be represented exactly in the numerical domain $\mathbb{D}$, we eliminate `??` using the *symbolic rewrite rule*:

$$s[\texttt{??}^{[n,n']}] \rightsquigarrow s[\texttt{A}] \qquad \text{(SR)}$$

Otherwise, if the hole $\texttt{??}^{[n,n']}$ occurs in an expression that cannot be represented exactly in the numerical domain $\mathbb{D}$, then we use the *explicit rewrite rule*:

$$s[\texttt{??}^{[n,n']}] \rightsquigarrow \texttt{\#if (A=}n\texttt{)}\ s[n]\ \texttt{\#elif}\ \ldots\ \texttt{\#elif (A=}n'\texttt{-1)}\ s[n'\texttt{-1}]\ \texttt{\#else}\ s[n']\ \ldots\ \texttt{\#endif} \qquad \text{(ER)}$$

The set of features $\mathcal{F}$ is also updated with the fresh feature `A`. We write `Rewrite`$(\hat{s})$ to be the resulting program family obtained by repeatedly applying rules (SR) and (ER) on a program sketch $\hat{s}$ to saturation.

Example 1. Reconsider the Loop1a and Loop2a sketches from Section 2. All holes `??` can be represented exactly in the interval domain, so we use the symbolic (SR) rule to obtain the program family. Consider the sketch: `int x; while (x` $\geq$ `0) x := ??*x+10`. The hole `??` cannot be represented exactly in any numerical domain $\mathbb{D}$. Thus, we use the explicit (ER) rule to obtain the program family. □

[3] Since any $k \in \mathcal{K}$ is a valuation function, we have that either $k \models \theta$ holds or $k \not\models \theta$ (which is equivalent to $k \models \neg\theta$) holds, for any $\theta \in \mathit{FeatExp}(\mathcal{F})$.

The following result establishes the correctness of our transformation. It can be proved by structural induction on statements and expressions.

Theorem 1. *Let $\hat{s}$ be a sketch with holes* `??`$_1, \ldots,$ `??`$_n$, *ϕ be a control function, and $\hat{s}^\phi$ be a candidate solution of $\hat{s}$. Let $\overline{s} =$* `Rewrite`$(\hat{s})$ *be a program family, in which features $A_1, \ldots, A_n$ correspond to holes* `??`$_1, \ldots,$ `??`$_n$. *We define a configuration $k \in \mathcal{K}$, s.t. $k(A_i) = \phi($*`??`$_i)$ *for $1 \leq i \leq n$. Then, we have: $[\![\hat{s}^\phi]\!] = [\![\pi_k(\overline{s})]\!]$.*

4 Decision Tree-based Lifted Analyses

In the context of program families, *lifting* means taking a static analysis that works on IMP single-programs, and transforming it into an analysis that works on $\overline{\text{IMP}}$ program families, without preprocessing them. In this work, we will use *lifted versions* of the (forward) numerical analysis [25] and the (backward) termination analysis [31] from the abstract interpretation framework [9]. They will be used to infer numerical invariants and piecewise-defined ranking functions in all program locations. We work with lifted analyses based on the lifted domain of decision trees [16], in which the leaf nodes belong to an existing single-program domain (e.g., a numerical or termination domain) and decision nodes are linear constraints over feature variables. This way, we encapsulate the set of configurations $\mathcal{K}$ into decision nodes where each top-down path represents a subset of configurations from $\mathcal{K}$, and we store in each leaf node the analysis property generated from the variants corresponding to the given configurations.

4.1 Abstract domain for decision nodes

The domain of decision nodes $\mathbb{C}_{\mathbb{D}_V}$ is the finite set of linear constraints defined over a set of variables $V = \{X_1, \ldots, X_k\}$. $\mathbb{C}_{\mathbb{D}}$ is constructed using the numerical domain $\mathbb{D}$ (see Section 4.2) by mapping a conjunction of constraints from $\mathbb{D}$ to a finite set of constraints in $\mathcal{P}(\mathbb{C}_{\mathbb{D}})$. We assume the set of variables $V = \{X_1, \ldots, X_k\}$ to be a finite and totally ordered set, such that the ordering is $X_1 > \ldots > X_k$. We impose a total order $<_{\mathbb{C}_{\mathbb{D}}}$ on $\mathbb{C}_{\mathbb{D}}$ to be the lexicographic order on the coefficients $\alpha_1, \ldots, \alpha_k$ and constant α_{k+1} of the linear constraints:

$$(\alpha_1 \cdot X_1 + \ldots + \alpha_k \cdot X_k + \alpha_{k+1} \geq 0) \;<_{\mathbb{C}_{\mathbb{D}}}\; (\alpha'_1 \cdot X_1 + \ldots + \alpha'_k \cdot X_k + \alpha'_{k+1} \geq 0)$$
$$\iff \exists j > 0. \forall i < j. (\alpha_i = \alpha'_i) \wedge (\alpha_j < \alpha'_j)$$

The negation of linear constraints is formed as: $\neg(\alpha_1 X_1 + \ldots \alpha_k X_k + \beta \geq 0) = -\alpha_1 X_1 - \ldots - \alpha_k X_k - \beta - 1 \geq 0$. For example, the negation of $X - 3 \geq 0$ is $-X + 2 \geq 0$. To ensure canonical representation of decision trees, a linear constraint c and its negation $\neg c$ cannot both appear as decision nodes. Thus, we only keep the largest constraint with respect to $<_{C_{\mathbb{D}}}$ between c and $\neg c$.

4.2 Abstract domain for leaf nodes

We assume the existence of a single-program abstract domain $\mathbb{A}$ defined over a set of variables $V = \{X_1, \ldots, X_k\}$. The domain $\mathbb{A}$ is equipped with sound operators for concretization $\gamma_{\mathbb{A}}$, ordering $\sqsubseteq_{\mathbb{A}}$, join $\sqcup_{\mathbb{A}}$, meet $\sqcap_{\mathbb{A}}$, bottom $\bot_{\mathbb{A}}$, top $\top_{\mathbb{A}}$,

widening $\nabla_{\mathbb{A}}$, and narrowing $\triangle_{\mathbb{A}}$, as well as sound transfer functions for tests (boolean expressions) FILTER$_{\mathbb{A}}$, forward assignments F-ASSIGN$_{\mathbb{A}}$, and backward assignments B-ASSIGN$_{\mathbb{D}}$. More specifically, FILTER$_{\mathbb{A}}(a : \mathbb{A}, be : BExp)$ returns an abstract element from $\mathbb{A}$ obtained by restricting a to satisfy the test be; F-ASSIGN$_{\mathbb{A}}(a : \mathbb{A}, \texttt{x:=}e : Stm)$ returns an updated version of a by abstractly evaluating `x:=`e in it; whereas B-ASSIGN$_{\mathbb{A}}(b : \mathbb{A}, \texttt{x:=}ae : Stm)$ returns an abstract element from $\mathbb{A}$ that can lead to the abstract element b to hold after evaluating `x:=`ae. Note that a in F-ASSIGN$_{\mathbb{A}}$ is an invariant in the initial location of `x:=`ae that needs to be propagated forward, while b in B-ASSIGN$_{\mathbb{A}}$ is an invariant in the final location of `x:=`ae that needs to be propagated backwards. We will sometimes write $\mathbb{A}_V$ to explicitly denote the set of variables V over which $\mathbb{A}$ is defined. In this work, we will use domains $\mathbb{A}_{Var}$, $\mathbb{A}_{\mathcal{F}}$, and $\mathbb{A}_{Var \cup \mathcal{F}}$.

For the forward numerical analysis, we will instantiate $\mathbb{A}$ with some of the known numerical domains $\langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$, such as Intervals $\langle I, \sqsubseteq_I \rangle$ [9,25], Octagons $\langle O, \sqsubseteq_O \rangle$ [25], and Polyhedra $\langle P, \sqsubseteq_P \rangle$ [25]. The elements of I are intervals of the form: $\pm X \geq \beta$, where $X \in V, \beta \in \mathbb{Z}$; the elements of O are conjunctions of octagonal constraints of the form $\pm X_1 \pm X_2 \geq \beta$, where $X_1, X_2 \in V, \beta \in \mathbb{Z}$; while the elements of P are conjunctions of polyhedral constraints of the form $\alpha_1 X_1 + \ldots + \alpha_k X_k + \beta \geq 0$, where $X_1, \ldots X_k \in V, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}$.

For the backward termination analysis, we will instantiate $\mathbb{A}$ with the termination decision tree domain $\mathbb{T}^T(\mathbb{C}_{\mathbb{D}_{Var \cup \mathcal{F}}}, \mathbb{F}_A)$, also written $\mathbb{T}^T$ for short, introduced by Urban and Miné [31,32], where $\mathbb{C}_{\mathbb{D}_{Var \cup \mathcal{F}}}$ is the domain for decision nodes and $\mathbb{F}_A$ is the domain of *affine functions* for leaf nodes. The elements of $\mathbb{F}_A$ are: $\{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\} \cup \{f : \mathbb{Z}^{|Var \cup \mathcal{F}|} \to \mathbb{N} \mid f(x_1, \ldots, x_n) = m_1 x_1 + \ldots + m_n x_n + q\}$, where $f \in \mathbb{F}_A$ is a natural-valued function of program and feature variables representing an upper bound on the number of steps to termination; the element $\bot_{\mathbb{F}}$ represents potential non-termination; and $\top_{\mathbb{F}}$ represents the lack of information to conclude. The leaf nodes belonging to $\mathbb{F}_A \backslash \{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$ and $\{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$ represent *defined* and *undefined* leaf nodes, respectively. A *termination decision tree* $t' \in \mathbb{T}^T$ is: either a leaf node $\ll f \gg$ with $f \in \mathbb{F}_A$, or $[\![c' : tl', tr']\!]$, where $c' \in \mathbb{C}_{\mathbb{D}_{Var \cup \mathcal{F}}}$ (denoted by $t'.c$) is the smallest constraint with respect to $<_{\mathbb{D}}$ appearing in the tree t', tl' (denoted by $t'.l$) is the left subtree of t' representing its *true branch*, and tr' (denoted by $t'.r$) is the right subtree of t' representing its *false branch*. The path along a decision tree establishes a set of program states and a set of configurations (those that satisfy the encountered constraints), and leaf nodes represent partially-defined ranking functions over the given program states and configurations. The transfer function B-ASSIGN$_{\mathbb{T}^T}(t', \texttt{x:=}ae)$ substitutes the arithmetic expression ae to the variable `x` in linear constraints occurring within decision nodes of t' and in functions occurring in leaf nodes of t', whereas the transfer function FILTER$_{\mathbb{T}^T}(t', be)$ generates a set of linear constraints J from test be and restricts t' such that all paths satisfy the constraints from J. Finally, both transfer functions increment the constant q of defined functions $f \in \mathbb{F}_A \backslash \{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$ in all leaf nodes of t'.

We refer to [25,31] for a precise definition of all operations and transfer functions of intervals, octagons, polyhedra, and termination decision tree domain.

Algorithm 1: $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}e, C)$ **when vars**$(ae) \subseteq$ ***Var***

1 **if** isLeaf(t) **then return** $\ll \texttt{ASSIGN}_{\mathbb{A}}(t, \texttt{x:=}e) \gg$;
2 **else return** $[\![t.c : \texttt{ASSIGN}_{\mathbb{T}}(t.l, \texttt{x:=}e, C \cup \{t.c\}), \texttt{ASSIGN}_{\mathbb{T}}(t.r, \texttt{x:=}e, C \cup \{\neg t.c\})]\!]$;

4.3 Decision tree lifted domains

We now define the decision tree lifted domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}, \mathbb{A}_{Var \cup \mathcal{F}})$, written $\mathbb{T}$ for short, for representing lifted analysis properties [16]. A *decision tree* $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ is either a leaf node $\ll a \gg$ with $a \in \mathbb{A}$, or $[\![c : tl, tr]\!]$, where decision node $c \in \mathbb{C}_{\mathbb{D}}$ (denoted by $t.c$) is *the smallest constraint* with respect to $<_{\mathbb{C}_{\mathbb{D}}}$ appearing in the tree t, tl (denoted by $t.l$) is the left subtree of t representing its *true branch*, and tr (denoted by $t.r$) is the right subtree of t representing its *false branch*. The path along a decision tree establishes the set of configurations (those that satisfy the encountered constraints), and the leaf nodes represent their analysis properties.

Operations. The *concretization function* $\gamma_{\mathbb{T}}$ of a decision tree $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ returns $\gamma_{\mathbb{A}}(a)$ for $k \in \mathcal{K}$ that satisfies the set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ of constraints accumulated along the top-down path to the leaf node $a \in \mathbb{A}$.

The binary operations rely on the algorithm for *tree unification* [16,31], which finds a common labelling of decision nodes of two trees t_1 and t_2. Note that the tree unification does not lose any information. All binary operations, including ordering $\sqsubseteq_{\mathbb{T}}$, join $\sqcup_{\mathbb{T}}$, meet $\sqcap_{\mathbb{T}}$, widening $\nabla_{\mathbb{T}}$, and narrowing $\triangle_{\mathbb{T}}$, are performed leaf-wise on the unified decision trees. For example, the ordering $t_1 \sqsubseteq_{\mathbb{T}} t_2$ of two unified decision trees t_1 and t_2 is defined recursively as:

$$\ll a_1 \gg \sqsubseteq_{\mathbb{T}} \ll a_2 \gg = a_1 \sqsubseteq_{\mathbb{A}} a_2, \;\; [\![c{:}tl_1, tr_1]\!] \sqsubseteq_{\mathbb{T}} [\![c{:}tl_2, tr_2]\!] = (tl_1 \sqsubseteq_{\mathbb{T}} tl_2) \wedge (tr_1 \sqsubseteq_{\mathbb{T}} tr_2)$$

The top is: $\top_{\mathbb{T}} = \ll \top_{\mathbb{A}} \gg$, while the bottom is: $\bot_{\mathbb{T}} = \ll \bot_{\mathbb{A}} \gg$.

Transfer functions. We define lifted transfer functions for tests, (forward and backward) assignments ($\text{ASSIGN}_{\mathbb{T}}$), and `#if`-s [16]. We consider several types of tests *be* and assignments `x:=`*ae*: when *be* and *ae* contain only program variables; and when *be* and *ae* contain both feature and program variables.

Transfer function $\text{ASSIGN}_{\mathbb{T}}$ [4] for handling an assignment `x:=`*ae* in the input tree t, when the set of variables in *ae* is $vars(ae) \subseteq Var$, is implemented by applying $\text{ASSIGN}_{\mathbb{A}}$ leaf-wise, as shown in Algorithm 1. Similarly, transfer function $\text{FILTER}_{\mathbb{T}}$ for handling tests $be \in BExp$, when $vars(be) \subseteq Var$, is implemented by applying $\text{FILTER}_{\mathbb{A}}$ leaf-wise.

Transfer function $\text{ASSIGN}_{\mathbb{T}}$ for `x:=`*ae*, when $\text{vars}(ae) \subseteq Var \cup \mathcal{F}$, is given in Algorithm 2. It accumulates into the set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ (initialized to $\mathcal{K}$) constraints encountered along the paths of the decision tree t (Line 2), up to the leaf nodes in which assignment is performed by $\text{ASSIGN}_{\mathbb{A}_{Var \cup \mathcal{F}}}$. That is, we first merge

[4] Note that ASSIGN is an abbreviation for both F-ASSIGN and B-ASSIGN.

Algorithm 2: $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$ **when vars**$(ae) \subseteq \boldsymbol{Var} \cup \mathcal{F}$

1 **if** isLeaf(t) **then return** $\texttt{ASSIGN}_{\mathbb{A}_{\boldsymbol{Var}\cup\mathcal{F}}}(t \uplus_{\boldsymbol{Var}\cup\mathcal{F}} C, \texttt{x:=}ae)$;
2 **else return** $[\![t.c : \texttt{ASSIGN}_{\mathbb{T}}(t.l, \texttt{x:=}e, C \cup \{t.c\}), \texttt{ASSIGN}_{\mathbb{T}}(t.r, \texttt{x:=}e, C \cup \{\neg t.c\})]\!]$;

Algorithm 3: $\texttt{FILTER}_{\mathbb{T}}(t, be, C)$ **when vars**$(be) \subseteq \boldsymbol{Var} \cup \mathcal{F}$

1 **if** isLeaf(t) **then**
2 $\quad a' = \texttt{FILTER}_{\mathbb{A}_{Var\cup\mathcal{F}}}(t \uplus_{Var\cup\mathcal{F}} C, be)$;
3 $\quad J = a' \restriction_{\mathcal{F}}$;
4 $\quad$ **if** *isRedundant*(J, C) **then return** $\ll a' \gg$;
5 $\quad$ **else return** $\texttt{RESTRICT}(\ll a' \gg, C, J \backslash C)$;
6 **else return** $[\![t.c : \texttt{FILTER}_{\mathbb{T}}(t.l, \texttt{x:=}e, C \cup \{t.c\}), \texttt{FILTER}_{\mathbb{T}}(t.r, \texttt{x:=}e, C \cup \{\neg t.c\})]\!]$;

constraints from the leaf node t defined over $Var \cup \mathcal{F}$ and constraints from decision nodes $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}})$ defined over $\mathcal{F}$, by using $\uplus_{Var\cup\mathcal{F}}$ operator, and then we apply $\text{ASSIGN}_{\mathbb{A}_{Var\cup\mathcal{F}}}$ on the obtained result (Line 1).

Transfer function $\text{FILTER}_{\mathbb{T}}$ for test be, when vars$(be) \subseteq Var \cup \mathcal{F}$, is described by Algorithm 3. Similarly to $\text{ASSIGN}_{\mathbb{T}}$ in Algorithm 2, it accumulates the constraints along the paths in a set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ up to the leaf nodes, and applies $\text{FILTER}_{\mathbb{A}_{Var\cup\mathcal{F}}}$ on an abstract element obtained by merging constraints in the leaf node and in C (Line 2). The obtained result a' is a new leaf node, and additionally a' is projected on feature variables using $\restriction_{\mathcal{F}}$ operator to generate a new set of constraints J that is added to the given path to a' by using the function `RESTRICT` [16] (Lines 3–5). The function $\texttt{isRedundant}(J, C)$ checks if the constraints from J are redundant with respect to the set C.

Finally, transfer function for `#if` directives is defined as:

$$[\![\texttt{\#if}\ (\theta)\ s\ \texttt{\#end}]\!]_{\mathbb{T}} t = [\![s]\!]_{\mathbb{T}} \text{FILTER}_{\mathbb{T}}(t, \theta, \mathcal{K}) \sqcup_{\mathbb{T}} \text{FILTER}_{\mathbb{T}}(t, \neg\theta, \mathcal{K})$$

where $[\![s]\!]_{\mathbb{T}}(t)$ is transfer function for s and $\text{FILTER}_{\mathbb{T}}(t, \theta, \mathcal{K})$ is defined by Algorithm 3 since θ contains only features. Transfer function for assertions is: $[\![\texttt{assert}(be)]\!]_{\mathbb{T}} = \text{FILTER}_{\mathbb{T}}(t, be, \mathcal{K})$.

After applying transfer functions, the obtained decision trees may contain some redundancy that can be exploited to further compress them. We use several optimizations [16]. E.g., if constraints on a path to some leaf are unsatisfiable, we eliminate that leaf node; if a decision node contains two same subtrees, then we keep only one subtree and we also eliminate the decision node, etc.

4.4 Decision tree-based lifted analysis

Operations and transfer functions of $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$ and $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$ are used to perform the numerical and termination lifted analysis of program families, respectively. The *numerical lifted analysis* derived from $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$, written as $\mathbb{T}^F$ for short, is a pure *forward* analysis that infers numerical invariants in all program locations.

We define the analysis function $[\![s]\!]_{\mathbb{T}^F} t$ that takes as input a decision tree t corresponding to the initial location of statement s, and outputs a decision tree over-approximating the numerical invariant in the final location of s. The input decision tree $t^{\mathcal{K}}_{in,F}$ at the initial location of a program family has only one leaf node $\top_{\mathbb{D}_{Var \cup \mathcal{F}}}$ and decision nodes that define the set $\mathcal{K}$. Lifted invariants are propagated forward from the initial location towards the final location taking assignments, `#if`-s, and tests into account with widening and narrowing around `while`-s. We apply delayed widening [9], which means that we start extrapolating by widening after a fixed number of iterations of a loop are analyzed explicitly.

Similarly, we define the *termination lifted analysis* derived from $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$, written as $\mathbb{T}^B$ for short. It is a pure *backward* analysis that infers ranking functions in all program locations. We define the analysis function $[\![s]\!]_{\mathbb{T}^B} t$ that takes as input a decision tree t in the final location of statement s, and outputs a decision tree over-approximating the ranking function in the initial location of s. The input decision tree $t^{\mathcal{K}}_{in,B}$ at the final location of a program family has only one leaf node 0 (zero function) and decision nodes that define the set $\mathcal{K}$. Lifted ranking functions are propagated backward from the final towards the initial location.

We establish correctness of the lifted analysis based on $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ by showing that it produces identical results with the `Brute-Force` enumeration approach based on the domain $\mathbb{A}$. Let $[\![s]\!]_{\mathbb{T}}$ denotes the transfer function of statement s of $\overline{\text{IMP}}$ in $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$, while $[\![s]\!]_{\mathbb{A}}$ denotes the transfer function of statement s of IMP in $\mathbb{A}$. Given $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$, we denote by $Pr_k(t) \in \mathbb{A}$ the leaf node of tree t that corresponds to the variant $k \in \mathcal{K}$.

Theorem 2. *$Pr_k([\![s]\!]_{\mathbb{T}}(t)) = [\![\pi_k(s)]\!]_{\mathbb{A}}(Pr_k(t))$ for all $k \in \mathcal{K}$.*

Example 2. In Figs. 7 and 8 we depict decision trees at locations ② and ⓗ inferred by performing (forward) numerical analysis based on the domain $\mathbb{T}(\mathbb{C}_P, P)$ of the Loop1A program family (see Section 2). In order to enforce convergence of the analysis, we apply the widening operator at the loop head, i.e. at the location ⓗ before the `while` test. We can see how the invariant at location ⑤ shown in Fig. 1 is inferred from the invariant at location ⓗ.

Subsequently, we perform a (backward) lifted termination analysis based on the domain $\mathbb{T}(\mathbb{C}_P, \mathbb{T}^T)$ of the Loop1A sub-family satisfying (`A-B` ≥ 3). Lifted decision trees inferred at locations ⓗ and ① are shown in Figs. 9 and 2, respectively. We can see how by back-propagating the tree at location ⓗ, denoted $t_{(h)}$ (see Fig. 9), via assignments `y := 0` and `x := A` at location ①, we obtain the tree at location ①, denoted $t_{(1)}$ (see Fig. 2). The transfer function B-ASSIGN$_{\mathbb{T}}(t_{(h)}$, `x := A`$)$ will generate the tree $t_{(1)}$ where `x` is replaced with `A`. The new decision node (`A` $\geq$ `B+1`) and the leaf node with ranking function 2 are eliminated from $t_{(1)}$ since they are redundant with respect to (`A-B` ≥ 3).

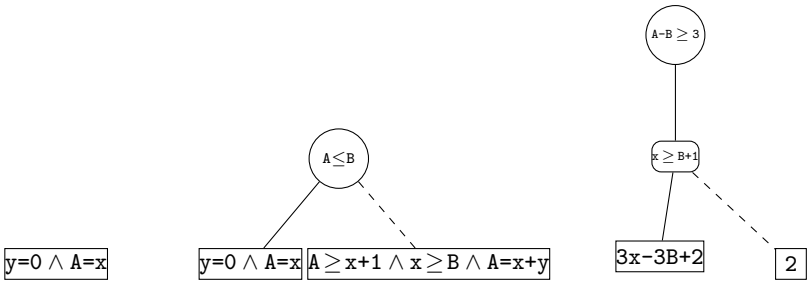


Fig. 7: Invariant at loc. ② of Loop1A.

Fig. 8: Invariant at loc. ⓗ of Loop1A.

Fig. 9: Ranking fun. at loc. ⓗ of Loop1A.

5 Synthesis Algorithm

We can now solve the quantitative sketching problem using lifted analysis algorithms. More specifically, we delegate the effort of conducting an effective search of all possible sketch realizations to an efficient lifted static analyzer, which combines the forward numerical and the backward termination analyses.

The synthesis algorithm SYNTHESIZE($\hat{s}$: Stm) for solving a sketch $\hat{s}$ is given in Algorithm 4. First, we transform the program sketch $\hat{s}$ into a program family $\overline{s} = \texttt{Rewrite}(\hat{s})$ (Line 1). Then, we call function $[\![\overline{s}]\!]_{\mathbb{T}^F} t^{\mathcal{K}}_{in,F}$ to perform the forward numerical lifted analysis of $\overline{s}$. The inferred decision tree t_F at the final location of $\overline{s}$ is analyzed by function FindCorrect (Line 3) to find the sets of variants for which non-$\bot_{\mathbb{D}}$ and non-$\top_{\mathbb{D}}$ leaf nodes are reachable. The set of variants for which $\bot_{\mathbb{D}}$ leaf node is reachable are "incorrect" with respect to the given assertions; whereas the set of variants for which $\top_{\mathbb{D}}$ leaf node is reachable are "I don't know" (inconclusive). For each non-$\bot_{\mathbb{D}}$ and non-$\top_{\mathbb{D}}$ leaf node, we generate the set of variants $\mathcal{K}' \subseteq \mathcal{K}$ that satisfy the encountered linear constraints along the top-down path to that leaf node as well as the given assertions. For each such "correct" set of variants $\mathcal{K}$', we perform the backward termination lifted analysis $[\![\overline{s}]\!]_{\mathbb{T}^B} t^{\mathcal{K}'}_{in,B}$. The inferred decision tree t_B is analyzed by function FindOptimal (Line 7). It calls the Z3 solver [26] to solve the following optimization problem: find a model that *minimizes* the value of ranking functions $t' \in \mathbb{T}^T$, such that the linear constraints along the top-down paths to those leaf nodes are satisfied. More formally, given a decision tree $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$, we define the function $\phi[C]t$ that finds a set of pairs (k, t') consisting of valid configurations $k \in \mathcal{K}$ and the corresponding ranking function $t' \in \mathbb{T}^T$ as follows:

$$\phi[C](\ll t' \gg) = \{(k, t') \mid k \in \mathcal{K}, k \models C\},\ \phi[C]([\![c\!:\!tl, tr]\!]) = \phi[C \cup \{c\}](tl) \cup \phi[C \cup \{\neg c\}](tr)$$

The optimization problem is the following. Given a decision tree $t_B \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$ inferred at the initial location of $\overline{s}$, find a configuration $k \in \mathcal{K}$ such that the corresponding ranking function is minimal. That is, $min_{k \in \mathcal{K}}\{t' \mid (k, t') \in \phi[\mathcal{K}]t_B\}$.

The configuration k with the minimal ranking function found by Z3 is reported as a "correct and optimal" solution to the quantitative sketching problem.

Theorem 3. *SYNTHESIZE($\hat{s}$) is correct and terminates.*

Algorithm 4: SYNTHESIZE($\hat{s} : Stm$)

```
1 s̄ = Rewrite(ŝ);
2 t_F = [[s̄]]_{T^F} t^K_{in,F};
3 C = FindCorrect(t_F);
4 while C ≠ ∅ do
5 |   K' = C.remove();
6 |   t_B = [[s̄]]_{T^B} t^{K'}_{in,B};
7 |   sol.insert(FindOptimal(t_B))
8 return sol
```

6 Evaluation

We evaluate our approach for program sketching by comparing it with the `Brute-Force` enumeration approach and the popular `Sketch` tool.

Implementation We have implemented our synthesis algorithm for quantitative program sketching [14] within the FamilySketcher tool [17]. It uses the lifted decision tree domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$, where $\mathbb{A}$ is instantiated either to numerical abstract domain $\mathbb{D}$ or to the termination decision tree domain $\mathbb{T}^T$. The abstract operations and transfer functions for the numerical domain $\mathbb{D}$: intervals, octagons, polyhedra, are provided by the APRON library [23], while for the termination decision tree domain are provided by the `Function` tool [32]. The tool is written in OCaml and consists of around 7K LOC. The current tool provides a limited support for arrays, pointers, `struct` and `union` types. The only basic data type is mathematical integers, which is sufficient for our evaluation.

Within the FamilySketcher, we have also implemented the `Brute-Force` enumeration approach that analyzes all variants (sketch realizations), one by one, using the single-program domains $\mathbb{D}$ and $\mathbb{T}^T$.

Experiment setup and Benchmarks All experiments are executed on a 64-bit Intel®Core^{TM} i7-1165G7 CPU@2.80GHz, VM LUbuntu 20.10, with 8 GB memory, and we use a timeout value of 300 seconds. All times are reported as average over five independent executions. We report times needed for the actual analysis task to be performed. The implementation is available from [14]: https://zenodo.org/record/5898643#.YhJLRejMLIU. We compare our approach with program sketching tool `Sketch` version 1.7.6 that uses SAT-based counterexample-guided inductive synthesis [30,29], and with the `Brute-Force` enumeration approach. The evaluation is performed on several C numerical sketches collected from the `Sketch` project [30,29], SV-COMP (https://sv-comp.sosy-lab.org/), and the SyGuS-Competition [1]. We use the following benchmarks: Loop1a and Loop1b (Sec. 2), Loop2a and Loop2b (Fig. 10), LoopCond (Fig. 11), NestedLoop (Fig. 12), vmcai2004 (Fig. 13).

Performance Results Table 1 shows the results of synthesizing our benchmarks. Note that `Sketch` reports only one "correct" solution for each sketch, which

```
void main() {
  int x;
  int y := ??1;
  while (x ≤ 10) {
    if (y ≥??2) x := x+1;
    else x := x-1; }
  assert (y > 2);
  //assert (y < 8);
}
```

Fig. 10: Loop2a.

```
void main(unsigned int x){
  int y := ??1;
  while (x ≥ 0) {
    x := x-1;
    if (y<??) y := y+1;
    else y := y-1; }
  assert (y ≥ 1);
}
```

Fig. 11: LoopCond.

```
void main(unsigned int x){
  int s := 0, y := ??1;
  int x0 := x, y0 := y;
  while (x ≥ 0) {
    x := x-1;
    while (y ≥??2) {
      y := y-1; s := s+1; }
  } assert (s ≥ x0+y0);
}
```

Fig. 12: NestedLoop.

```
void main(){
  int x := ??1, y:=0;
  while (x > 0) {
    x := -??2*x+10;
    y := y+1;
  }
  assert (y≤2);
}
```

Fig. 13: vmcai2004.

does not have to be "optimal" with respect to the given quantitative objective. FamilySketcher and `Brute-Force` use the polyhedra domain as parameter.

The Loop1a and Loop1b sketches are handled symbolically by (SR) rule. Thus, our approach does not depend on sizes of hole domains. FamilySketcher terminates in (around) 0.016 sec for Loop1a and in 0.026 sec for Loop1b. In contrast, `Brute-Force` and `Sketch` do depend on the sizes of holes. `Sketch` terminates in 37.74 sec (resp., 2.44 sec) for 16-bits sizes of holes for Loop1a (resp., Loop1b). It times out for bigger sizes of Loop1a. `Sketch` often reports "correct & optimal" solutions for both sketches. Similarly, our tool can handle symbolically Loop2a and Loop2b in 0.060 sec and 0.047 sec. However, `Sketch` cannot resolve them, since it uses 8 unrollments of the loop by default. If the loop is unrolled 11 times, `Sketch` terminates but often reports the empty solution.

The LoopCond sketch contains one hole that can be handled symbolically by (SR) rule. FamilySketcher has similar running times for all domain sizes reporting the solution `??` ≥ 2 and ranking function `4x+8`. In contrast, `Sketch` resolves this example only if the loop is unrolled as many times as is the size of the hole and inputs (e.g., 32 times for 5-bits). Hence, `Sketch`'s performance declines with the growth of size of the hole, and times out for 16-bits.

The NestedLoop sketch contains two holes that can be handled symbolically by (SR) rule. FamilySketcher terminates in (around) 0.126 sec for all sizes of holes. The "correct" solution is $(\texttt{??}_1 - \texttt{??}_2 \geq 0) \wedge (\texttt{Min} \leq \texttt{??}_2 \leq 1)$, while the "correct & optimal" solution is $(\texttt{??}_1 = \texttt{??}_2 = 0)$ with ranking function 13. On the other hand, `Brute-Force` takes 65.03 sec for 5-bit size of holes and times out for larger sizes, while `Sketch` cannot resolve this benchmark.

The vmcai2004 sketch contains two holes. The first one $??_1$ is handled symbolically by (SR) rule while the second one $??_2$ explicitly by (ER) rule. The performance of FamilySketcher depends on the size of $\texttt{??}_2$. The decision tree inferred in the location before the assertion contains one leaf node for each possible value of feature `B` (features `A` and `B` represent $??_1$ and $??_2$). The sub-family of "correct" solutions is: $(1 \leq \texttt{A} \leq \texttt{Max}) \wedge (\texttt{B} \geq 10)$, while the "correct & optimal" solution is $(\texttt{A}{=}1) \wedge (\texttt{B}{=}10)$ with ranking function 6. `Sketch` scales better in this case reporting one "correct" (but not "optimal") solution. However, FamilySketcher still outperforms the `Brute-Force` approach.

Table 1: Performance results of FamilySketcher vs. `Sketch` vs. `Brute-Force`. FamilySketcher and `Brute-Force` use Polyhedra domain. All times in sec.

Bench.	5 bits			6 bits			16 bits		
	Family Sketcher	`Sketch`	`Brute Force`	Family Sketcher	`Sketch`	`Brute Force`	Family Sketcher	`Sketch`	`Brute Force`
`Loop1a`	0.016	0.192	4.66	0.017	0.197	21.33	0.017	37.74	timeout
`Loop1b`	0.026	0.203	4.77	0.026	0.216	21.38	0.027	2.44	timeout
`Loop2a`	0.060	0.200	8.66	0.060	0.202	42.81	0.061	0.348	timeout
`Loop2b`	0.047	0.203	8.45	0.047	0.205	36.04	0.049	0.521	timeout
`LoopCond`	0.042	0.207	1.19	0.042	0.209	2.56	0.043	timeout	timeout
`NestedLoop`	0.126	timeout	65.03	0.126	timeout	timeout	0.128	timeout	timeout
`vmcai2004`	4.69	0.192	5.12	15.52	0.229	19.12	timeout	0.292	timeout

Discussion In summary, we can see that FamilySketcher often outperforms `Sketch`, especially in case of sketches that can be handled symbolically by (SR) rule. But, for sketches with holes that need to be handled by (ER) rule, the performances of our tool decline, which is the consequence of the need to explicitly consider all values of those holes. However, even in this case FamilySketcher scales better than `Brute-Force`. This is due to the fact that `Brute-Force` compiles and executes the fixed-point iterative algorithm once for each variant, while our approach does it once per whole family plus there are still possibilities for sharing. Moreover, FamilySketcher reports the "correct & optimal" solution, while `Sketch` reports the first found "correct" solution.

Threats to validity The current tool has only limited support for arrays, pointers, `struct` and `union` types. However, the above features are largely orthogonal to the solution proposed here. In particular, these features complicate the semantics of single-programs and implementation of domains for leaf nodes, but have no impact on the semantics of variability-specific constructs. We perform lifted analysis of relatively small benchmarks. However, the focus of our approach is to combat the realization space blow-up of sketches, not their LOC size. So, we expect to obtain similar or better results for larger benchmarks. Although we analyze relatively small set of benchmarks, we expect the results to carry over the other benchmarks.

7 Related Work

The proposed program sketcher uses numerical abstract domains as parameters, so it can be applied for synthesizing programs with numerical data types. The existing widely-known sketching tool `Sketch` [29,30], which uses SAT-based counterexample-guided inductive synthesis, is more general and especially suited for synthesizing bit-manipulating programs. However, `Sketch` reasons about loops by unrolling them, so is very sensitive to the degree of unrolling. Our approach being based on abstract interpretation does not have this constraint,

since we use the widening extrapolation operator to handle unbounded loops and an infinite number of execution paths in a sound way. This is stronger than fixing a priori a bound on the number of iterations of loops as in the `Sketch` tool. Moreover, `Sketch` may need several iterations to converge reporting only one solution. On the other hand, our approach needs only one iteration to perform lifted analysis reporting several, and very often all, "correct" solutions. This is the key for applying our approach to solve the quantitative sketching problem. Another work for solving a quantitative sketching problem is proposed by Chaudhuri et. al [6]. The quantitative optimum they consider is that the expected output value on probabilistic inputs is minimal [5]. They use smoothed proof search and probabilistic analysis to implement this approach in the FERMAT tool built on top of `Sketch`. In contrast, in this work the quantitative optimum we consider is that the worst-case behavior of the program is minimal.

Recently, there have been proposed several works that solve the sketching synthesis problem using product line analysis and verification algorithms. Ceska et. al. [4] use a counterexample guided abstraction refinement technique for analyzing product lines to resolve probabilistic `PRISM` sketches. The work [17] uses a (forward) numerical lifted analysis based on abstract interpretation to resolve numerical sketches. We extend here this approach by considering the more general quantitative sketching problem, where we additionally employ a (backward) termination lifted analysis to find a solution that is not only "correct" but also "optimal" to the given quantitative objective.

Several lifted analysis based on abstract interpretation have been proposed recently [24,11,12,16,18,15,13] for analyzing program families with `#if`-s. Midtgaard et. al. [24] have proposed the lifted tuple-based analysis, while the work [11,12] improves the tuple representation by using lifted binary decision diagram (BDD) domains. They are applied to program families with only Boolean features. Subsequently, the lifted decision tree domain has been proposed to handle program families with both Boolean and numerical features [16,18], as well as dynamic program families where features can change during run-time [15]. The above lifted analyses are forward and infer numerical invariants, while a backward termination analysis for inferring ranking functions is proposed in [13].

Decision-tree abstract domains have been used in abstract interpretation community recently [10,7,32]. Segmented decision tree abstract domains have enabled path dependent static analysis [10,7]. Their elements contain decision nodes that are determined either by values of program variables [10] or by the `if` conditions [7], whereas the leaf nodes are numerical properties. Urban and Miné [31,32] use decision tree abstract domains to prove program termination.

8 Conclusion

In this work, we proposed a new approach for synthesis of program sketches, such that the resulting program satisfies the combined boolean and quantitative specifications. We have shown that both reasoning tasks can be accomplished using a combination of forward and backward lifted analysis. We experimentally demonstrate the effectiveness of our approach on a variety of C benchmarks.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013. pp. 1–8. IEEE (2013), http://ieeexplore.ieee.org/document/6679385/
2. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Computer Aided Verification, 21st International Conference, CAV 2009. Proceedings. LNCS, vol. 5643, pp. 140–156. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_14, https://doi.org/10.1007/978-3-642-02658-4_14
3. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: Spl^{lift}: statically analyzing software product lines in minutes instead of years. In: ACM SIGPLAN Conference on PLDI '13. pp. 355–364 (2013)
4. Ceska, M., Dehnert, C., Jansen, N., Junges, S., Katoen, J.: Model repair revamped: On the automated synthesis of markov chains. In: Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday. LNCS, vol. 11500, pp. 107–125. Springer (2019). https://doi.org/10.1007/978-3-030-31514-6_7, https://doi.org/10.1007/978-3-030-31514-6_7
5. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: Computer Aided Verification, 22nd International Conference, CAV 2010. Proceedings. LNCS, vol. 6174, pp. 380–395. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_34, https://doi.org/10.1007/978-3-642-14295-6_34
6. Chaudhuri, S., Clochard, M., Solar-Lezama, A.: Bridging boolean and quantitative synthesis using smoothed proof search. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14. pp. 207–220. ACM (2014). https://doi.org/10.1145/2535838.2535859, https://doi.org/10.1145/2535838.2535859
7. Chen, J., Cousot, P.: A binary decision tree abstract domain functor. In: Static Analysis - 22nd International Symposium, SAS 2015, Proceedings. LNCS, vol. 9291, pp. 36–53. Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_3, https://doi.org/10.1007/978-3-662-48288-9_3
8. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conf. Record of the Fourth ACM Symposium on POPL. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973, http://doi.acm.org/10.1145/512950.512973
10. Cousot, P., Cousot, R., Mauborgne, L.: A scalable segmented decision tree abstract domain. In: Time for Verification, Essays in Memory of Amir Pnueli. LNCS, vol. 6200, pp. 72–95. Springer (2010). https://doi.org/10.1007/978-3-642-13754-9_5, https://doi.org/10.1007/978-3-642-13754-9_5
11. Dimovski, A.S.: Lifted static analysis using a binary decision diagram abstract domain. In: Proceedings of the 18th ACM SIGPLAN International Conference on GPCE 2019. pp. 102–114. ACM (2019). https://doi.org/10.1145/3357765.3359518, https://doi.org/10.1145/3357765.3359518
12. Dimovski, A.S.: A binary decision diagram lifted domain for analyzing program families. J. Comput. Lang. **63**, 101032 (2021).

https://doi.org/10.1016/j.cola.2021.101032, https://doi.org/10.1016/j.cola.2021.101032
13. Dimovski, A.S.: Lifted termination analysis by abstract interpretation and its applications. In: GPCE '21: Concepts and Experiences, Chicago, IL, USA, October, 2021. pp. 96–109. ACM (2021). https://doi.org/10.1145/3486609.3487202, https://doi.org/10.1145/3486609.3487202
14. Dimovski, A.S.: Tool artifact for "quantitative program sketching using lifted static analysis". Zenodo (2022). https://doi.org/10.5281/zenodo.5898643, https://zenodo.org/record/5898643#.YhJLRejMLIU
15. Dimovski, A.S., Apel, S.: Lifted static analysis of dynamic program families by abstract interpretation. In: 35th European Conference on Object-Oriented Programming, ECOOP 2021. LIPIcs, vol. 194, pp. 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.14, https://doi.org/10.4230/LIPIcs.ECOOP.2021.14
16. Dimovski, A.S., Apel, S., Legay, A.: A decision tree lifted domain for analyzing program families with numerical features. In: Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Proceedings. LNCS, vol. 12649, pp. 67–86. Springer (2021), https://arxiv.org/abs/2012.05863
17. Dimovski, A.S., Apel, S., Legay, A.: Program sketching using lifted analysis for numerical program families. In: NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings. LNCS, vol. 12673, pp. 95–112. Springer (2021). https://doi.org/10.1007/978-3-030-76384-8_7, https://doi.org/10.1007/978-3-030-76384-8_7
18. Dimovski, A.S., Apel, S., Legay, A.: Several lifted abstract domains for static analysis of numerical program families. Sci. Comput. Program. **213**, 102725 (2022). https://doi.org/10.1016/j.scico.2021.102725, https://doi.org/10.1016/j.scico.2021.102725
19. Dimovski, A.S., Brabrand, C., Wasowski, A.: Variability abstractions for lifted analysis. Sci. Comput. Program. **159**, 1–27 (2018)
20. Dimovski, A.S., Brabrand, C., Wasowski, A.: Finding suitable variability abstractions for lifted analysis. Formal Aspects Comput. **31**(2), 231–259 (2019). https://doi.org/10.1007/s00165-019-00479-y, https://doi.org/10.1007/s00165-019-00479-y
21. Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M., Apel, S.: Preprocessor-based variability in open-source and industrial software systems: An empirical study. Empirical Software Engineering **21**(2), 449–482 (2016). https://doi.org/10.1007/s10664-015-9360-1, https://doi.org/10.1007/s10664-015-9360-1
22. Iosif-Lazar, A.F., Melo, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Effective analysis of C programs by rewriting variability. Art Sci. Eng. Program. **1**(1), 1 (2017). https://doi.org/10.22152/programming-journal.org/2017/1/1, https://doi.org/10.22152/programming-journal.org/2017/1/1
23. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceedings. LNCS, vol. 5643, pp. 661–667. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52, https://doi.org/10.1007/978-3-642-02658-4_52
24. Midtgaard, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Systematic derivation of correct variability-aware program analyses. Sci. Comput. Program. **105**, 145–170 (2015). https://doi.org/10.1016/j.scico.2015.04.005, http://dx.doi.org/10.1016/j.scico.2015.04.005

25. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Programming Languages **4**(3-4), 120–372 (2017). https://doi.org/10.1561/2500000034, https://doi.org/10.1561/2500000034
26. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
27. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag, Secaucus, USA (1999)
28. von Rhein, A., Liebig, J., Janker, A., Kästner, C., Apel, S.: Variability-aware static analysis at scale: An empirical study. ACM Trans. Softw. Eng. Methodol. **27**(4), 18:1–18:33 (2018). https://doi.org/10.1145/3280986, https://doi.org/10.1145/3280986
29. Solar-Lezama, A.: Program sketching. STTT **15**(5-6), 475–495 (2013). https://doi.org/10.1007/s10009-012-0249-7, https://doi.org/10.1007/s10009-012-0249-7
30. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation. pp. 281–294. ACM (2005). https://doi.org/10.1145/1065010.1065045, https://doi.org/10.1145/1065010.1065045
31. Urban, C.: Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs. (Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes). Ph.D. thesis, École Normale Supérieure, Paris, France (2015), https://tel.archives-ouvertes.fr/tel-01176641
32. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Static Analysis - 21st International Symposium, SAS 2014. Proceedings. LNCS, vol. 8723, pp. 302–318. Springer (2014). https://doi.org/10.1007/978-3-319-10936-7_19, https://doi.org/10.1007/978-3-319-10936-7_19

SixthSense: Debugging Convergence Problems in Probabilistic Programs via Program Representation Learning

Saikat Dutta(✉), Zixin Huang, and Sasa Misailovic

University of Illinois, Urbana, Illinois, 61820, USA
{saikatd2,zixinh2,misailo}@illinois.edu

Abstract. Probabilistic programming aims to open the power of Bayesian reasoning to software developers and scientists, but identification of problems during inference and debugging are left entirely to the developers and typically require significant statistical expertise. A common class of problems when writing probabilistic programs is the lack of convergence of the probabilistic programs to their posterior distributions.

We present SixthSense, a novel approach for predicting probabilistic program convergence ahead of run and its application to debugging convergence problems in probabilistic programs. SixthSense's training algorithm learns a classifier that can predict whether a previously unseen probabilistic program will converge. It encodes the syntax of a probabilistic program as *motifs* – fragments of the syntactic program paths. The decisions of the classifier are interpretable and can be used to suggest the program features that contributed significantly to program convergence or non-convergence. We also present an algorithm for augmenting a set of training probabilistic programs that uses guided mutation.

We evaluated SixthSense on a broad range of widely used probabilistic programs. Our results show that SixthSense features are effective in predicting convergence of programs for given inference algorithms. SixthSense obtained Accuracy of over 78% for predicting convergence, substantially above the state-of-the-art techniques for predicting program properties Code2Vec and Code2Seq. We show the ability of SixthSense to guide the debugging of convergence problems, which pinpoints the causes of non-convergence significantly better by Stan's built-in warnings.

Keywords: Probabilistic Programming · Debugging · Machine Learning

1 Introduction

Probabilistic programs (PP) express complicated Bayesian models as simple computer programs, used in various domains [22, 38, 44, 54], including the important applications like epidemic modeling [23] and single-cell genomics [42]. Probabilistic languages extend the conventional languages with constructs for sampling from probabilistic distributions (prior), conditioning on data, and probabilistic queries, such as the distribution reshaped by conditioning on the data (posterior) [26]. Probabilistic programming

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 123–144, 2022.
https://doi.org/10.1007/978-3-030-99429-7_7

systems (PP systems) compile the programs and compute the results using an efficient inference algorithm, while hiding the intricate details of inference. Most practical inference algorithms are non-deterministic and approximate. For instance, Markov Chain Monte Carlo (MCMC) algorithms [28, 40, 48] run a probabilistic program multiple times (each of which is referred to as an *iteration*) to sample data points from the posterior distribution. They drive today's popular PP systems, such as Stan [9].

MCMC algorithms have a nice theoretical property: in the limit, the samples they generate come from the correct posterior distribution. But, in practice, a user can only execute the algorithm for a finite time budget and hence needs to fine-tune the algorithms to balance between quality of inference and execution time. This complicates development: the programmer needs to write the program in a way that interacts well with the algorithm and select some parameters specific for the inference algorithms. For instance, inference may fail to properly initialize, silently produce inaccurate results, or generate non-independent samples from the posterior distribution. Even identifying and afterward resolving these challenges currently requires significant statistical expertise.

An important property for successful inference is *convergence*, since non-convergence is often a cause of inaccurate (or wrong) result. Convergence means the samples generated by the inference algorithm represent the target distribution. While there exists metrics for convergence (e.g. Gelman-Rubin diagnostic [25]) in statistic literature, there lacks a comprehensive study of what model features could cause non-convergence. Thus, getting a data-driven understanding of the causes could help developers to debug the non-convergence issues, and does not require expert knowledge. Moreover, the existing convergence diagnostics are *not predictive* – they cannot be determined ahead of time i.e. without running the program. Building prediction model for converges ahead of time would save the time to run programs (often taking minutes or more). It would also enable a faster program debug/update cycle.

1.1 SixthSense

We present SixthSense, the first approach for identifying convergence problems in probabilistic programs ahead-of-run. SixthSense adopts a learning approach: its trains a classifier that can, for a previously unseen probabilistic program and its data, predict whether the program will converge in a specified number of steps (for a given threshold of Gelman-Rubin diagnostic). The decisions of the classifier are interpretable and can be used to suggest which program features leads to the convergence/non-convergence of the program.

To train such a classifier, SixthSense needs to overcome several challenges that are beyond the big-code techniques studied for conventional languages [4, 5, 31, 37, 47]. First, probabilistic programs are small (20-100 lines of code) compared to conventional programs but their execution is complicated, with conditioning statements for data and non-standard semantics that performs Bayesian inference. Second, due to their relative novelty, there are few publicly-available probabilistic programs that can be used for training. Finally, we should be able to interpret why the programs are predicted to convergence or non-convergence in order to guide developers to debug the non-convergence issues.

Representing Structural, Data, and Runtime Features: To learn a classifier, we embed the syntactic and semantic program features in a numerical vector. To encode program structure, we observe that many snippets of code in probabilistic programs form patterns (sampling from distributions, hierarchical models, relations between variables) that may repeat within the single program or across programs. We identify those patterns as *motifs* – fragments of probabilistic program code, consisting of several adjacent abstract-syntax-tree nodes (e.g., neighboring statements or expressions).

SixthSense learns the set of features from the subset of motifs it identifies in the code. It groups together similar motifs by calculating a low-dimensional representation of the motifs using randomized discrete projections [8]. This way, it can balance the accuracy of prediction and the size of the learned models. We also engineered a set of data features (e.g., means, variances) and the runtime features – diagnostics from early warmup iterations that the inference algorithms compute as they execute. These features cannot be learned by the approaches that focus on static code features [4, 5, 31, 47].

Mutation-Based Program Generation: We present a novel technique based on program and data mutations that produces a diverse set of probabilistic programs with a good balance between converging and non-converging programs, with the goal to augment the training set. Our technique takes a set of seed programs as input, analyzes them and applies a set of pre-defined mutations which aim to change the semantics of generated programs. To obtain better diversity, our algorithm identifies (via locality-sensitive hashing [6]) and discards any mutant that is too similar to the one that was generated before.

Interpretable Predictor Results: For problem diagnosis and debugging of probabilistic programs, it is important to be able to interpret why the algorithm predicted non-convergence. Our learning algorithm leverages random forests for this task. It relates the likely cause of non-convergence to specific statements or expressions in the program code.

1.2 Results

In this work, we learn the classifiers for convergence of three popular classes of probabilistic programs: *Regression*, *Time Series*, and *Mixture Models*. We obtained 166 seed programs, across the three classes, from an open source repository of Stan programs [52]. For each class, SixthSense generated more than 10,000 mutants. We train our classifiers for multiple thresholds of the convergence score (Gelman-Rubin diagnostic) to evaluate the sensitivity of our classifiers.

Our evaluation shows the effectiveness of SixthSense in predicting convergence of probabilistic programs compared to two state-of-the-art learning algorithms for conventional code: Code2Vec [5] and Code2Seq [4]. We measure the prediction quality via *Accuracy* (ratio of sum of True Positives and True Negatives to total tested programs), *Precision* (ratio of True Positives to total classified as Positives) and *Recall* (ratio of True Positives to total actual Positives). Here True Positive is a program that is predicted to converge and it indeed converges; the others are defined analogously.

SixthSense obtains an average Accuracy score across the three model classes of 78% for convergence prediction (with almost equally high prediction and recall). SixthSense, with just code features outperforms Code2Vec [5] by 8 percentage points on

average and Code2Seq [4] by 5 percentage points on average (for a tight convergence threshold). Moreover, we also show that Accuracy scores increase to over 83% when adding runtime features obtained after just the first 10-200 samples from the warmup stage of the inference algorithm (which is less than 10% of its run-time). SixthSense also has higher precision for all model classes, and recall higher than Code2Vec but similar to Code2Seq. SixthSense's prediction time is less than a second and the model size is modest – less than 20 MB, which is 25-37% smaller than Code2Vec/Code2Seq.

We further demonstrate, by studying 40 non-converging programs, that SixthSense can pinpoint the locations in the code that cause non-convergence for 29 programs. In contrast, Stan's runtime warnings point to non-convergence causes in only 5 programs.

1.3 Contributions

We highlight the main contributions of this paper:

- **SixthSense System**[1]**.** SixthSense is a system for learning to predict convergence of probabilistic programs that aids programmers in pinpointing and understanding the sources of convergence problems in PPs.
- **Predicting convergence of probabilistic programs.** We present the first approach for learning predictors for convergence of probabilistic programs based on encoding the structure of probabilistic programs using code motifs.
- **Program generation for training set augmentation.** We present a new mutation algorithm for augmenting the training set with PPs that have diverse structural and runtime characteristics.
- **Experimental evaluation.** We show that SixthSense predicts convergence for three popular classes of programs, with higher accuracy, precision, and recall than two state-of-the-art approaches. In our case study SixthSense helps pinpoint likely cause of non-convergence for 29 out of 40 non-converging programs, compared to 5 programs for which Stan's runtime warnings help.

2 Example

We describe how SixthSense computes motifs, trains the predictor and demonstrate how we can use it to guide the debugging of probabilistic programs. Figure 1 shows two variants of a Mixture model in Stan. A Mixture Model is a probabilistic model that assumes that each observed data point comes from one out of N independent sub-distributions of values. Each sub-distribution has an associated probability (called mixing ratio) of being chosen.

The programs **A** and **B** in Figure 1(a), 1(b) have several (unknown) parameters: mean *mu* and variance *sigma* of the normal sub-distribution; *theta* is the mixing ratio of the sub-distributions and *p1* is an auxiliary parameter. The programs also access the array of observations, y, of size K. Each observation in y is assumed to be sampled from one of these two sub-distributions: a normal distribution (as *normal_lpdf*) or a uniform distribution (as the constant 0.5). For the program **B**,

[1] SixthSense is publicly available at https://github.com/uiuc-arc/sixthsense.

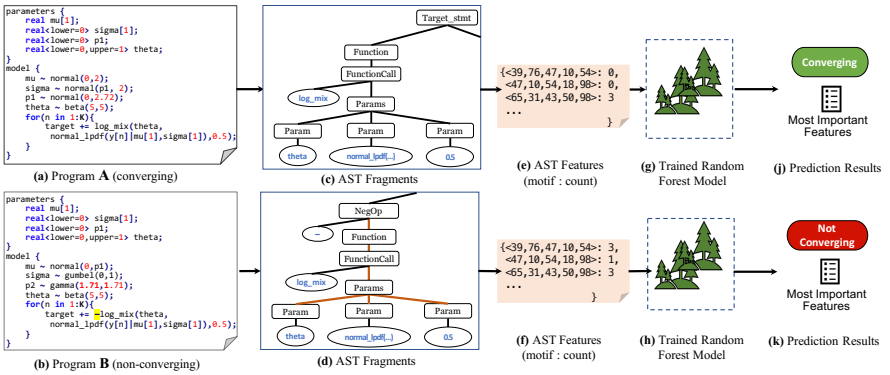


Fig. 1: An example of two models with different convergence behaviors. We obtain the features from the Abstract Syntax Tree (AST) of source code and data (not shown here). We use them as inputs to the trained Random Forest Model for predicting the label (Converging/Not Converging). We can also obtain the most important features which likely contributed to (non)-convergence.

consider a novice developer, who was confused about Stan's target statement [51], calculated the negative likelihood instead.

When run with Stan's default NUTS inference algorithm for 1000 iterations, the program **A** converges and the program **B** does not converge. Our goal is to predict, before running the programs, whether they will converge. If they do not converge, we would also want to know why and use this information to debug the program.

Feature Extraction. First, we extract different classes of features for each program in the corpus of mutants. These include *motifs* – fragments of the AST, augmented with data features, and run-time features. To extract motifs, we parse each program and construct an AST. Then, starting from each node, we obtain all AST paths of length L by traversing the ancestors of the node. Figures 1(c) and 1(d) present one sub-tree for the function call statement(in loop) in the programs **A** and **B** respectively and several motifs that SixthSense extracts. The elements in the motif are the sequence of the node type IDs as feature vectors.

A good learning algorithm should be able to combine similar motifs and operate only on groups of them. To identify such groups of motifs, we apply *random discrete projections*, a well-known technique for reducing the dimensionality of the feature space. It maps the feature vectors of the IDs onto a hash value with a much smaller dimension. The random projections algorithm has a *distance-preserving* property, which means that the similar vectors (even when they are not grouped together) will have similar low-dimensional representations. This property allows us to apply standard learning algorithms on this low-dimensional representation while preserving the similarity of the original motifs.

Computing Reference Solutions and Labels. To compute the program labels (i.e., 'converging', 'not-converging'), SixthSense runs them for the default 1000 iterations using Stan's MCMC algorithm (NUTS). For convergence, we calculate a

well-known diagnostic called Gelman-Rubin ($\hat{R}$) statistic [25]. *If the $\hat{R}$ statistic is within a certain bound (close to 1.0), it indicates that the program* converged.

Training. Given a sufficient number of training programs, SixthSense extracts the features and gets the labels for convergence. SixthSense then generates precise and interpretable predictors. We build separate models for predicting convergence for each model class, since models in three classes are significantly different in both semantics and the way they interact with inference algorithms. The model classes are easy to identify for users without expertise or through simple analytical tools.

Prediction. We use the classifier trained using the batch of Mixture Models for convergence. We use a threshold of 1.05 for Gelman-Rubin diagnostic (a very tight bound). SixthSense correctly predicts *True* label for program in Figure 1(a) and *False* label for program in Figure 1(b). The total time required for computing the features and doing the prediction for a single program is less than a second, compared to 53 seconds on average to run a program.

Interpretation and Debugging. Our combination of random projections – which groups very similar motifs together, even if they appear at different locations in the program – and the random forest classification – which can easily explain its decisions – proves effective in identifying the parts of the program that impede convergence. Namely, we can employ SixthSense's random forest classifier to identify top features. When SixthSense predicts non-convergence, the user can debug the program according to the top features.

Now consider the scenario where a novice Stan developer used negative log-likelihood in Stan's target statement, and wrote program **B** (Figure 1(b)). SixthSense predicts that **B** does not converge, and gives the topmost feature as the path segment (motif) starting from the negative sign to the parameters in the log-likelihood calculation (function *log_mix*). Figure 2 presents this motif. There were three such motifs in program **B** (one for each argument of the *log_mix* function), highly contributing to non-convergence prediction. In contrast, this motif is missing from program **A** (Figure 1(a)), and thus has negatively contributed in the converging prediction. This observation validates our earlier intuition about the cause of difference in the nature of two programs and is correctly inferred by our prediction model.

It is intuitive for the user to fix a non-converging program by altering program code that corresponds to the top features. For program **B**, after the topmost motif indicates the location that contributes to non-convergence, removing the negative sign would allow program to converge. After applying the change, the user can use SixthSense to predict again, or even iteratively search for a good fix. This iterative debugging would be much faster than running through the full compilation and execution with Stan. At the same time, SixthSense can provide more directed warning messages.

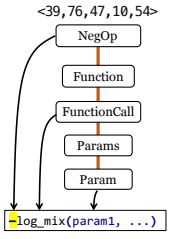


Fig. 2: Topmost motif in program **B**

3 Overview

Figure 3 shows the architecture of SixthSense. We next describe each of its components.

Feature Computation. SixthSense's features can be broadly divided into three major groups: (1) automatically-selected AST (Abstract Syntax Tree) based features - motifs - which represent fragments of the AST; (2) Data Features, and (3) runtime features of the inference algorithm. We present our feature selection and summarization in Section 4.

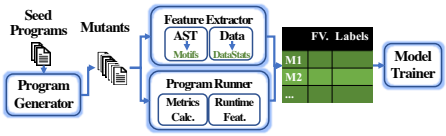


Fig. 3: SixthSense Training Workflow

Program Generation. The generator uses the input set of seed programs to generate a batch of mutants. We use two sets of transformations to mutate the program: (1) *Expansive Mutations* produce more complex models compared to the original ones (e.g., add a new parameter), and (2) *Reducing Mutations* simplify the models by simplifying arithmetic expressions, removing conditional statements, etc. Our adaptive mutator uses nearest neighbor algorithms to efficiently explore the feature space of the programs. We explain the mutations and the algorithms in Section 5.

Program Runner. It runs each generated mutant and collects several statistics such as samples from MCMC iterations and runtimes.

Metric Calculator. Typically, the MCMC algorithms provide samples for each parameter from the posterior distribution. The metric calculator computes the convergence for each parameter using the samples from the posterior.

Model Trainer. Using the syntax, data and runtime features and metrics computed by the previous components, the Model Trainer builds a machine learning model for predicting the behavior of probabilistic models for the given inference algorithm. Here, we used Random Forest Classifier.

We build models to predict, for given metric thresholds, (1) Convergence of the models using static features of model and data, (2) Convergence of the models using static features and run-time diagnostics from initial phases of sampling, and (3) Predict iteration count for which the model will converge.

Deploying the Trained Model. Once the trainer produces the model, we can use it to predict the convergence of new programs. For a given program and its dataset, SixthSense runs the feature extractor, runs it through the predictor and outputs the convergence label. It also reports on the features that contributed most to the prediction, and relates them back to the source code.

4 Learning Program Features

We present the description of the programs and SixthSense's approach for collecting code, data, and runtime.

Probabilistic Programs Syntax. A probabilistic program is an imperative program with additional constructs for sampling from distributions, conditioning the model on observed data values, and one or more queries for either the posterior distribution or expected value of a parameter. In this work, we use a subset of syntax of Storm-IR [19] for representing probabilistic program, as shown in Figure 4.

```
x    ∈ Vars
c    ∈ Consts∪{−∞,∞}
aop  ∈ {+,−,∗,/,}
bop  ∈ {=,>,...}
Dist ∈ {Normal,Uniform,...}
ID   ∈ String
```

```
Type    ::= Int | Float
Decl    ::= x : Type | x : [c+]
Expr    ::= c | x | Expr aop Expr | Expr bop Expr
Stmt    ::= x = Expr | Decl | observe(Dist(Expr+),x)
            | x ∼ Dist(Expr+) | for x ∈ 1..n;{Stmt*}
            | if (Expr) then Stmt* else Stmt*
Query   ::= posterior(x) | expectation(x)
Program ::= Stmt* Query*
```

Fig. 4: Syntax of Storm-IR [19]

Representing Program Paths. To understand the causes of non-convergence and for better debuggability, we select a representation that is easy to train and interpret. Existing approaches Code2Vec/Code2Seq [4, 5] aim to predict variable names through natural-language semantics, and they encode the path between any two terminal nodes in the Abstract Syntax Tree (AST). Instead, we encode the sequences of AST nodes with limited length to pinpoint the semantic issues. We formalize our notions:

Definition 1. (Abstract Syntax Tree) Similar to [5], we define an AST for a program P as a tuple $<N,T,X,s,\delta,\phi,\psi>$. N is a set of non-terminal nodes, T is the set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \rightarrow (N \cup T)*$ is a function which maps each non terminal node to a list of its children, $\phi : T \rightarrow X$ is a function which maps each terminal node to some value, and $\psi : N \rightarrow \mathbb{N}$ maps each non-terminal node to a unique natural number.

Definition 2. (AST Path) An AST path is a path between the nodes in the AST, which starts from one non-terminal node and ends at another non-terminal node, passing through the ancestors of each node at each step.

Definition 3. (Motifs) A Motif encodes an AST path from a node passing through the ancestors of length up to L. For a given AST Path : $\langle N_1, N_2, ..., N_L \rangle$, where $N_i \in \delta(N_{i+1})$, $\forall i \in 1..L-1$, we can define the motif as the list: $\langle I_1, I_2, ..., I_L \rangle$, where $I_m = \psi(N_m), \forall m \in 1..L$.

4.1 Extracting Features from Programs

Motivation. Two major challenges in efficiently encoding the motifs in a feature vector include (1) the large numbers of different paths that a program may have, and (2) the variability of length between different paths. A general approach to solve both problems is to design a flexible scheme for *dimensionality reduction*, which encodes the rich structures, like our motifs as a smaller set of program properties.

We rest our approach on two observations. First, despite a huge number of possible syntactic paths, *similar motifs repeat often in a single program and across multiple programs*. Therefore, we need to think only about the subsets of all possible paths that appear in the corpus of programs. Second, the variability between motifs is often local, and *many similar (though not-identical) motifs may lead to the same program behaviors*. Therefore, instead of encoding each motif in the feature vector independently, we can group similar motifs and encode only the group.

To reduce the dimensionality of available paths and group together similar motifs, we use *Random Discretized Projections (RDP)* [8], hashing technique for reducing dimensionality of large feature vectors. It is well-known in data mining, not been used for big-code representation. RDP calculates hash values that are used to group similar items into the same buckets with high probability based on a similarity metric (e.g. cosine similarity). The hash value represents the motif-group in the feature vector.

Extracting Features from Individual Programs. Line 5-9 in Algorithm 1 describes the procedure to extract motifs from a program. We iterate over the nodes in the AST and for each node, to extract a sequence of nodes by visiting the parent nodes up to level L, using the function *GetMotifAt* (line 6), which we define recursively as $GetMotifAt(N,L) = N :: GetMotifAt(parent(N),L-1)$ and base cases $GetMotifAt(\varnothing,L) = \varnothing$ and $GetMotifAt(N,0) = \varnothing$.

The function *SimilarityHash* (line 7) computes a hash key of each motif using the Random Discretized Projections (RDP) [8]. If the size of the motif is smaller than L (e.g., because the node does not have sufficient number of parents), *PadRight* pads the motif to the maximum size with unused elements. We increase the count for the hash each time a similar motif has obtained the similar hash function (line 8). The RDP has a flexible number of projections and the size of bins. These parameters can be tuned to make similarity more or less fine-grained. They also control, indirectly, the size of the feature vector, the construction of which we describe next.

Calculating Feature Vectors. Given a batch of programs *Batch* and the motif length L, we iterate over the batch to extract the motifs for each program (line 5-9), as described in the paragraph above. Then, to store all the motifs, we first use *InitFVTable* to create a feature vector table F whose column length is equal to the number of programs and the row length is equal to the number of unique motifs (features) across all programs in the batch (line 10). Each row of F is the feature vector of the program *prog*, and each cell stores the count of a motif m in *prog* (line 11-13). *index* maps between the motif hash code and the column index in F. Finally, we output all the feature vectors.

Algorithm 1 Compute Feature Vectors

```
   Input: Batch of Programs Batch, Motif depth L
   Output: Feature Vectors F
 1: procedure CalculateFeatures
 2:     batchMotifs ← ∅
 3:     for prog ∈ Batch do
 4:         progMotifCount = {0,...,0}
 5:         for node ∈ nodes(AST) do
 6:             m ← GetMotifAt(node,L)
 7:             h ← SimilarityHash(PadRight(m,L))
 8:             progMotifCount[h] ← progMotifCount[h]+1
 9:         batchMotifs(prog) ← progMotifCount
10:     F,index ← InitFVTable(Batch,batchMotifs)
11:     for prog ∈ Batch do
12:         for m ∈ batchMotifs(prog) do
13:             F[prog][index(m)] ← batchMotifs(prog)[m]
14: return F
```

4.2 Data Features

The nature of the data-set may determine the performance of the probabilistic model when run using an inference algorithm. For instance, in absence of sufficient data, the choice of prior distributions become very important. Similarly, a strong prior with very small variance is unlikely to converge to the correct results in such a scenario [2]. SixthSense computes data metrics like *sparsity* (number of non-zero elements), *auto-correlation* (correlation between values of a time series), *skewness* (asymmetry of the distribution), maximum/minimum variances of the model's prior distributions, and several others for observed and predictor data variables.

4.3 Runtime Features

For inference algorithms like MCMC, diagnostics from the early stages (warmup) of sampling can often indicate the presence or absence of problems with the model and

associated data. Such diagnostics can help in discovering problems earlier so that the users can update their model for more efficient performance. Unfortunately, they are not predictive in nature: manually observing the raw values may not provide a good intuition about the program execution. However, our prediction engine can infer useful information from them.

To validate this intuition, we collect several runtime features from MCMC chains during the early stages of warmup iterations. These features are algorithm specific. For NUTS, they include *posterior log density* (log probability that the data is produced by the model using current set of the parameters), *tree depth*, *divergence* of the simulated trajectory, *acceptance rate* of the generated sample, *step-size* (the distance between consecutive samples), *leapfrog steps*, and *energy* estimate of Hamiltonian.

5 Program Generation for Training Set Augmentation

In this section, we describe our approach of generating mutant programs from a corpus of seed programs. To produce mutants from the original seed programs, we define two kinds of transformations – for code and data.

5.1 Code Mutations

Our Code Mutations can be broadly classified into two sets: (1) *expansive mutations*, which make more complicated models from the original one, and (2) *reducing mutations*, which reduce the complexity of the models.

Expansive Mutations. These include *Auxiliary Parameter Creator* which converts a distribution argument to a parameter in the program, *Conjugate Replacer* which replaces prior distributions with distributions conjugate [46] to the likelihood when possible, *Dimension Expander* which expands the dimension of a scalar parameter to match the data dimension, *Constant Replacer* lifts a constant in the program to a parameter with an appropriate distribution, and *Data to Parameter Transformer* randomly replaces a real valued data array with a parameter with the same dimension.

Reducing Mutations. The transformations include *Arithmetic Simplifier*, which replaces arithmetic expressions with either of the operands or changes the arithmetic operation, *Conditional Eliminator* which replaces conditional statements with either of the branches *Distribution Simplifier* which replaces complex distributions like *Laplace*, *Weibull* with common distributions like *Normal* or *Uniform*, *Math-Function Call Eliminator* which replaces common math functions like *log*, *exp*, etc. with constants. These transformations have been previous used by [19] for testing PP systems.

5.2 Data Mutations

Apart from source code transformations, we also added several data transformations. Such transformations help in changing the distribution of values in the data set, which could produce challenging scenarios for the probabilistic model or inference algorithm to work with. The data mutations include scaling by a constant, adding arbitrary noise, Box-Cox transformation [49], scaling to new mean and standard deviation, cube root transform, and random replacement of values with values from the same data set.

5.3 Adaptive Algorithm for Mutant Generation

To generate programs with different runtime behaviors, it is important to explore programs with diverse semantic and syntactic features. Our mutation algorithm randomly applies several mutations to the original program. However, to diversify the generated mutants it uses a nearest neighbor based algorithm (Locality Sensitive Hashing [12]), which only selects a representative set of mutants in multiple rounds.

Algorithm 2 Selecting Mutants

```
Input: Seed Programs S, Programs M, BatchSize B
Output: Program Set progs
procedure SelectMutants
    rdp ← InitializeLSH()
    progs ← ∅
    while |progs| < M do
        for s ∈ S do
            seed ← chooseSeed(s,progs)
            p ← GeneratePrograms(seed,B)
            for k ∈ p do
                fv ← feature_vector(k)
                if rdp.neighbours(fv) < 1 then
                    rdp.store_vector(k)
                    progs ← progs.append(k)
return progs
```

Algorithm 3 Generating Mutants

```
Input: Seed program S, Programs M, Max Changes C
Output: Program Set progs
procedure GeneratePrograms
    progs ← ∅
    i ← 0
    while i < M do
        p' ← p
        for t ∈ {1..C} do
            m ← chooseMutation()
            p' ← m.mutate(p')
        if p' ≠ p then
            progs ← progs.append(p')
        i ← i+1
return progs
```

Algorithm 2 presents the mutant selection algorithm. The inputs for the algorithm are seed programs S, total number of programs to generate M, and the number of programs to generate in each batch B from each seed program. The algorithm returns the selected mutant programs set *progs* as output. First, we initialize the LSH (Locality Sensitive Hashing) engine. We used four *Random Discrete Projections* hash functions. Next, in each round, we first choose a seed program using the *chooseSeed* function. The *chooseSeed* function randomly chooses among the original seed program s and the mutants generated (in *progs*) from it in earlier rounds. Next, we generate a new batch of programs of size B using *generatePrograms*.

For each new generated program k, we compute its *feature_vector* and number of neighbors among the already generated programs. We select the program only if it has no neighbors in the already selected set of programs. Finally, the algorithm returns the selected set of programs once it has generated the target M programs. The *generatePrograms* algorithm (Algorithm 3) generates M mutants for a seed program S. For each program, in each iteration, it applies a set of randomly chosen mutations and adds it to the set of new programs. Finally, it returns the set of new programs to the caller. Using this algorithm, we obtained a diverse set of probabilistic programs with a good balance of converging/non-converging behavior.

6 Methodology

We present the methodology for collecting seed probabilistic models and the program features and metrics we compute.

Seed Probabilistic Models. We collected a corpus of probabilistic models from the most comprehensive open-source repository of Stan Models [52] [2]. Out of total 505 models, we selected the three most common categories: Regression (120 models), Time-Series (23), and Mixture Models (23, augmented with 3 from [33]). The models come with their datasets.

Inference Engine and Sampling. NUTS, the default inference engine of Stan [24]. We executed all programs using 4 MCMC chains with 1000 iterations each for warmup phase and sampling. This configuration is default for Stan. We also checked the eventual convergence by running the programs for many more iterations. We used 100,000 as the maximum number (the convergence metrics do not change significantly even for 10^6 iterations for the seed models).

Feature Extraction. We used a Python based implementation of Randomized Discretized Projection [1]. We set its hyper-parameters P=5 and bin-width B=5, which worked well to reduce the dimensionality of the vector space.

Random Forests. We used Random Forests Classifier from Scikit-Learn package in Python for training. We use 5-fold cross validation for training. We extract top features using TreeInterpreter [56].

Execution Setup. We performed the mutant generation and feature computation on an Intel Xeon 3.6 GHz machine with 6 cores and 32 GB RAM. We used Azure Batch Scheduling Service to run all the programs and metrics computations. We capped the MCMC execution under 240 minutes.

6.1 Baselines, Metrics, and Classification

Baselines. We compare SixthSense to three baselines: The first, Code2Vec [5], and the second, Code2Seq [4], are state-of-the-art predictors based on Deep Neural Networks for big-code. They were originally used to predict function names from code. We adapted these systems to do classification for each threshold of convergence, by extracting *path contexts* (subsets of paths similar to our motifs) form the code. Finally, the third baseline, the majority classifier assigns the most likely label during the training to all the predicted programs. It indicates the prediction 'hardness' when the training set is disbalanced.

Metrics. We used a common metric for measuring convergence, called the *Gelman-Rubin* ($\hat{R}$) [25] diagnostic. Ideally, the value of this metric should be close to 1.0. If the observed value of $\hat{R}$ is e.g., 1.05 it is considered as good indication of convergence. The larger values, e.g., 1.5 and greater, are considered as weaker evidences for convergence. Given the threshold, we assign the label *True* to a program if the metric value is within the threshold and *False* otherwise.

[2] The number of publicly available probabilistic programs in public sources is low, compared to conventional languages. This is in part due to the novelty of these languages and expertise required to create and interpret those models. As a further challenge, Stan programs require the corresponding data set of sufficient size, which many Stan programs on Github do not have. Finally, most of publicly available programs are tuned to converge to their available data-sets.

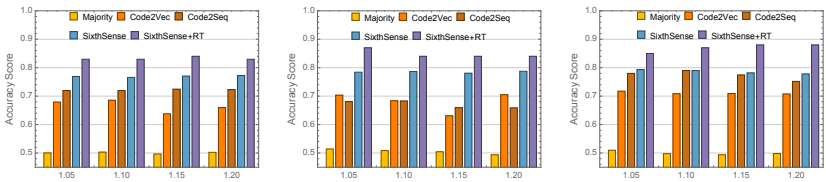
6.2 Evaluation Experimental Setup

Training and Test Sets. We generate a corpus of mutants programs for each seed program using the approach discussed in Section 5.3. We create a *test-train split* for every seed program in the following way: (1) *Test set* consists of a single seed program and *all* its mutants; (2) *Training set* contains all other seeds and mutants. Thus, the training is not aware of any mutants of the test seed program. For each such split, we train a classifier using the training set and evaluate its performance (using the metrics below) on the test set. With this strategy we obtain metrics for each split (each representing one seed program and its mutants). Finally, we compute the *average* performance across the splits.

Training a predictor by leaving out each model and its mutants in test set allows us to stress-test the model predictor. We choose this evaluation strategy because the number of original seed programs in each class is low compared to conventional big-code data-sets. Every seed probabilistic program represents a different statistical model and using this strategy helps us evaluate the sensitivity of the classifiers for each such model.

Classification Scores. We used Precision, Recall, Accuracy, and AUC [21] to evaluate the performance of the learned classifier. They range between 0 and 1 (higher better). We use the same metric for all the baselines.

Accuracy and AUC are adequate metrics for our scenario: Since we perform training by creating a test-train split for every seed program and its mutants (Section 6.2), in some cases the test-set can become imbalanced, e.g. no or few positive labels/no true and false positives or extremely different sizes of the splits.



(a) Regression models (b) Mixture models (c) TimeSeries models

Fig. 5: SixthSense Prediction Accuracy for Convergence (Measured Using Gelman-Rubin Diagnostic)

7 Evaluation

7.1 Predicting Convergence of Inference

Figure 5 presents the prediction scores for SixthSense when predicting convergence of MCMC algorithms (NUTS in this case). The Y-axis shows the accuracy scores for each prediction model (higher is better). The X-axis shows the four thresholds (1.05-1.2) of the convergence metric, Gelman-Rubin diagnostic, that we considered in our evaluation. We chose this range to test how general the prediction can be as the

Table 1: Precision (**P**) and recall (**R**) ($\hat{R}$=1.05)

Class	6s-AST		Code2Vec		Code2Seq	
	P	R	P	R	P	R
Regression	0.71	0.71	0.63	0.69	0.66	0.72
Mixture	0.77	0.74	0.67	0.67	0.67	0.72
Time Series	0.79	0.75	0.69	0.74	0.74	0.77

Table 2: AUC scores ($\hat{R}$=1.05)

Class	6s	6s+RT	Code2Vec
Regression	0.82	0.88	0.73
Mixture	0.84	0.90	0.74
Time Series	0.86	0.89	0.79

individual program labels change. For each threshold, we plot the accuracy scores of our prediction model (SixthSense) together with Code2Vec, Code2Seq and a Majority Label Classifier, as vertical bars in different colors. We evaluated the trained model on a held-out test set (see Section 6.2).

Comparison with Code2Vec/Code2Seq. Figure 5 shows that SixthSense, with solely AST motifs is better than Code2Vec and Code2Seq (see also the ablation study in Section 8). The results show that SixthSense's learned classifiers have an accuracy score close to 0.8. These prediction rates are already useful for the user because it helps them avoid wasting time for compiling and running programs which would likely not converge. Our training algorithm is able to learn classifiers that generalize well across different thresholds.

For Regression and Mixture models, SixthSense has consistently better accuracy than the other approaches across all thresholds. For the tightest convergence bound $\hat{R} = 1.05$, its accuracy is by 5 percentage points higher than the alternatives for Regression, and 8 percentage points higher for Mixture. For TimeSeries models, the accuracy scores of SixthSense is by 1 percentage point higher than Code2Seq.

Table 1 presents the precision and recall for $\hat{R} = 1.05$. SixthSense exhibits consistently higher precision over Code2Vec (8 to 10 percentage points) and Code2Seq (5 to 10 percentage points). SixthSense also has higher recall than Code2Vec (1 to 7 percentage points), while the recalls of SixthSense and Code2Seq are comparable (within 2 percentage points). Recall that the precision/recall are averaged over those for different splits and can be more sensitive to small and unbalanced splits.

Table 2 shows the AUC scores for SixthSense, SixthSense with runtime features and Code2Vec. Code2Seq does not provide its probability of predictions, which prevents us from computing its AUC score. The results show that SixthSense improves in AUC score over Code2Vec for all classes.

The prediction accuracy, prediction, and recall from Tables 1 and 2 persist for higher thresholds of $\hat{R}$.

Comparison to Majority Label Classifier. Figure 5 shows the comparison of SixthSense to a naive Majority Label Classifier, which has the classification accuracy of 0.5. It indicates the significant level of improvement of SixthSense over the uninformed random choice.

Predicting with Warm-up Runtime Features. Figure 5 presents the impact of SixthSense's AST features augmented with runtime features (Section 4.3) sampled from the first 200 iterations of the warmup stage (at this point Stan still does not issue warnings for our programs). Recall, the results of these iterations are dropped by the inference algorithm, as in this phase the mixing of the MCMC chains has just begun. However they can be useful in addition to code features: they help improve the prediction by further 6 percentage points for Regression and Timeseries, and 8 percentage points for Mixture models ($\hat{R}$=1.05).Table 2 also shows the improvement

in AUC of both AST and Run-Time features over the AST-only version of SixthSense. However, note that collecting run-time features still requires compiling the program and starting its execution. While this time differs among the systems and datasets, it may be non-trivial, as is the case for Stan (e.g. around 30 seconds for compilation). This time may be an important factor when deciding to use a runtime-predictor for different PP systems. We also present a feature ablation study in Section 8.

7.2 Debugging Non-Converging Programs

When SixthSense's learned model predicts that a model will not converge, two natural follow-ups are (1) ask which part of the program is likely culprit for non-convergence and (2) how many iterations would be sufficient to run the model to converge, if it converges.

Debugging Approach. We interpret the outcomes SixthSense predicts, and leverage the AST features and the random forests to help pinpoint which part of the program leads to non-convergence.

To obtain the set of programs, we randomly selected 40 probabilistic programs from our *test sets*, equally across the three model classes, which SixthSense correctly identified as non-converging for 1000 iterations. For each program, we obtained the most important features from the learned random forest. We selected *top-5 features* (motifs) and inspected the model to identify whether the parts of the motifs contains the culprit of non-convergence. The top-5 features typically only cover 5% of all the motifs, which means SixthSense points to a relatively small scope to debug.

We make up to two manual updates to each model by making changes only to the AST elements identified by the motifs or the referenced observed data. These changes represent simple semantic modifications that a user of probabilistic program might make as they explore various possible models for their data. We simulate a *try and check* interactive search with these localized transformations. For instance, SixthSense identified a constant array in a regression equation as one of the top motifs. Converting that constant into a parameter made the model converge. Some of our attempted updates include changing the variance (constant) of a distribution, changing the distribution for a parameter, changing a parameter to a constant, and removing mathematical functions (e,g. *abs, log*) when they are redundant.

After transforming the model, we run inference to see if it converges. We further check if the model become accurate (or correct) after the fix, since non-convergence often causes inaccurate (or wrong) result. For each model, we apply accuracy tests from Bayesian model checking [25, Ch.6]: we compute the mean squared error to compare the new model result to its correct data and also do visual inspection on the result density plot to check if it matches the correct distribution. Multiple student authors inspected the updates and agreed that these changes followed the protocol described above.

Results. Table 3 presents the results for this debugging application. Column 1 (**Class**) presents the classes of randomly sampled models. Column 2 (**#Models**) presents the number of mutant models we randomly selected from each class. Column 3 (**6s Upd.**) presents the number of programs that we manually updated to converge using the method above. Column 4 (**Stan Warn.**) presents the number of programs

which Stan issued a warning during sampling. Column 5 (**Stan Upd.**) presents the number of programs for which Stan's warnings helped update the program to converge.

Table 3: Debugging Non-Converging Models

Class	#Models	6s Upd.	Stan Warn.	Stan Upd.
Regression	14	11	4	2
Mixture	13	9	4	1
TimeSeries	13	9	4	2

Overall, we were able to identify the problem and let 29 updated models converge out of 40 models. Specifically, we corrected 16 models by replacing a parameter indicated by SixthSense with a constant; corrected 6 by simplifying mathematical functions, 3 by changing constants in distributions, 2 by converting constants to parameters, and 2 by changing distributions for parameters. All the code elements we changed were pointed by top three motifs SixthSense returned. For 11 models that we were not able to update, we believe that the model correction would require more complex changes than those we specified in setup above.

Out of 29 updated, now converging models, we ran SixthSense again. It correctly predicted that 21 will converge (with 8 from Regression, 8 from TimeSeries and 5 from Mixture); this is, interestingly, close to the prediction rates from Section 7.1. This illustrates that SixthSense can be useful in the iterative debugging loop.

These results demonstrate the advantage of interpretability SixthSense's learned model. Using *motifs* from the AST as features and a simple learning model (random forests) helps the user easily identify key program components which affect the runtime behavior of a probabilistic model. In comparison, identifying such important features is hard for other complex neural network-based models and might require more low-level handling of the learned model. In particular, Code2Vec and Code2Seq do not provide a way to interpret how their prediction worked.

Comparison to Stan's runtime warnings. Compared to Stan's runtime warnings, SixthSense motifs reveal more fine-grained patterns that hinder convergence. For most of the non-converging models (29 out of the 40 in this experiment), Stan did not issue a warning (beyond the low $\hat{R}$ value at the end of inference) The 12 warnings issued by Stan only have regards to function domains. Seven out of 12 were not related to non-convergence. For instance, one program returns "*Warning: normal_lpdf: Scale parameter is -0.0799029, but must be* >0." Changing the scale parameter limits does not help. Instead SixthSense identifies the fix that is not at this location.

The remaining 5 Stan runs indicate non-convergence and can help with updating the model. However, they were not as helpful in locating the causes as SixthSense. One example where both SixthSense and Stan indicated problem is in the program with the expression $normal(exp(w0) + sqrt(abs(w1)) * x1 + w2 * x2, s)$. Stan warned about the overflow in the first argument of *normal*, disregarding its sub-expressions. SixthSense traced the problem to the *sqrt* and *abs* sub-expressions that indeed helped fix the non-convergence, by simplifying the function expressions.

8 Sensitivity Analysis

We present various sensitivity analyses of SixthSense to justify our design choices.

Table 4: Ablation Study ($\hat{R}$=1.05)

Class	A	A+D	A+RT	A+D+RT
Regression	0.77	0.77	0.83	0.83
Mixture	0.78	0.78	0.87	0.87
TimeSeries	0.79	0.79	0.84	0.85

Table 5: Training w. Noisy Labels ($\hat{R}$=1.05)

Label Flip Pr.	1%		3%		5%	
Model Class	R	B	R	B	R	B
Regression	0.765	0.760	0.763	0.765	0.760	0.764
Mixture	0.772	0.784	0.774	0.782	0.783	0.785
TimeSeries	0.786	0.789	0.794	0.781	0.781	0.788

8.1 Feature Ablation Study

Table 4 shows the Accuracy score for convergence predictions when trained with different combinations of feature groups (AST features, AST and data features, and all features). Runtime features are from 200 warmup iterations. The AST features (motifs) alone contribute a major portion to the Accuracy scores in all cases. Data features do not have much impact on these models. Runtime features, after a certain number of iterations further improve prediction (they are in fact a strong predictor, but do not establish a relation with the program code). Obtaining runtime statistics comes at a cost of compiling and running the program. This cost is often over 30 seconds for Stan.

Impact of the noisy labels on the prediction. To evaluate the robustness of our prediction, we perturb the class labels in the training set with different noise levels. We use the version of SixthSense, which applies Rank Pruning [41]. Table 5 shows the Accuracy scores for the different model classes for several noise levels (1-5%). For each noise level, *Robust* column shows the scores when trained using the Rank pruning algorithm and *Basic* column shows the scores for baseline SixthSense. Even in the presence of significant training noise, our learning approach maintains high Accuracy scores. For instance, the performance of Mixture Models remains almost constant (close to 78%) even when 5% labels are wrong.

Other sensitivity studies. We also performed other sensitivity studies on the features and generated programs. First, we looked at different motif sizes. For three motif sizes (5, 10, 20) on the threshold $\hat{R}$=1.05, we do not see a significant increase in the Accuracy score. This reflects that even smaller motifs obtained from probabilistic programs can be very effective for predicting their runtime behavior. Therefore, we used Motif size of 5 in all our experiments.

We then removed overlapped motifs, which resulted in the reduction of the Accuracy scores (by 2 to 5 percentage points). Other experiments, such as different LSH configurations to remove syntactically similar programs from the training set did not show substantial deviation from the reported scores.

9 Related Work

Probabilistic Programming. Probabilistic programming languages (PPLs) and their underlying inference systems have recently gained significant interest from research and industry [9, 10, 26, 27, 29, 36, 38, 45, 55, 58]. Tyically, PPLs (e.g., Stan) only provide simple runtime diagnostics and timing information as they run. In contrast, SixthSense is a predictive data-driven approach that complements these efforts.

The prior debugging approach for PPLs [39] requires augmenting Bayesian network representation with additional labels and requires extending the inference algorithm.

However, its applicability is limited since state-of-the-art PP systems typically do not use Bayesian network representation. Our approach learns program features useful for debugging without modifications to the inference algorithm. Existing tools [15, 19] find lower-level implementation bugs in probabilistic programming systems.

Several recent approaches have explored the nature of regression tests in probabilistic and machine learning applications such as the causes and fixes for flaky tests [17, 18], usage of seeds in tests [14], and speeding up expensive regression tests [16].

Predicting Program Properties from Big-Code. Much attention has recently been devoted to uses of machine learning to analyze and predict various program properties. Notable examples include predicting variable names/types via statistical program models [47], predicting patches [35], summarizing code [3, 31], and API discovery [5, 57]. However, all of these works apply learning on conventional programs (C/Java/Javascript), obtained from massive code repositories. Moreover, many of these approaches predict static program properties (e.g., names/types), rather than execution properties like convergence. While some of these approaches benefit from the natural-language semantics of identifiers [4, 5], we are interested in semantics of the program itself, which are better represented by the sequence of AST nodes.

We also present how to augment the corpus of programs with diverse programs via guided mutation. While our approach bears similarity to data augmentation in machine learning [11, 50, 53], probabilistic programs have complex structure defined by many syntactic (and often semantic) rules.

Predicting Algorithm Performance. Researchers developed machine learning approaches that predict hardness of NP-hard problems (e.g., SAT, SMT, ILP) [7, 32, 34]. These works are complementary and their syntax and semantics are considerably simpler than for probabilistic programs. Researchers also proposed models for performance of other machine learning architectures [13, 20, 30, 43], but their techniques and applications are orthogonal to ours.

10 Conclusion

We presented SixthSense, a novel approach and system, which predicts convergence for probabilistic programs and helps guide the debugging of convergence issues. We show SixthSense is effective in extracting features from probabilistic programs and learning a prediction model. Compared to the state-of-the-art techniques, our results show significant improvement in accuracy.

Acknowledgments

This research was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2008883, USDA NIFA Grant No. NIFA-2024827, a gift from Facebook, a Facebook Graduate Fellowship, and Microsoft Azure Credits. We would also like to thank Prof. Jian Peng for the useful comments on an earlier draft.

References

1. Nearpy (2011), https://github.com/pixelogik/NearPy
2. Prior choice recommendations in stan (2011), https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations
3. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning. pp. 2091–2100 (2016)
4. Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: Generating sequences from structured representations of code. In: International Conference on Learning Representations (2019), https://openreview.net/forum?id=H1gKYo09tX
5. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages **3**(POPL), 40 (2019)
6. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Communications of the ACM **51**(1), 117 (2008)
7. Balunovic, M., Bielik, P., Vechev, M.: Learning to solve smt formulas. In: Advances in Neural Information Processing Systems. pp. 10338–10349 (2018)
8. Bingham, E., Mannila, H.: Random projection in dimensionality reduction: applications to image and text data. In: Proceedings of the international conference on Knowledge discovery and data mining (KDD). ACM (2001)
9. Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M.A., Guo, J., Li, P., Riddell, A.: Stan: A probabilistic programming language. JSTATSOFT **20**(2) (2016)
10. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: FSE (2013)
11. Cubuk, E.D., Zoph, B., Mane, D., Vasudevan, V., Le, Q.V.: Autoaugment: Learning augmentation policies from data. arXiv preprint arXiv:1805.09501 (2018)
12. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the twentieth annual symposium on Computational geometry. pp. 253–262. ACM (2004)
13. Deng, B., Yan, J., Lin, D.: Peephole: Predicting network performance before training. arXiv preprint arXiv:1712.03351 (2017)
14. Dutta, S., Arunachalam, A., Misailovic, S.: To seed or not to seed? an empirical analysis of usage of seeds for testing in machine learning projects. In: ICST (2022)
15. Dutta, S., Legunsen, O., Huang, Z., Misailovic, S.: Testing probabilistic programming systems. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 574–586. ACM (2018)
16. Dutta, S., Selvam, J., Jain, A., Misailovic, S.: Tera: Optimizing stochastic regression tests in machine learning projects. In: ISSTA (2021)
17. Dutta, S., Shi, A., Choudhary, R., Zhang, Z., Jain, A., Misailovic, S.: Detecting flaky tests in probabilistic and machine learning applications. In: ISSTA (2020)
18. Dutta, S., Shi, A., Misailovic, S.: Flex: fixing flaky tests in machine learning projects by updating assertion bounds. In: FSE (2021)
19. Dutta, S., Zhang, W., Huang, Z., Misailovic, S.: Storm: Program reduction for testing and debugging probabilistic programming systems. In: FSE (2019)
20. Dutta, S., Joshi, G., Ghosh, S., Dube, P., Nagpurkar, P.: Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. arXiv preprint arXiv:1803.01113 (2018)

21. Fawcett, T.: An introduction to roc analysis. Pattern recognition letters **27**(8), 861–874 (2006)
22. Flaxman, S., Mishra, S., Gandy, A., Unwin, H.J.T., Mellan, T.A., Coupland, H., Whittaker, C., Zhu, H., Berah, T., Eaton, J.W., et al.: Estimating the effects of non-pharmaceutical interventions on covid-19 in europe. Nature pp. 1–5 (2020)
23. Gelman, A.: Stan being used to study and fight coronavirus (2020), https://discourse.mc-stan.org/t/stan-being-used-to-study-and-fight-coronavirus/14296, Stan Forums
24. Gelman, A., Lee, D., Guo, J.: Stan a probabilistic programming language for bayesian inference and optimization. Journal of Educational and Behavioral Statistics (2015)
25. Gelman, A., Stern, H.S., Carlin, J.B., Dunson, D.B., Vehtari, A., Rubin, D.B.: Bayesian data analysis. Chapman and Hall/CRC (2013)
26. Goodman, N., Mansinghka, V., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. arXiv preprint arXiv:1206.3255 (2012)
27. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages (2014)
28. Hoffman, M.D., Gelman, A.: The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. Journal of Machine Learning Research **15**(1), 1593–1623 (2014)
29. Huang, Z., Dutta, S., Misailovic, S.: Aqua: Automated quantized inference for probabilistic programs. In: International Symposium on Automated Technology for Verification and Analysis. pp. 229–246. Springer (2021)
30. Istrate, R., Scheidegger, F., Mariani, G., Nikolopoulos, D., Bekas, C., Malossi, A.C.I.: Tapas: Train-less accuracy predictor for architecture search. arXiv preprint arXiv:1806.00250 (2018)
31. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 2073–2083 (2016)
32. Khalil, E.B., Le Bodic, P., Song, L., Nemhauser, G., Dilkina, B.: Learning to branch in mixed integer programming. In: Thirtieth AAAI Conference on Artificial Intelligence (2016)
33. Inference case studies in knitr (2019), https://github.com/betanalpha/knitr_case_studies
34. Leyton-Brown, K., Hoos, H.H., Hutter, F., Xu, L.: Understanding the empirical hardness of np-complete problems. Communications of the ACM **57**(5), 98–107 (2014)
35. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: ACM SIGPLAN Notices. vol. 51, pp. 298–312. ACM (2016)
36. Mansinghka, V., Selsam, D., Perov, Y.: Venture: a higher-order probabilistic programming platform with programmable inference. arXiv preprint 1404.0099 (2014)
37. Mendis, C., Renda, A., Amarasinghe, S., Carbin, M.: Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In: ICML (2019)
38. Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., Bronskill, J.: Infer.NET 2.5 (2013), microsoft Research Cambridge. http://research.microsoft.com/infernet
39. Nandi, C., Grossman, D., Sampson, A., Mytkowicz, T., McKinley, K.S.: Debugging probabilistic programs. In: MAPL (2017)
40. Neal, R.M.: An improved acceptance procedure for the hybrid monte carlo algorithm. Journal of Computational Physics **111**(1), 194–203 (1994)
41. Northcutt, C.G., Wu, T., Chuang, I.L.: Learning with confident examples: Rank pruning for robust classification with noisy labels. In: Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence. UAI'17, AUAI Press (2017), http://auai.org/uai2017/proceedings/papers/35.pdf

42. Obermeyer, F.: Deep probabilistic programming with pyro (2020), https://www.broadinstitute.org/talks/deep-probabilistic-programming-pyro, models, Inference, and Algorithms
43. Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R.: sk_p: a neural program corrector for moocs. In: Companion Proceedings of the 2016 OOPSLA. pp. 39–40. ACM (2016)
44. Modeling censored time-to-event data using pyro (2019), https://eng.uber.com/modeling-censored-time-to-event-data-using-pyro/
45. Pyro (2018), http://pyro.ai
46. Raiffa, H., Schlaifer, R.: Applied statistical decision theory (1961)
47. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from big code. In: ACM SIGPLAN Notices. vol. 50, pp. 111–124. ACM (2015)
48. Robert, C., Casella, G.: Monte Carlo statistical methods. Springer Science & Business Media (2013)
49. Sakia, R.: The box-cox transformation technique: a review. Journal of the Royal Statistical Society: Series D (The Statistician) **41**(2), 169–178 (1992)
50. Simard, P.Y., Steinkraus, D., Platt, J.C.: Best practices for convolutional neural networks applied to visual document analysis. In: Icdar. vol. 3 (2003)
51. Stan. using target += syntax (2016), https://stackoverflow.com/questions/40289457/stan-using-target-syntax
52. Stan Example Models (2018), https://github.com/stan-dev/example-models
53. Taylor, L., Nitschke, G.: Improving deep learning using generic data augmentation. arXiv preprint arXiv:1708.06020 (2017)
54. Tehrani, N.K., Arora, N.S., Noursi, D., Tingley, M., Torabi, N., Lippert, E.: Bean machine: A declarative probabilistic programming language for efficient programmable inference. In: PGM (2020)
55. Tran, D., Kucukelbir, A., Dieng, A.B., Rudolph, M., Liang, D., Blei, D.M.: Edward: A library for probabilistic modeling, inference, and criticism. arXiv (2016)
56. Tree interpreter package (2020), https://github.com/andosa/treeinterpreter
57. Wang, K., Su, Z.: Learning blended, precise semantic program embeddings. ArXiv, vol. abs/1907.02136 (2019)
58. Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: AISTATS (2014)

Finding Semantic Bugs Fast*

Lukas Grätz(✉), Reiner Hähnle, and Richard Bubel

Technical University of Darmstadt, Darmstadt, Germany
{graetz,haehnle,bubel}@cs.tu-darmstadt.de

Abstract. Finding semantic bugs in code is difficult and requires precious expert time. Lacking comprehensive formal specifications, deductive verification is not an option. We propose an incremental specification procedure: With the help of automatic verification tools, a domain expert is guided through program runs and source code locations. The expert validates a run at certain locations and creates lightweight annotations. Formal methods training is not required. We demonstrate by example that this approach is capable to quickly detect different kinds of semantic bugs. We position our approach in the middle ground between fully-fledged deductive verification and bug finding without semantic guidance.

1 Introduction

The main obstacle against using program verification tools for bug finding is not their efficiency, but a lack of meaningful formal specifications that capture the intended semantics of a given program [2,9]. This is unfortunate, because *semantic bugs* are dominant over memory-related bugs [15], but cannot be found by existing bug finding approaches [1,3,6,7,18], which look for syntactic patterns or generic errors (such as uncaught exceptions, memory faults, etc.).

A notorious, relatively recent example was an alleged error in a software used in the UK to send mammography invitations to women in a certain age group [11]. Not all letters were sent according to the specification, which would statistically have led to belated diagnosis and possibly premature death of some women. As it turned out, the specification was drawn up in hindsight, *after* the software had been in use for years. To detect the mismatch would have required an expert to look at exactly the right decision points in the code and to compare the implicit assumptions with the specification. This is a challenging task: (i) There might be a vast number of inputs and runs: how to choose the ones that give insight into a possible semantic bug? (ii) Keeping track of implicit assumptions and to check their validity in a given run is tedious, time-consuming, and error-prone.

In this paper we propose a novel approach to help experts finding *semantic bugs*: These are bugs where functional and expected behavior in a domain context deviate, without domain-independent symptoms like abrupt termination or blocking. We address the issues above by dedicated tool and language support.

* Supported by Deutsche Forschungsgemeinschaft (DFG) - Project number 351097374.

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 145–154, 2022.
https://doi.org/10.1007/978-3-030-99429-7_8

The main ingredients are: (1) to render implicit assumptions in the code explicit, traceable, and automatically checkable in the form of lightweight (Boolean Java) specification expressions and a simple labeling mechanism; (2) to use (automatic) deductive verification to guide the expert and to validate assumptions.

We do not aim at a fully automatic process, which we deem futile to detect non-trivial semantic bugs. We also are not interested in complete contract-based specification [13] as typically used in deductive verification [9], which we consider unrealistic in many cases, because of the required effort and the need for training in writing formal specifications. In contrast, the *partial* specifications we aim at are incrementally produced by a software engineer, guided by tool support. The annotations do not cover the full functionality of the analyzed software, but only part of the input space and source code. Therefore, the resulting annotations can stay simple and close to a designer's understanding of the code. Specific training in formal methods is not required.

The flip side of our approach is that we are unable to provide formal guarantees about the absence of bugs. This is in common with other bug-finding technology, such as systematic debugging [18], bug finding tools [1], test case generation [7], or code inspection [6]. On the other hand, all of the mentioned techniques either look for a fixed set of syntactic conditions or assume the presence of a specification, whereas we *guide* the user to come up with *semantically relevant* specification annotations. Consequently, we hope to occupy a sweet spot between fully-fledged deductive verification and bug finding without semantic guidance. In addition, partial specifications can help static verification as well as deductive verification tools. In this NIER paper we sketch our approach and illustrate how it works with simple examples. A robust implementation and full evaluation is envisaged.

2 Validating Program Runs

To explain our approach we use the `min` method shown in Fig. 2. More realistic examples are provided in Section 4. We are given a software system and its source code. The system could be a method, a command line interface, or some other piece of software in which we want to find bugs. We assume the system is already free of memory-related or termination bugs (covered by existing tools)—in particular, for any given input, there is no runtime exception and the system terminates in some final state. We will have a (virtual) code position for all final states: This is where the validation routine will start, see below.

Our validation process is performed by a domain expert, guided by a software assistant. A domain expert knows how the system *should* behave. In general, there is no (formal) specification, so we need an expert, who might be a software tester, code reviewer, or debugging specialist. We assume the expert understands source code and is able to validate the behavior of a given program run.

Program runs are supplied in the beginning. These could have been collected from log files while the system was running. We can often reconstruct a run, if all inputs (and events) are given. We assume the runs cover potential semantic

bugs, the more runs, the better. Although the expert could validate every single run, it is unrealistic to look into all of them—there are far too many.

2.1 Syntax

We illustrate our approach with JAVA source code, but it is applicable to any other (imperative) programming language with suitable tool support.

The expert will stepwise annotate the software under validation with partial specifications. We contribute a simple annotation syntax. Annotations are placed after `//@` or between `/*@` and `*/`, compatible with JML [12]. We do not use full JML, but only a fragment consisting of the *labeled* assertions and assumptions produced by the grammar in Fig. 1. The asserted/assumed expressions `Expr` are also simplified: A domain expert only needs to write side effect-free boolean JAVA expressions—quantifiers or other JML constructs are not required.

Assumptions and assertions are labeled using a prefixed identifier `ALabel` inside `<` and `>`. Labeled assumptions/assertions are only effective when explicitly referred to—*they are not assumed in general*. To make such references, we extend **`assert`** statements with the keyword **`assuming`**. In program ④ of Fig. 2, the assertion labeled `aRes` holds when assuming `aGb`.

The syntax allows an assertion to be **`assuming`** a logical combination of (other) labeled assertions/assumptions. A conjunction of `ALabels` is written as `<l1,l2,...>`. Any *positive* combination of `ALabels` in positive disjunctive normal form (PDNF) is supported: One can build a complex (acyclic) graph of assumptions and assertions depending on each other. We will see that the PDNF is naturally obtained by the validation steps. The PDNF also makes checking/verifying assertions easy, see Sect. 3.

Our annotations bear resemblance to [4,5], where the keyword ***`verified`*** is used instead of **`assuming`**. In contrast to [4,5], labeled assumptions are not *expected* to hold true in every run (in Fig. 2, assumption `aGb` is true for half of the inputs). Hence, in our setting, there is no point in trying to verify assumptions. Instead, to check or verify a claim, we can use a labeled *assertion* with **`assuming <>`**, that is, an assertion without an assumption. For example, we write **`assert`** `x>0` **`assuming <>`** instead of **`assume`** `x>0` to assume *and* verify that variable `x` is greater than 0.

When the system under validation reaches a termination point, we are not asserting any specific claim, however, usually a number of assertions must hold *before* the (virtual) termination point. These are listed in an `OnlyAssuming` clause. If the system boundaries are given by a method (as in Fig. 2), the `OnlyAssuming` declaration is placed before the method—in the example, **`<aRes> or <bRes>`**. This corresponds to a JML *method specification clause* [12].

```
Assumption     := [ <ALabel> ] assume Expr;
Assertion      := [ <ALabel> ] assert Expr [ assuming ALabelPDNF ];
OnlyAssuming   := assuming ALabelPDNF;
ALabelPDNF     := < [ ALabel [, ALabel]* ] > [ or ALabelPDNF ]
```

Fig. 1. Syntax of labeled assumptions/assertions.

```
//@ assuming <aRes>;
int min(int a, int b) {                    (1)

  int m = a;
  if (b < m) m = b;
  //@ <aRes>assert m==a;
  return m;
}
```

```
//@ assuming <aRes>;
int min(int a, int b) {                    (2)
  //@ <aLb>assume a<=b;
  int m = a;
  if (b < m) m = b;
  //@ <aRes>assert m==a assuming <aLb>;
  return m;
}
```

```
//@ assuming <aRes> or <bRes>;             (3)
int min(int a, int b) {

  //@ <aLb>assume a<=b;
  int m = a;
  if (b < m) m = b;
  //@ <bRes>assert m==b;
  //@ <aRes>assert m==a assuming <aLb>;
  return m;
}
```

```
//@ assuming <aRes> or <bRes>;             (4)
int min(int a, int b) {
  //@ <aGb>assume a>=b;
  //@ <aLb>assume a<=b;
  int m = a;
  if (b < m) m = b;
  //@ <bRes>assert m==b assuming <aGb>;
  //@ <aRes>assert m==a assuming <aLb>;
  return m;
}
```

Fig. 2. Simple implementation of `int min(int,int)` with four validation steps.

2.2 Validation Procedure

A *validation assistant* software is intended to guide an expert in validating all program runs without having to scrutinize each single run. In each *validation step*, the expert validates a single program run against an assertion and provides justifications for his or her judgment in form of assumptions. The validation assistant knows a current set of assertions G at certain source code locations and a set of program runs R. In the beginning, G includes merely one implicit, trivial assertion (`assert true`, always satisfied) at the (virtual) termination point of the program. The set G grows after each validation step.

Example 1. We perform the validation procedure for `int min(int,int)`, Fig. 2. In the initial setting (omitted from the figure) we just have the source code without any annotation.

In the first validation step (1), the expert is given a program run with input `a==3`, `b==7`, return value `m==3` and the implicit assertion at the termination point. The expert judges this run to be valid, places `assuming <aRes>` above the method (virtually at the termination point), and then `<aRes>assert m==a` as justification. Verification tools check the implicit (and trivial) `assert true` at the virtual termination point under `<aRes>`. In (2), the expert looks at the same program run, but now he has to give assumptions for the assertion `aRes`. The program run is still valid under the new assertion. The expert now adds an `assuming <aLb>`, and places the assumption `<aLb>assume a<=b` at the start of the method. Tools check assertion `aRes` under `aLb`.

In (3), the expert is given a different program run `a==9`, `b==0`, `m==0`, plus the trivial assertion at the virtual termination point. The expert adds `or <bRes>` in the corresponding `assuming`, and `<bRes>assert m==b` before the method returns. Again, tools verify. In (4), the expert looks at the same program run as before, Since assertion `bRes` is now assuming `aGb`, assumption `aGb` is added.

Tools check and the validation procedure ends at this point successfully with a partially specified program, no bug was found.

We consolidate into the general description of the validation procedure:

Validation Assistant. *Given sets of program runs R and assertions G, the latter containing only the implicit* `assert true` *at termination point. Repeat:*

1. *Choose*[1] *$r \in R$, $g \in G$ such that:*
   (a) *Assertion g is reached and satisfied in r.*
   (b) *If g has an* `assuming` *clause, then none of the disjuncts in its PDNF is already satisfied in r.*
   If there is no such r, g, the validation terminates.
2. *Validation step (see below).*

Validation Step. *Given assertion $g \in G$ and program run $r \in R$:*

1. *The expert judges r under g to be valid.*
   Otherwise, a bug has been found and validation is aborted to fix it.
2. *The expert adds a conjunction* `<`$AL_1, \ldots, AL_n$`>` *as a disjunct in the* `assuming` *of g.*
   (a) *In case of* `assuming <>`*, continue with 4.*
3. *For $1 \leq k \leq n$ such that assertion/assumption* AL_k *does not exist yet, do one of the following:*
   (a) *Expert adds* assumption *labeled with* AL_k*.*
   (b) *Expert adds* assertion *labeled with* AL_k *(initially without* `assuming`*). The new assertion is added to G.*
4. *Verification tools check assertion g under* `<`$AL_1, \ldots, AL_n$`>` *as follows:*
   (a) *All assumptions/assertions* $AL_1, \ldots, AL_n$ *are satisfied in r.*
   (b) *For all $\tilde{r} \in R$: if* $AL_1, \ldots, AL_n$ *are satisfied in $\tilde{r}$ then g is also satisfied.*
   (c) *Attempt to formally verify g assuming* $AL_1, \ldots, AL_n$*, see Sect. 3.*

3 Checking/Verification

Formal verification can be achieved by translating labeled assertions into ordinary JML assertions as described below. The latter can be handled with state-of-the-art verification tools: For example, we can combine static verification and, for each program run separately, run-time assertion checking.

The translation processes one single assertion and its corresponding assumptions at a time and generates a separate verification task for each. For example, take assertion `aRes` from ④ in Fig. 2. There is just one corresponding assumption `aLb`, so we delete all other assumptions in the source file. The resulting code is left with only two annotations: `//@ assume a<=b;` and `//@ assert m==b;` without labels and `assuming`.

The translation of general `ALabelPDNF`s is more complex, for example, assertion `pricePlausible` in Fig. 4, line 19. We must show the assertion holds,

[1] If possible, choose the same run or the same assertion as in a previous iteration. This simplifies the validation step for the expert.

given either `<dscdReg,minPr>` or one of the other two disjuncts. We create three verification tasks. In the first `dscdJun` and `minPr` are JML assumptions and `pricePlausible` becomes a JML assertion (similar for the other two). We obtain `assume discount==0`, `assume movie.getPrice() > 5.60`, as well as the assertion `assert dscdPrice >= 5.00`. Observe that the labeled assertion `dscdJun` is translated into an assumption.

After translation, we can perform checks with any tool that understands JML and Java. We plan to use deductive verification as well as run-time assertion checking tools for every single program run. Depending on the result from the tools, disjuncts in the assuming are highlighted in different colors, as in Figs. 3, 4:

`white` Assertion unchecked
`red` Assertion is violated in some run
`green` Assertion is formally verified
`blue` All runs are fine, but verification only partial due to system limitation
`yellow` All runs are fine, but verification failed and gave a counter example

To demonstrate our approach, we wrote a script to translate annotations of all three examples in this paper [8]. We successfully reproduced the respective assertion verification. We expect that the performance of deductive verification tools is practical, as a side gain from the restricted syntax.

4 Examples

We demonstrate the validation procedure with two examples. Example 3 is less algorithmic and oriented towards real-world software, where an expert familiar with the application domain is essential for validating a software's behavior. Example 2 features an implementation of `int max(int[])`, which produces incorrect results for certain inputs. We will find the bug in two validation steps.

Example 2. Fig. 3 displays an implementation of `int max(int[])`. It produces incorrect results for some inputs. However, we might not detect this immediately, as it gives correct results in the majority of cases. Moreover, it does not throw an exception, except when `a.length==0`. The supplementary material [8] contains a list of 100 random input arrays we used in the experiments. Each array contains between one and four random entries with values in $[0, 100]$ (equally distributed). From that list, 11 of 100 runs give an invalid result.

Initially, the code in Fig. 3 is not annotated. Then we start the validation procedure. The set of initial goals consists of the return point of `max(a)` and we will, as usual, start there. The assistant chooses a program run, for example, corresponding to input `a = {35,38,36,55}`. Now the domain expert performs the first validation step. The expert observes that result `55` is correct. The expert slightly generalizes this: Whenever `a.length == 4` and `a[3]` is greater/equal than each of the other three elements, then the result must be `a[3]`. Consequently, the expert adds assertion `max3res` and assumption `max3of4` as in Fig. 3. Both names were chosen by the expert. Now the tool checks whether the assertion holds, whenever `max3res` holds. It turns out we have six runs (of 100

```
1  //@ assuming <max3res> or <max0res>;
2  int max(int a[]) {
3    //@ <max3of4>assume a.length==4 && a[0]<=a[3] && a[1]<=a[3] && a[2]<=a[3];
4    //@ <max0of1>assume a.length==1;
5    //@ <max0of4>assume a.length==4 && a[1]<=a[0] && a[2]<=a[0] && a[3]<=a[0];
6    int m = a[0];
7    for (int k=0; k < a.length; k++) {
8      if ( m < a[k]) {
9        m = a[k++];
10     }
11   }
12   //@ <max3res>assert m==a[3] assuming <max3of4>;
13   //@ <max0res>assert m==a[0] assuming <max0of1> or <max0of4>;
14   return m;
15 }
```

Fig. 3. int max(**int**[]) with conditioned assertion after some validation steps.

input arrays), where the assumption `max3res` holds: For one run, the assertion is violated—for input `{56,56,69,91}`, the program outputs `69` instead of `91`.

Since an invalid run was found, we are done. Observe that the domain expert merely scrutinized the initial program run, where the result was still correct. There are cases where more iterations are necessary. For example, the validation assistant could have started with a singleton array `{70}` or with array `{81,73,26,15}`. For either we would need two or more iterations as these program runs do not have any similarity with one of the 11 invalid runs. See Fig. 3 for the annotated program with further assumptions `max0of1` and `max0of4`.

Example 3. In a price calculation for cinema tickets, there are movies with different age restrictions and ticket prices, and there are several age groups with different discounts. The example might get much more complex with discount criteria such as happy hours, theme days, or vouchers. We conjecture that our validation approach works in these cases, too.

The relevant fragment of the ticket price calculation software is displayed in Fig. 4. Our initial goal is to validate program runs of the method `nextTicket`, starting from the termination points. The expert might first place an assertion in the called method `calcDscdPrice`, and then place the corresponding age group assumption in lines 5–7 of `nextTicket`.

There is a subtle bug which manifests in assertion `pricePlausible` (line 19) under assumption `senior`. Let's say the expert placed this assertion, because of the cinema's policy to sell tickets for at least 5 €. Assertion `minPr` guarantees that the normal price for each movie is more than `5.60` €. This holds for all program runs but cannot be formally proven, because the implementation of `Movie` is outside of boundary of the system under validation. Accordingly, the corresponding assuming `<>` is highlighted blue. Going back to `pricePlausible`, assume movie 2 has price `5.70` in some program run, this becomes with senior discount `4.85`, hence `<dscdSen,minPr>` is marked red.

```
1  //@ assuming <pricePlausible> or <tooYoung>;
2  public void nextTicket(Scanner input) {
3    System.out.print("Enter age: ");
4    int age = input.nextInt();
5    //@ <junior>assume age < 16;
6    //@ <regular>assume 16 <= age && age < 65;
7    //@ <senior>assume 65 <= age;
8    System.out.print("Select movie (1/2): ");
9    int movieNumber = input.nextInt();
10   //@ <mv1>assume movieNumber == 1;
11   //@ <mv2>assume movieNumber == 2;
12   Movie movie = movies[movieNumber];
13   //@ <tooYoung>assert age < movie.getMinAge() assuming <junior,mv1>;
14   if (age < movie.getMinAge()) {
15      System.out.println("Too young for this movie.");
16      return;
17   }
18   double dscdPrice = calcDscdPrice(movie, age);
19   //@ <pricePlausible>assert dscdPrice >= 5.00
           assuming <dscdReg,minPr> or <dscdJun,minPr> or <dscdSen,minPr>;
20   System.out.printf("Your price: %.2f €\n", dscdPrice);
21 }
22 private double calcDscdPrice(Movie movie, int age) {
23   //@ <dscdReg>assert getDiscount(age) == 0 assuming <regular>;
24   //@ <dscdJun>assert getDiscount(age) == 10 assuming <junior>;
25   //@ <dscdSen>assert getDiscount(age) == 15 assuming <senior>;
26   //@ <minPr>assert movie.getPrice() > 5.60 assuming <>;
27   return movie.getPrice() * (1 - getDiscount(age)/100.0);
28 }
```

Fig. 4. Cinema Example.

5 Conclusion and Related Work

We presented a procedure to validate program runs by a software engineer while iteratively generating a partial specification. This helps finding semantic bugs fast. The annotations can be re-used, for example, in regression verification.

Our validation procedure incorporates usage of verification and assertion checking tools. Assertion annotations are in use since the 1970s [14], verification has an even longer tradition. In *contract-based verification* [9], specifications are structured along method declarations, whereas our approach allows arbitrary dependencies via labeled assumptions, syntactically inspired by [4,5].

In [16], it is observed that *in-house tests* do not match the behavior of *field program runs*. Our approach directly validates the latter. Our validation is finished if every program run is covered by assertions highlighted in green/blue. This suggests an alternative to their proposed solution—*generating test cases mimicking field runs* [17].

Even if an assertion could not be formally verified (blue/yellow/red) we check it against said program field runs. We believe that this will suffice in our setting, without excluding future enhancements. Notably, there is an approach to generate test cases for partially unverified assertions [5].

An attempt to improve code reviews by animated symbolic execution is reported in [10]. In contrast, we guide an expert systematically through the code.

References

1. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: Using static analysis to find bugs. IEEE Software **25**(5), 22–29 (2008). https://doi.org/10.1109/MS.2008.130
2. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification – specification is the new bottleneck. In: Cassez, F., Huuck, R., Klein, G., Schlich, B. (eds.) Proc. 7th Conf. on Systems Software Verification. EPTCS, vol. 102, pp. 18–32 (2012). https://doi.org/10.4204/EPTCS.102.4
3. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification, 23rd Intl. Conf., Snowbird, UT, USA. LNCS, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
4. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Giannakopoulou, D., Méry, D. (eds.) Formal Methods, 18th Intl. Symp., Paris, France. LNCS, vol. 7436, pp. 132–146. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_13
5. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Dillon, L.K., Visser, W., Williams, L.A. (eds.) Proc. 38th Intl. Conf. on Software Engineering, Austin, TX, USA. pp. 144–155. ACM (2016). https://doi.org/10.1145/2884781.2884843
6. Fagan, M.E.: Design and code inspections to reduce errors in program development. IBM Systems Journal **15**(3), 182–211 (1976). https://doi.org/10.1147/sj.153.0182
7. Godefroid, P.: Test generation using symbolic execution. In: D'Souza, D., Kavitha, T., Radhakrishnan, J. (eds.) IARCS Ann. Conf. on Foundations of Software Technology and Theoretical Computer Science, Hyderabad, India. LIPIcs, vol. 18, pp. 24–33. Dagstuhl (2012). https://doi.org/10.4230/LIPIcs.FSTTCS.2012.24
8. Grätz, L., Hähnle, R., Bubel, R.: Examples for FASE NIER paper "finding semantics bugs fast" (artifact). In: 25th Intl. Conf. on Fundamental Approaches to Software Engineering, Munich, Germany. Zenodo (2022). https://doi.org/10.5281/zenodo.5806351
9. Hähnle, R., Huisman, M.: Deductive verification: from pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science: State of the Art and Perspectives, LNCS, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
10. Hentschel, M., Hähnle, R., Bubel, R.: Can formal methods improve the efficiency of code reviews? In: Ábrahám, E., Huisman, M. (eds.) Integrated Formal Methods, 12th Intl. Conf., Reykjavik, Iceland. LNCS, vol. 9681, pp. 3–19. Springer (2016). https://doi.org/10.1007/978-3-319-33693-0_1
11. The Independent Breast Screening Review 2018, House of Commons, HC, vol. 1799. UK Department of Health and Social Care (Dec 2018), https://www.gov.uk/government/publications/independent-breast-screening-review-report
12. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML reference manual (2013), revision: 2344.
13. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
14. Stucki, L.G., Foshee, G.L.: New assertion concepts for self-metric software validation. In: Proc. Intl. Conf. on Reliable Software, Los Angeles, California, USA. p. 59–71. Association for Computing Machinery (1975). https://doi.org/10.1145/800027.808425

15. Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C.: Bug characteristics in open source software. Empirical Software Engineering **19**(6), 1665–1705 (2014). https://doi.org/10.1007/s10664-013-9258-8
16. Wang, Q., Brun, Y., Orso, A.: Behavioral execution comparison: Are tests representative of field behavior? In: Intl. Conf. on Software Testing, Verification and Validation, Tokyo, Japan. pp. 321–332. IEEE Computer Society (2017). https://doi.org/10.1109/ICST.2017.36
17. Wang, Q., Orso, A.: Improving testing by mimicking user behavior. In: Intl. Conf. on Software Maintenance and Evolution, Adelaide, Australia. pp. 488–498. IEEE (2020). https://doi.org/10.1109/ICSME46990.2020.00053
18. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Elsevier, second edn. (2009)

SMC4PEP: Stochastic Model Checking of Product Engineering Processes

Hassan Hage[1,2](✉), Emmanouil Seferis[1,3], Vahid Hashemi[2], and Frank Mantwill[1]

[1] Helmut-Schmidt-University, Holstenhofweg 85, 22043 Hamburg, Germany
hassan.hage@hsu-hh.de
[2] AUDI AG, Auto-Union-Straße 1, 85057 Ingolstadt, Germany
[3] Technical University of Munich, Arcisstraße 21, 80333 Munich, Germany

Abstract. Product Engineering Processes (PEPs) are used for describing complex product developments in big enterprises such as automotive and avionics industries. The Business Process Model Notation (BPMN) is a widely used language to encode interactions among several participants in such PEPs. In this paper, we present SMC4PEP as a tool to convert graphical representations of a business process using the BPMN standard to an equivalent discrete-time stochastic control process called Markov Decision Process (MDP). To this aim, we first follow the approach described in an earlier investigation to generate a semantically equivalent business process which is more capable of handling the PEP complexity. In particular, the interaction between different levels of abstraction is realized by events rather than direct message flows. Afterwards, SMC4PEP converts the generated process to an MDP model described by the syntax of the probabilistic model checking tool PRISM. As such, SMC4PEP provides a framework for automatic verification and validation of business processes in particular with respect to requirements from legal standards such as Automotive SPICE. Moreover, our experimental results confirm a faster verification routine due to smaller MDP models generated from the alternative event-based BPMN models.

Keywords: Product Engineering Processes · Verification and validation · Probabilistic model checking · Markov decision processes · Probabilistic reward CTL.

1 Introduction

The ever-increasing technical challenges in products, for instance autonomous driving in automotive industries, requires *Original Equipment Manufacturers (OEMs)* to restructure their *Product Engineering Process (PEP)* from a mechanical-oriented to a system-oriented development to enable a rigorous verification and validation of its processes with respect to safety and non-safety requirements [5]. Additionally, legal authorities oblige OEMs to address consistency and traceability in their PEPs through compliance with standards such as *Automotive Software Process Improvement and Capability Determination (A-SPICE)* [21]. As the quality of a product is dependent on its processes's quality [17], consistent and qualitative processes are required for adequately addressing technical challenges, legal compliance and customer satisfaction.

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 155–162, 2022.
https://doi.org/10.1007/978-3-030-99429-7_9

A well known and most common modelling language of processes in industrial PEPs is *Business Process Model and Notation (BPMN)* [7] which we refer to as *pool-based BPMN (*pBPMN*)*. pBPMNs provide different users with their internal process workflows in a graphical notation and show the communication and dependency between different organization within the PEP. With the aim of facing the above mentioned challenges, the previous work in [8] shows the need for a revision of the BPMN language which is called *event-based BPMN (*eBPMN*)* in this paper. The processes, which are modelled according to the BPMN guidelines, are enriched with events and time symbols while message-flows of all processes are removed. On that way we ensure to capture time aspects like milestones of PEPs, to enable a communication between processes on different levels of abstraction by means of events, to determine the logical dependencies between processes and finally to remove process redundancies for ensuring consistency and traceability in PEPs. These argumentations on the process design motivated us to consider eBPMNs as a better design language in SMC4PEP. We discuss later that the eBPMN is more beneficial than its pBPMN counterpart in generating smaller MDPs and hence, enabling faster verification routine. The core part of the SMC4PEP relies on converting pBPMNs to eBPMNs while implicitly reducing the model size which is in turn done by removing redundant processes without losing information. As a bi-product, it realizes consistency in PEPs by message passing on different levels of abstraction which is not the case if pBPMN is used as a design language. Then, SMC4PEP converts the generated eBPMN to an equivalent MDP described in the syntax of the probabilistic model checking tool PRISM [15]. SMC4PEP ensures not only the consistency in PEPs but also allows for automated verification of generated MDPs against formal description of requirements from legal standards such as A-SPICE.

2 Related Tools

There exist different tools for analyzing business processes. Due to the wide industrial use of the pBPMN standard, the most common tools for analyzing business processes use this graphical representation of processes as an initial model.

The work of Ou-Yang and Lin in [19] provides an approach to translate pBPMNs to the Modified BPEL4WS representation and then to the Colored Petri-net XML (CPNXML) that can finally be verified by using CPN tools. This approach has restrictions in the support of split and merge conditions. The approach of Daclin et al. in [1] or Mendoza Morales in [18] realize a conversion of pBPMNs to a set of Timed Automata (TA) that uses Clocked Computation Tree Logic (CCTL) for the verification. In the work of Lam in [16] pBPMNs are converted to the New Symbolic Model Verifier (NuSMV) language. Then NuSMV enables an analysis of the processes using model checking techniques and verifying properties by the Computation Tree Logic (CTL). The approaches discussed in [1, 16, 18, 19] do not consider probability distributions and non-deterministic choices of processes which are required for complex processes such as PEP. Duran et al. [3] develop the approach of Rewriting Logic to enrich pBPMNs with timing and probabilistic properties. They verify stochastic properties such as synchronization time, probability distributions by means of the Parallel Statistical Model Checking And Quantitative Analysis (PVeStA) tool. However, mes-

sage passing between different processes especially on different levels of abstraction is not considered. Finally, Herbert in [14] develop an algorithm for converting pBPMNs into MDPs, where resources like timing and probabilities are considered while message passing is performed between sub-processes. Nevertheless, the size of investigated processes is small and limited and hence, message passing between large processes in particular with different levels of abstraction is not considered. Moreover, the process model is designed with less message passing and complexity to avoid the already known state-space explosion in the generated MDP model which consequently means that this approach is not applicable on complex processes like PEPs.

3 SMC4PEP **Architecture and Workflow**

As shown in Fig. 1, SMC4PEP consists of three modules, namely: (I) Differentiator, (II) Converter and (III) Generator. The Differentiator determines if the input model is a pBPMN or eBPMN. In case it is a pBPMN, the Converter converts the process model automatically to an eBPMN and moves then to the Generator. Otherwise with an eBPMN as input, SMC4PEP skips the Converter and moves automatically to the Generator. Finally, the Generator converts the eBPMN into an MDP described in the PRISM syntax which can directly be analyzed in PRISM. The process of generating the output PRISM model consists of three steps discussed as follows.

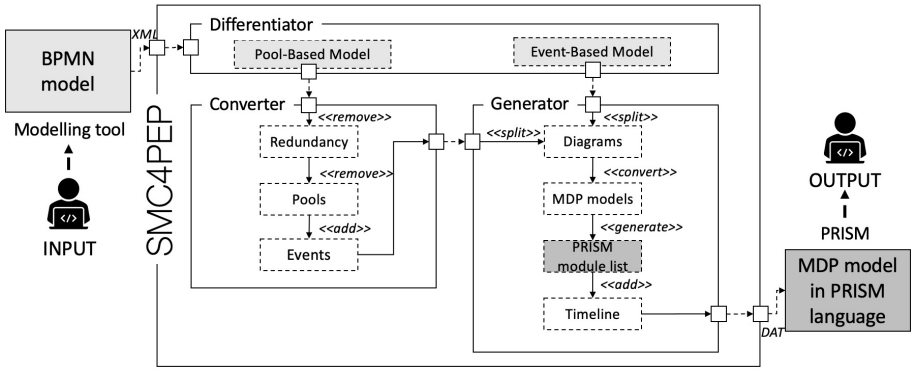


Fig. 1. Architecture of the tool SMC4PEP.

Input. SMC4PEP requires a business process model as input with no limitation of abstraction levels. Process models can be designed either according to the guidelines in [7] or [8] with different modelling tools such as Enterprise Architect [4]. Each process model needs to be exported as an XML document for the readability of SMC4PEP.

SMC4PEP**.** The Differentiator of SMC4PEP receives the input document and checks the content of the BPMN model based on the syntactic and semantic differences between eBPMN and pBPMN. According to [7] message passing between processes is

performed by message flows from tasks to task of the associated sub-processes, while each sub-process obtains its own boundary called pool. In the eBPMN approach message flows and pools are eliminated [8] and each sub-process obtains its own diagram. Then the process is enriched by events to enable message passing between each process. In case of a detected pBPMN, the Differentiator triggers the Converter, otherwise the Converter will be skipped and SMC4PEP starts automatically the Generator.

The Converter of SMC4PEP analyzes the number of identical processes within the whole process model to remove first redundant processes of pBPMN that may occur on different levels of abstraction. Redundant processes are determined when one process is equal to a second process in all elements of the model. That means in all number and content of tasks, number and content of events, number and content of gateways, role/responsible person of the process as well as number and order of sequence flows. The definition of these elements is available in [7]. When equal processes are detected, SMC4PEP eliminates all equal processes apart from one. Afterwards, all pools of the process models are removed and each sub-process obtains its own diagram. Finally, message flows are eliminated and replaced with events to ensure message passing and logical dependencies between the processes on different levels of abstraction. Note that message passing of the removed processes are also considered so that only one process enables a communication between different levels of abstraction. Finally, the pBPMN initial model is converted into an eBPMN and the Converter triggers the Generator.

The Generator requires an eBPMN which is provided either from the Differentiator or Converter. Then the process model is split into its number of diagrams. Afterwards, the Generator converts each diagram to an MDP taking into account message passing on different levels of abstraction by events, probability distributions and non-deterministic choices. Followed by the next step, the Generator of SMC4PEP generates for each MDP model a PRISM module list which are then combined to one main PRISM module list. Finally, in case of an available timeline [8] in the process model, the PRISM module list is enriched by the values of the timeline to consider time aspects and process execution costs as rewards in the MDP model described in the PRISM syntax.

Output. SMC4PEP saves the generated MDP model described in the syntax of PRISM as a DAT document which can be uploaded into the probabilistic model checker PRISM. It is worthwhile to mention that there are quite a number of tools which are able to read the PRISM modelling language. Among others, model checkers Storm [2], PARAM [10], ePMC [11] and Modest [12] can read our generated PRISM model for doing model checking various properties of interests.

4 Case Studies

For the evaluation of SMC4PEP, we converted two different use cases with SMC4PEP. Before, we developed an algorithm inspired by the work of [14] to convert a pBPMN directly into an MDP. Note that this conversion is not applicable on complex processes with different levels of abstraction. Complexity means a higher number of message passing between processes, probability distributions and non-deterministic choices. Therefore, for the evaluation we assumed that in pBPMN a communication between

different levels of abstraction is possible by merging all diagrams to one main diagram, although in real processes it is not the case. This assumption is met to obtain the MDP sizes of the pBPMN. On that way MDP sizes generated through a pBPMN and eBPMN model can be compared and the effectiveness of the eBPMN can be approved. The first use case describes the process of testing an autonomous park pilot with three levels of abstraction and includes five roles where each role performs its associated task of the process. The second use case handles a more complex process of an urgent request for a change of the vehicle construction during the PEP. In total this use case extends over four levels of abstraction and includes eleven roles. Both use cases are provided by an automotive OEM. We run all experiments on an Core i7 laptop running Windows 10.

Table 1 provides promising results generated based on SMC4PEP. The generated MDP model of the first use case with two levels of abstraction is for the eBPMN 33.8% in states and 40.7% in transitions less than for the pBPMN. Moreover, the generated MDP model in the third level of abstraction is in the eBPMN 67.78% in states and 73.11% in transitions less than in the pBPMN. The build time of the MDP model for the eBPMN with three levels of abstraction is higher compared to the pBPMN. Note that the MDP model is built only once which has no impact on the run-time of model checking MDPs. This is indeed the case for generating a formalism like MDP from giant BPMN models and use it several times for model checking various properties. The generated MDP models of use case two with four levels of abstraction are large compared to the first use case due to the high number of activities, probability distributions and non-deterministic choices of the processes. Nevertheless, the effectiveness of the eBPMN for complex processes is strongly confirmed by the generated MDP size of the second use case on four levels of abstraction which is far less than the MDP size of pBPMN. Finally, our generated MDP models from eBPMN have much smaller sizes compared to the approach discussed in [14]. In particular, for the second use case we got several order of magnitudes reduction in model size which is significant for an efficient model checking routine. However, similar to [14] we also realize the state space explosion problem which can be alleviated using bisimulation minimization techniques [6,9,13].

Table 1. Results of the analyzed processes.

BPMN model	Use case	Abstraction level	MDP model States	MDP model Transitions	Built time (s)
pBPMN	1	2	423	1143	0.071
eBPMN	1	2	280	685	0.037
pBPMN	1	3	5276	21503	0.170
eBPMN	1	3	1700	5782	0.551
pBPMN	2	4	$93x10^{16}$	$14x10^{16}$	4.263
eBPMN	2	4	$17x10^{10}$	$19x10^{11}$	0.871

At the end, we take the PRISM tool for model checking some properties of interest described in the *Probabilistic Reward Computation Tree Logic (PRCTL)* [20]. It is worthwhile to note that for SMC4PEP we provide the first use case as an eBPMN to

capture time and cost aspects of the PEP by a timeline while the second use case is described first in pBPMN and then converted to eBPMN. Firstly, we verify some properties based on the A-SPICE guidelines [21] by φ_1, φ_2 and φ_3. The properties are taken from the *Generic Practice (GP)* of A-SPICE Level 2 [21] where each level of A-SPICE determines the quality of the processes. The property GP 2.1.7 of A-SPICE denoted as φ_1 which requires ensuring no deadlocks in the processes and reaching the final state of the process with the probability of 100%. Additionally by φ_2 we denote the property GP 2.1.2 which ensures the ability of performing the process to fulfil the identified objectives similar to φ_1. Moreover, the GP 2.1.3 is denoted by φ_3 through which we ensure that our process does not deviate from its original setting according to A-SPICE. Finally for use case one, the non-functional properties are denoted by φ_4 which delivers the minimum days (d) for performing the whole process, and by φ_5 which enables the expected cost estimation of the process obtained in accumulated working days (wd). We have to note here that φ_4 is obtained by the GUI simulator of PRISM. The results of the property verification obtained from PRISM are depicted in Table 2.

Table 2. Model checking of eBPMN processes.

Abstraction level	Use Case	MDP model States	MDP model Transitions	Properties φ_1	φ_2	φ_3	φ_4(d)	φ_5(wd)
2	1	280	685	✓	✓	✓	78	267.9
3	1	1700	5782	✓	✓	✓	110	346.5
4	2	$17x10^{10}$	$19x10^{11}$	✓	✓	✓	-	-

5 Conclusion

In this paper we presented the new tool SMC4PEP to enable in the first phase an automated conversion of complex process models such as PEPs that are modelled according to the BPMN standard [7] into revised process models based on the modelling approach of [8]. This conversion paves the way for consistency and traceability of complex PEPs by removing redundant processes and enabling an exchange between different levels of process abstraction. In the second phase, SMC4PEP converts the new process model into an MDP to capture stochastic properties of a PEP and to enable an automated verification of the MDP using PRISM against formal descriptions of requirements. In case of designing a new PEP based on [8], SMC4PEP considers also the timeline of processes to capture time and cost aspects of a PEP that are essential for developing a new product in particular in automotive and avionics industries. Finally, we approved the effectiveness of our tool in an automotive case study where we compared pBPMNs with eBPMNs and verified some properties of interest such as legal regulations from A-SPICE.

Acknowledgments. This work is supported by the Helmut-Schmidt-University in Hamburg and by the AVAI project at AUDI AG in Ingolstadt.

References

1. Daclin, N., Vallespir, B., Vincent, C.: Enabling model checking for collaborative process analysis: from bpmn to 'network of timed automata'. In: Enterprise Inforation Systems. vol. 9, pp. 279–299. Taylor and Francis (2015)
2. Dehnert, C., Junges, S., Katoen, J., Volk, M.: The probabilistic model checker storm (extended abstract). CoRR **abs/1610.08713** (2016), http://arxiv.org/abs/1610.08713
3. Duran, F., Rocha, C., Salaün, G.: Stochastic analysis of bpmn with time in rewriting logic. In: Science of Computer Programming. pp. 168, pp. 1–17. Elsevier (2018)
4. Europe, S.S.C.: Enterprise Architect 15.2 [Software] (2021), https://www.sparxsystems.de
5. Gausemeier, J., Dumitrescu, R., Steffen, D., Czaja, A., Wiederkehr, O., Tschirner, C.: Systems engineering in der industriellen praxis. Heinz Nixdorf Institut, Frauenhofer Institut, UNITY AG (2013)
6. Gebler, D., Hashemi, V., Turrini, A.: Computing behavioral relations for probabilistic concurrent systems. In: ROCKS 2012. pp. 117–155. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
7. Group, O.O.M.: Business process model and notation (bpmn). Website (2014), https://www.omg.org/spec/BPMN
8. Hage, H., Hashemi, V., Mantwill, F.: Towards a systems engineering based automotive product engineering process. In: Software Architecture - 14th European Conference. Communications in Computer and Information Science, vol. 1269, pp. 527–541. Springer (2020)
9. Hahn, E.M., Hashemi, V., Hermanns, H., Turrini, A.: Exploiting robust optimization for interval probabilistic bisimulation. In: Agha, G., Van Houdt, B. (eds.) Quantitative Evaluation of Systems. pp. 55–71. Springer International Publishing, Cham (2016)
10. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Param: A model checker for parametric markov models. In: CAV. pp. 660–664 (2010)
11. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscas m c: a web-based probabilistic model checker. In: International Symposium on Formal Methods. pp. 312–317. Springer (2014)
12. Hartmanns, A., Hermanns, H.: The modest toolset: An integrated environment for quantitative modelling and verification. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 593–598. Springer (2014)
13. Hashemi, V., Hermanns, H., Song, L., Subramani, K., Turrini, A., Wojciechowski, P.: Compositional bisimulation minimization for interval markov decision processes. In: Language and Automata Theory and Applications. pp. 114–126. Springer (2016)
14. Hebert, L.: Specification, verification and optimisation of business process. Technical University of Denmark (2014)
15. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with prism: A hybrid approach. In: Katoen, J.P., Stevens, P. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 52–66. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
16. Lam, V.S.W.: Formal analysis of BPMN models: a nusmv-based approach. Int. J. Softw. Eng. Knowl. Eng. **20**(7), 987–1023 (2010), https://doi.org/10.1142/S0218194010005079
17. Martin Glinz, S.F.: Software quality selected chapter, chapter 7, process quality. University of Zürich, Institut for Informatics (2007)
18. Mendoza Morales, L.: Business process verification: The application of model checking and timed automata. CLEI Electronic Journal **17**, 3–3 (08 2014)
19. Ou-Yang, C., Lin, Y.D.: BPMN-based business process model feasibility analysis: a Petri net approach. vol. 46, pp. 3763–3781. Taylor and Francis (2008), https://doi.org/10.1080/00207540701199677

20. Parker, D.: Lecture 14 model checking for MDPs. University of Oxford, Department Science (2011)
21. SIG, V.Q.W.G...A.: Automotive SPICE Process Assessment / Reference Model. Automotive SPICE (2017)

Symbolic Predictive Cache Analysis for Out-of-Order Execution

Zunchen Huang (✉) and Chao Wang

University of Southern California, Los Angeles CA 90089, USA
{zunchenh, wang626}@usc.edu

Abstract. We propose a trace-based symbolic method for analyzing cache side channels of a program under a CPU-level optimization called out-of-order execution (OOE). The method is predictive in that it takes the in-order execution trace as input and then analyzes all possible out-of-order executions of the same set of instructions to check if any of them leaks sensitive information of the program. The method has two important properties. The first one is *accurately* analyzing cache behaviors of the program execution under OOE, which is largely overlooked by existing methods for side-channel verification. The second one is *efficiently* analyzing the cache behaviors using an SMT solver based symbolic technique, to avoid explicitly enumerating a large number of out-of-order executions. Our experimental evaluation on C programs that implement cryptographic algorithms shows that the symbolic method is effective in detecting OOE-related leaks and, at the same time, is significantly more scalable than explicit enumeration.

Keywords: program analysis · out-of-order execution · side channel · SMT solver

1 Introduction

There has been growing interest in recent years in detecting side-channel leaks in software using automated program analysis and verification techniques, due to the increased awareness of the threat of real-world side-channel attacks [4,15,18]. These are *side-channel* attacks because they exploit dependencies between sensitive information of the program and non-functional properties of the computing platform, including cache-related timing variations caused by CPU-level optimizations such as pipelining and branch prediction. While there are existing methods for detecting these side channels based on static analysis [6,28,31] and symbolic execution [3, 10–12, 29], they do not accurately model an important CPU-level optimization called out-of-order execution (OOE).

Out-of-order execution is widely adopted by modern CPUs. It is possible for a program to be free of side-channel leaks when instructions are executed in the *program order* but have leaks when they are executed out of order. Here, the program order refers to the order in which instructions appear in the program. However, modeling out-of-order execution during program analysis is a

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 163–183, 2022.
https://doi.org/10.1007/978-3-030-99429-7_10

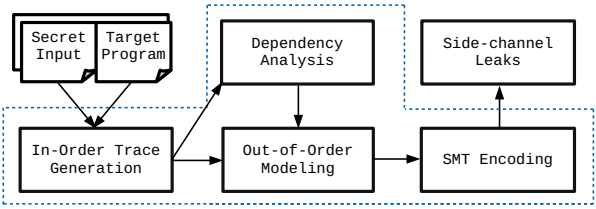


Fig. 1. SPRECA – symbolic predictive analysis for out-of-order execution.

challenging task due to the inherently large number of possible scenarios that must be considered. Generally speaking, instructions within a fixed window (an imaginary window used to model the effect of hardware features including the reorder buffer, issue queue, and load-store queue) may be executed in any order as long as it respects the semantics of the program. Thus, given N instructions, the number of possible execution orders can be as large as $O(N!)$. Since it is practically intractable to examine these execution orders individually, existing methods had to choose from the following two undesired outcomes: if they over-approximate, they may report bogus leaks since some infeasible execution orders will be included; but if they under-approximate, they may miss real leaks since some feasible execution orders will be excluded.

To solve the aforementioned problem, we propose a *trace-based symbolic predictive analysis* to accurately *and* efficiently analyze the OOE related cache behaviors. Here, *accurately* means that our method does not over- or under-approximate the OOE behaviors but precisely encodes these behaviors as a set of logical constraints; *efficiently* means that our method avoids enumerating the out-of-order executions explicitly to avoid the exponential blowup; instead it leverages an off-the-shelf SMT solver to conduct a symbolic analysis of the logical constraints. Our method is *predictive* in that, given an in-order execution trace of the program, it analyzes the cache behaviors of all out-of-order executions of the instructions that appeared in the in-order execution, instead of executing them.

Fig. 1 shows the overall flow of our method, named SPRECA, which takes an annotated C program as input; the annotation marks program inputs as either public or private (secret). Internally, our method has three steps. In the first step, it utilizes the `LLVM` compiler to parse the C program, compute the program dependencies, and use the information to instrument the LLVM bit-code. The instrumented program, at run time, can generate the in-order execution trace. In the second step, our method encodes the set of all possible OOE related cache behaviors as a set of logical constraints, to be solved by an off-the-shelf SMT solver. In the third step, our method checks if there are *secret-dependent* divergent cache behaviors, e.g., an out-of-order execution causing a cache hit for one value of the secret variable but a cache miss for another value of the secret variable.

The main contribution of our work is symbolically modeling the OOE related cache behaviors accurately and efficiently. We design the SMT encoding (to be presented in Section 5) carefully to make it compact. For example, a straightforward encoding of all possible permutations of N instructions would lead to an SMT formula of size $O(N^3)$, since any instruction may have any other instruction as its predecessor and, as a result, the update function must be encoded for each predecessor's cache state and the current cache state. Our method, in contrast, avoids most of these update functions by leveraging the program dependency relations recorded in the in-order execution trace to prune away the infeasible permutations.

Our method differs significantly from the method of Guo et al. [10,11] based on symbolic execution. While their method also uses symbolic analysis, they only made the program input symbolic, whereas the out-of-order executions are still enumerated explicitly (this is evident based on their use of a technique designed for speeding up explicit enumeration, called *partial order reduction*). In other words, for each out-of-order execution, they had to generate an SMT formula to check if it has divergent cache behaviors; as a result, they did not avoid the exponential blowup. In contrast, our method generates a single SMT formula to encode all possible out-of-order executions associated with the in-order execution. In addition to being more efficient, our single-formula based encoding can be more easily adapted to model other CPU-level optimizations by slightly modifying how dependencies are encoded as logical constraints.

We have implemented our method in a software tool by leveraging the open-source LLVM compiler [17] and the Z3 SMT solver [19]. Specifically, we use LLVM to parse the C program, compute the program dependencies, and instrument the bit-code, to generate the in-order execution trace at run time. We use Z3 to implement symbolic analysis of the out-of-order executions. We evaluated our method on a set of C programs from `OpenSSL` that implement well-known block ciphers and cryptographic hash functions. The experimental results show that our method, by accurately modeling the OOE related cache behaviors, can detect OOE-related side-channel leaks that otherwise would have been missed. The results also show that our SMT solver-based symbolic analysis is significantly more scalable than explicit enumeration.

To summarize, this paper makes the following contributions:

- We propose a trace-based symbolic predictive analysis for detecting OOE related cache side-channel leaks.
- We rely on an off-the-shelf SMT solver to *accurately* and *efficiently* analyze the out-of-order executions associated with an in-order execution trace.
- We demonstrate the effectiveness of our method on C programs from an open-source library that implements well-known cryptographic algorithms.

The remainder of this paper is organized as follows. First, we motivate our work using examples in Section 2. Then, we provide the technical background in Section 3. Next, we present our method in Sections 4 and 5, followed by the experimental results in Section 6. We review the related work in Section 7. Finally, we give our conclusions in Section 8.

2 Motivation

In this section, we use examples to illustrate the cache behaviors of the in-order execution and an out-of-order execution. We also explain the high-level idea of our trace-based symbolic analysis.

2.1 The Example Program

Fig. 2 shows the code snippet which, for ease of presentation, is written in a mixture of C and simplified assembly language. Here, assume $i \in \{0, 1, 2\}$ is a secret variable and each array element $A[i]$ occupies 4 bytes in memory. Furthermore, while our method handles realistic cache size and configurations, in this motivating example, we assume the cache has only one set, consisting of 3 cache lines, with each cache line holding only 4 bytes. We assume the cache is fully associative, and uses the LRU (least recently used) replacement policy. Under these assumptions, each array element $A[i]$ occupies an entire cache line.

```
1     load  A[0];
2     load  A[1];
3     load  A[2];
4     store A[i]; /* Can the secret value i affect the cache behavior? */
5     load  B;
```

Fig. 2. An example program where the value of i is a secret.

2.2 The Execution Order

The order in which instructions are written in a program is called the *program order*. During the in-order execution, instructions are executed according to their program order. Without loss of generality, we assume that there are two types of instructions: memory-related instructions such as Load and Store, and non-memory-related instructions, such as ALU and branch instructions. As far as this work is concerned, our focus is on memory-related instructions because non-memory instructions do not affect cache behavior [1].

Fig. 3 compares the in-order execution on the left with a possible out-of-order execution on the right. The out-of-order execution is a permutation of instructions of the in-order execution that, at the same time, must respect the semantics of the original program. In both of these two execution traces, each row represents an instruction and its associated memory address. Note that while a program may have if-else statements and thus multiple paths, an execution trace corresponds to only one program path.

[1] Non-memory instructions may impose ordering constraints over memory-related instructions. These constraints are computed by our method, and used to constrain the analysis of out-of-order executions; details are in Section 4.

```
1   In-order                                          Out-of-order
2   I1: load  0x77ef5bd0     /*A[0]*/      ||   I1: load  0x77ef5bd0     /*A[0]*/
3   I2: load  0x77ef5bd4     /*A[1]*/      ||   I2: load  0x77ef5bd4     /*A[1]*/
4   I3: load  0x77ef5bd8     /*A[2]*/      ||   I3: load  0x77ef5bd8     /*A[2]*/
5   I4: store 0x77ef5bd0,... /*A[i]*/      ||   I5: load  0x77ef5bdc     /*B   */
6   I5: load  0x77ef5bdc     /*B   */      ||   I4: store 0x77ef5bd0,... /*A[i]*/
```

Fig. 3. Two execution orders of the example program in Fig. 2.

2.3 The Cache State

Given a program execution, regardless of whether it is the in-order execution or one of the out-of-order executions, it is straightforward to compute changes of the cache state at each step. The cache state of our running example can be defined as a tuple $S = \langle Age(A[0]),\ Age(A[1]),\ Age(A[2]),\ Age(B)\rangle$, consisting of the ages of cache lines associated with the four program variables. Since we assume that the cache holds at most 3 variables (lines) at any moment if a variable is inside the cache, its age must be 0, 1, or 2; and if it is evicted from the cache, its age must be 3. Initially, the cache state is $S_0 = \langle -1, -1, -1, -1\rangle$, where -1 is a special symbol meaning it is not loaded into cache yet.

In-Order Cache Behavior As shown in Fig. 4 for the in-order execution, executing the first instruction `load` $A[0]$ changes the cache state to $S_{I_1} = \langle 0, -1, -1, -1\rangle$ from S_0, where S_{I_1} is the cache state after executing I_1. That is, variable $A[0]$ now occupies the youngest cache line. Similarly, after executing the first three instructions, the cache state becomes $S_{I_3} = \langle 2, 1, 0, -1\rangle$, meaning that $A[2]$ occupies the youngest cache line and $A[0]$ occupies the oldest cache line. Thus, executing the instruction `store` $A[i]$ results in a *cache hit* regardless of whether $i = 0, 1$, or 2. At this moment, the age of variable B remains -1 since it has not yet been loaded to the cache.

```
1   In-order (for i==1)                                  In-order (for i==0)
2   S_I1 = ⟨ 0,-1,-1,-1⟩ /*A[0] ColdMiss*/      ||   S_I1 = ⟨ 0,-1,-1,-1⟩ /*A[0] ColdMiss*/
3   S_I2 = ⟨ 1, 0,-1,-1⟩ /*A[1] ColdMiss*/      ||   S_I2 = ⟨ 1, 0,-1,-1⟩ /*A[1] ColdMiss*/
4   S_I3 = ⟨ 2, 1, 0,-1⟩ /*A[2] ColdMiss*/      ||   S_I3 = ⟨ 2, 1, 0,-1⟩ /*A[2] ColdMiss*/
5   S_I4 = ⟨ 2, 0, 1,-1⟩ /*A[i] Hit     */      ||   S_I4 = ⟨ 0, 2, 1,-1⟩ /*A[i] Hit     */
6   S_I5 = ⟨ 3, 1, 2, 0⟩ /*B    ColdMiss*/      ||   S_I5 = ⟨ 1, 3, 2, 0⟩ /*B    ColdMiss*/
```

Fig. 4. Cache behavior of the *in-order* execution does not depend on the secret value i; that is, for all $i = 0, 1, 2$, accessing $A[i]$ results in a cache hit.

Out-of-Order Cache Behavior There can be many out-of-order executions, or permutations of instructions, corresponding to an in-order execution. While they must preserve the semantics of the in-order execution, they do not have to preserve its cache behavior. Thus, even if the in-order execution does not have

divergent cache behaviors (with respect to a secret variable), one of the out-of-order executions may have divergent cache behaviors. As shown in Fig. 5 for this particular out-of-order execution that reorders `store A[i]` and `load B`, when $i \neq 0$, accessing $A[i]$ results in a cache hit, but when $i = 0$, it results in a cache miss.

```
1   Out-of-order (for i==1)                          Out-of-order (for i==0)
2   S_I1'= ⟨ 0,-1,-1,-1⟩ /*A[0] ColdMiss*/     ||   S_I1'= ⟨ 0,-1,-1,-1⟩ /*A[0] ColdMiss*/
3   S_I2'= ⟨ 1, 0,-1,-1⟩ /*A[1] ColdMiss*/     ||   S_I2'= ⟨ 1, 0,-1,-1⟩ /*A[1] ColdMiss*/
4   S_I3'= ⟨ 2, 1, 0,-1⟩ /*A[2] ColdMiss*/     ||   S_I3'= ⟨ 2, 1, 0,-1⟩ /*A[2] ColdMiss*/
5   S_I5'= ⟨ 3, 2, 1, 0⟩ /*B    ColdMiss*/     ||   S_I5'= ⟨ 3, 2, 1, 0⟩ /*B    ColdMiss*/
6   S_I4'= ⟨ 3, 0, 2, 1⟩ /*A[i] Hit     */     ||   S_I4'= ⟨ 0, 3, 2, 1⟩ /*A[i] Miss    */
```

Fig. 5. Cache behavior of the *out-of-order* execution depends on the secret value i; that is, accessing $A[i]$ results in a cache hit when $i \neq 0$ but a cache miss when $i = 0$.

2.4 The Side-channel Leak

Whenever the cache behavior of an execution (regardless of whether it is the in-order execution or an out-of-order execution) depends on the value of a secret variable, it is called a side-channel leak. This is a security risk because, in modern CPUs, a cache hit only takes 1-3 CPU cycles whereas a cache miss may take up to a hundred CPU cycles. By observing the difference in the execution time of a victim program, the attacker may be able to deduce a certain amount of information about the secret.

In our running example, since `store A[i]` is dependent on the value of the secret variable i, we need to check if executing `store A[i]` leads to divergent cache behaviors. During the in-order execution, the answer is no, since it results in a cache hit for all $i = 0, 1$, and 2. Thus, the in-order execution has no side-channel leak. During one of the out-of-order executions, however, the answer is yes, since it results in a cache hit for some value of i but a cache miss for some other value of i. Thus, the out-of-order execution has a leak.

Generally speaking, there are two types of side-channel analysis techniques: approximate and accurate. While over- or under-approximation may be fast, it leads to poor results, i.e., reporting bogus leaks or missing real leaks. Thus, we are only concerned with accurate analysis techniques. In this context, while it is possible to examine each individual out-of-order execution, it will lead to exponential blowup. Our method, in contrast, encodes the cache behaviors of all out-of-order executions in a single logical formula. The formula is then solved using an efficient, off-the-shelf SMT solver to avoid an exponential blowup.

3 Preliminaries

In this section, we present the technical background related to our analysis of the out-of-order executions and divergent cache behaviors.

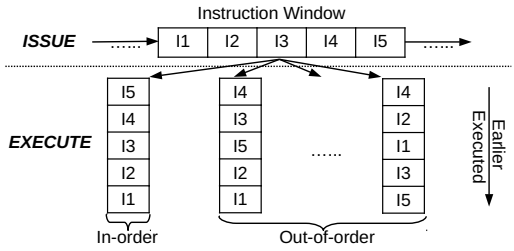


Fig. 6. The instruction window and the different execution orders.

3.1 The Execution Model

Recall that modern CPUs may execute instructions of a program in any order as long as the end result remains the same. The default order is the *program order*, i.e., the order in which instructions appear in the program. For performance reasons, however, the CPU does not always follow the program order, because some instructions may be significantly slower than others and, instead of waiting for the slower instructions to complete, the CPU may choose to execute some subsequent instructions as long as the program semantics is preserved.

Instruction Window As shown in Fig. 6, we use an imaginary *instruction window* to abstract the behavior of various hardware components inside the CPU for supporting out-of-order execution. The size of this instruction window depends on the CPU, including but not limited to the sizes of its reorder buffer, issue queue, and load-store queue. For this work, however, there is no need to delve into the hardware details. Instead, it suffices to assume that within this imaginary window of N instructions, the CPU may choose any execution order as long as the end result remains the same.

Data Hazards To make sure that the end result remains the same, only the out-of-order executions that respect the data dependencies of the original program are allowed. In the computer architecture literature, violations of such dependencies are called *hazards*. Specifically, there are three types of hazards, named RAW (read after write), WAR (write after read), and WAW (write after read), respectively. It is worth noting that RAR (read after read) is not a hazard.

3.2 The Cache Model

Without loss of generality, we assume the cache has K cache lines in total and each cache line has 64 bytes. The cache lines are further divided into M sets, which means each set has (K/M) cache lines. The memory is also divided into 64-byte blocks, each of which is mapped to a unique set. Within the same set, however, the 64-byte block may occupy any of the cache lines. Thus, within the set, it is called *fully associative*; overall, the entire cache is called *set associative*. In this context, a *fully associative* cache is a special case (K-way set associative), while a *direct mapped* cache is another special case (1-way set associative).

The Cache State The cache state is a tuple $S_I = \langle Age(v_1), \ldots, Age(v_n)\rangle$, where each $v_i \in Vars$ $(1 \leq i \leq n)$ is a variable in the program, and $Age(v_i)$ is the age of the cache line associated with v_i. $Vars$ is the set of all variables. Here, we use the subscript in S_I to indicate that it is the cache state resulting from executing the instruction I. Assume that K is the number of cache lines in a set. The domain of $Age(v_i)$ is $\{0, 1, \ldots, K, -1\}$, where an age from 0 to $K-1$ means the variable is inside the cache, while K means the variable is evicted from cache and -1 means it has never been loaded into cache.

We assume that the cache uses the LRU (least recently used) replacement policy. Given a cache state S_I and an instruction I', the new cache state $S_{I'}$ is computed by the $Update(S_I, I')$ function. Assuming that $v \in Vars$ is the variable used by the instruction I', $u_1 \in Vars$ is another variable whose age was younger than v in S_I, and $u_2 \in Vars$ is yet another variable whose age was older than v in S_I, we compute the new cache state $S_{I'} = \langle Age'(v_1), \ldots, Age'(v_n)\rangle$ as follows:

- $Age'(v) = 0$;
- $Age'(u_1) = Age(u_1) + 1$;
- $Age'(u_2) = Age(u_2)$.

That is, the most recently used variable (v) occupies the youngest cache line, any variable (u_1) whose age was younger than v in S_I increases its age by 1, and any variable (u_2) whose age was older than v in S_I keeps its age unchanged.

3.3 The Side-channel Leak Condition

Whenever there is a dependency between the secret and some divergent cache behaviors of an execution, there is a side-channel leak. Thus, there are two requirements. First, there must be divergent cache behaviors, i.e., memory-related instruction causing a cache miss for some input value but a cache hit for some other input value. Second, the input value causing divergent cache behaviors must be a secret, e.g., a password, security token, or cryptographic key.

Thus, the side-channel leak condition can be defined as follows:

$$\exists\, E, I, v_1, v_2 \;.\; CacheStatus(E, I, v_1) \neq CacheStatus(E, I, v_2)$$

Here, E denotes an execution, and $I \in E$ is an instruction in E; v_1 and v_2 are two values of a secret variable $v_s \in Vars$; and $CacheStatus(E, I, v_s)$ is a function that returns the cache status (hit or miss) when instruction I is executed in E using v_s.

4 Analyzing the In-Order Execution

In this section, we present our method for generating, and then analyzing the in-order execution trace. There are two tasks. The first one is to compute the dependencies of memory-related instructions. The second one is to compute the default cache states. Both the dependencies and the default cache states will be used during our symbolic analysis of the out-of-order executions.

4.1 Computing the Dependencies

There are two types of dependencies associated with the in-order execution of a program: explicit dependencies and implicit dependencies.

Explicit Dependencies Explicit dependencies refer to data conflicts that can be directly observed during the execution, by looking at the actual addresses of memory blocks used by the instructions at run time. Consider the in-order execution example in Fig. 3 (left). Since both instructions I_4 and I_1 access the memory block at the address 0x77ef5bd0, and at least one of them is a `store` operation, these two instructions have an explicit dependency; that is, they cannot be reordered during out-of-order.

```
1       load r1 A[0]   /*LD A[0]*/
2       mul  r1 5      /* */
3       add  r2 r1     /* */
4       mov  r3 r2     /* */
5       ...
6       store A[1] r2  /*ST A[1]*/
```

Fig. 7. Example implicit dependency that cannot be observed in the execution trace.

Implicit Dependencies Implicit dependencies, on the other hand, refer to data conflicts that cannot be directly observed during the in-order execution. Fig. 7 shows an example. The code snippet shows that `store A[1]` is dependent on `load A[0]`, through the def-use chain of (register) variables `r1-r3`. Since non-memory instructions (`mul, add, mov` in this example) do not show up in the logged execution trace, their constraints on the memory instructions would have been lost if we do not compute and record them explicitly into the execution trace.

In our method, we compute the implicit dependencies by statically analyzing the LLVM bit-code of the program before instrumenting the bit-code to add *self-logging* capabilities. Then, we execute the instrumented code to obtain the trace. As a result, the implicit dependencies will be captured in the execution trace as a special relation (DEP_{sta}). Static program analysis has a global view of the program and thus is well suited for computing the implicit dependencies. Inside LLVM, the bit-code is represented in a Single Static Assignment (SSA) format, meaning each variable is defined only once, which makes it possible to efficiently compute the implicit dependencies [20].

In addition to the implicit dependencies (DEP_{sta}) computed by static analysis, we also compute the explicit dependencies (DEP_{dyn}) based on the actual addresses appeared in the execution trace: for each memory address, instructions that use the address are checked to see if they have data hazards (RAW, WAR, or WAW). For instructions that have data hazards, their relative execution order during in-order execution cannot be violated; otherwise, the original program semantics may be changed.

Given both the statically computed DEP_{sta} and the dynamically computed DEP_{dyn}, we compute their transitive closure to obtain $DEP = (DEP_{sta} \cup DEP_{dyn})^*$, which represents the complete set of dependency constraints that must be respected at all time, to ensure that the out-of-order executions examined by our symbolic analysis are feasible.

The fact that static analysis is *conservative* in nature will not affect the correctness of our subsequent symbolic analysis. Since not all memory-addressing instructions can be statically resolved, as shown by the example instruction `store A[i]` in Fig. 2, static analysis may soundly over-approximate the possible dependencies of memory-related instructions. This is not a problem because it guarantees that, as long as two instructions are marked as *independent*, it is always safe to reorder these instructions during out-of-order execution. This is crucial for ensuring that leaks detected by our method are feasible.

4.2 Computing the Default Cache States

Given the in-order execution trace, we perform an in-order simulation to compute the default cache states, which will be used during our symbolic analysis of the out-of-order executions.

We regard the in-order execution trace as a sequence of instructions $\mathcal{T}_{ino} = \{I_1, \ldots, I_n\}$. The type of each instruction may be Load, Store, Symbolic Load, or Symbolic Store. Each Load/Store instruction is associated with an actual memory address. Each Symbolic Load/Store instruction is associated with a range of addresses that it may use.

Starting with an initial cache state S_0, we compute the sequence of cache states $\mathcal{T}_{cache} = \{S_0, S_{I_1} \ldots, S_{I_n}\}$ using the update function defined in Section 3.2. While the update function in Section 3.2 uses the LRU replacement policy, other cache replacement policies can also be implemented easily.

The result of in-order simulation will be given to our symbolic analysis, to examine the set of all possible out-of-order executions. Here, an out-of-order execution, denoted $\mathcal{T}_{ooe} = \{I'_1, ..., I'_n\}$, is a permutation of instructions of the in-order execution. That is, for all $1 \leq i \leq n$ and instruction $I_i \in \mathcal{T}_{ino}$, there exists $1 \leq j \leq n, i \neq j$ such that $I'_j \in \mathcal{T}_{ooe}$ and $I'_j = I_i$, and vice versa.

5 Analyzing the Out-of-Order Executions

In this section, we present our method for symbolically analyzing the out-of-order executions.

5.1 Symbolic Encoding

Our method uses a single logical formula (Φ) to encode the behaviors of all out-of-order executions of instructions within a sliding window of size N, together with the condition under which an out-of-order execution has secret-dependent, divergent cache behaviors. It guarantees that Φ is satisfiable if and only if there

exists such a side-channel leak in the sliding window of size N. Thus, when setting the value of N, there is a trade-off between coverage and scalability.

Before explaining how Φ is constructed from the in-order execution trace, however, we need to define the notations used in the symbolic encoding.

- **Sliding Window:** We focus on a sliding window of N instructions appeared in the in-order execution trace. Within this window, instructions may be executed in any order as long as they respect the DEP relation; outside of this window, instructions are executed in-order.
- **Program Counter:** We use $(N+1)$ variables $PC\_I_0, PC\_I_1, \ldots, PC\_I_N$ to represent the time when we execute the N instructions $I_1, \ldots, I_N$. The special variable PC_I_0 represents the start time, and each PC_I_i (where $1 \leq i \leq N$) represents the time immediately after I_i is executed.
- **Age of Address after Executing an Instruction:** We use $Age\_addr_k\_I_i$ to represent the cache line age of a memory block at $addr_k$ after we execute instruction I_i. Thus, for all memory addresses $addr_1, \ldots, addr_M$, we have integer variables $Age\_addr_1\_I_i, \ldots, Age\_addr_M\_I_i$ for all $0 \leq i \leq N$.

With these notations, we define the formula Φ as a conjunction of the following subformulas:

$$\Phi = \Phi_{pc} \wedge \Phi_{cs} \wedge \Phi_{ics} \wedge \Phi_{rep} \wedge \Phi_{dep} \wedge \Phi_{divc}$$

where Φ_{pc} is the program counter constraint, Φ_{cs} is the cache state constraint, Φ_{ics} is the initial cache state constraint, Φ_{rep} is the cache replacement constraint, Φ_{dep} is the dependency constraint, and Φ_{divc} is the divergence condition constraint.

Program Counter Constraint (Φ_{pc}) To get a total order of the N instructions, we require that, for all $0 \leq i \leq N$, the value of PC_I_i is unique; furthermore, we require $0 \leq PC\_I_i \leq N$. Thus, the constraint is defined as

$$\Phi_{pc} = \bigwedge_{0 \leq i \leq N} (0 \leq PC\_I_i \leq N) \wedge \bigwedge_{0 \leq i,j \leq N \text{ and } i \neq j} (PC\_I_i \neq PC\_I_j)$$

Cache State Constraint (Φ_{cs}) Let MAX be the cache's associativity, or the maximal number of cache lines that can be mapped to a memory address. After executing an instruction I_i, if $0 \leq Age\_addr_k\_I_i < MAX$, it means the memory block at $addr_k$ is inside the cache; but if $Age\_addr_k\_I_i = MAX$, it means the memory block is evicted out of the cache [2]. Thus, the constraint is defined as

$$\Phi_{cs} = \bigwedge_{0 \leq i \leq N \text{ and } 0 \leq k \leq M} (-1 \leq Age\_addr_k\_I_i \leq MAX)$$

[2] $Age\_addr_k\_I_i = -1$ means it has never been loaded to the cache yet.

Initial Cache State Constraint (Φ_{ics}) Before the first instruction is executed, the cache must be set to a proper initial state. In other words, variables $Age\_addr_1\_I_0, \ldots, Age\_addr_M\_I_0$ must be initialized based on the default cache states computed by in-order simulation (Section 4.2). Thus, the constraint is defined as

$$\Phi_{ics} = \bigwedge_{0 \le k \le M} (Age\_addr_k\_I_0 = init\_age\_addr_k)$$

Replacement Constraint (Φrep) Assuming that instruction I_j is immediately before I_i during an out-of-order execution, we define the cache line ages after executing I_i based on their ages after executing the predecessor instruction I_j. Let $addr_k$ be the address used by I_i, $addr_{k1}$ be any address whose age was younger than that of $addr_k$ immediately before executing I_i, and $addr_{k2}$ be any address whose age was older than that of $addr_k$. According to the update function defined in Section 3.2, we set $Age\_addr_k\_I_i$ to 0, set $Age\_addr_{k1}\_I_i$ to $(Age\_addr_{k1}\_I_j + 1)$, and set $Age\_addr_{k2}\_I_i$ to $Age\_addr_{k2}\_I_j$. Let the relation $UpdateRel(I_i, I_j)$ be the conjunction of the constraints defined above.

If a symbolic address (secret-dependent) is used by I_i, we encode it into the update relation as follows: for each concrete address that may be instantiated from the symbolic address, we construct an update relation $UpdateRel()$ under the assumption that it may be the actual address used by I_i.

Overall, the cache replacement constraint is defined as

$$\Phi_{rep} = \bigwedge_{0 \le i,j \le N \text{ and } i \ne j} (PC\_I_i = PC\_I_j + 1) \implies UpdateRel(I_i, I_j)$$

Dependency Constraint (Φ_{dep}) To ensure that out-of-order executions are feasible, we enforce the relative order of any two instructions if they have dependencies according to the DEP relation. Thus, the constraint is defined as

$$\Phi_{dep} = \bigwedge_{0 \le i,j \le N \text{ and } i \ne j \text{ and } DEP(I_i, I_j)} (PC\_I_i < PC\_I_j)$$

That is, if I_j depends on I_i, I_i must be executed before I_j.

Divergent Cache Constraint (Φ_{divc}) Let Var_s be a symbolic (secret) variable whose values include $v_1, v_2, \ldots$ and let I_i be a symbolic instruction whose actual addresses include $addr_{v_1}, addr_{v_2}, \ldots$ Here, the value v_1 corresponds to $addr_{v_1}$ and the value v_2 corresponds to $addr_{v_2}$. If accessing the memory block at $addr_{v_1}$ leads to a cache hit and accessing $addr_{v_2}$ leads to a cache miss (or vice versa), the target instruction I_i has divergent cache behaviors. Thus, the constraint is defined as

$$\Phi_{divc} = \bigvee_{\forall v_1, v_2} (0 \le Age\_addr_{v_1}\_I_i < MAX) \wedge (Age\_addr_{v_2}\_I_i \ge MAX)$$

Conjoining all of the subformulas defined above, we can construct the entire formula Φ which is satisfiable (SAT) if and only if there is a side-channel leak during one of the out-of-order executions.

5.2 The Overall Algorithm

The overall algorithm for predictive cache analysis is shown in Algorithm 1, which takes the in-order execution trace $\mathcal{T}_{ino} = \{I_1, \ldots, I_n\}$, the in-order cache state trace $\mathcal{T}_{cache} = \{S_0, \ldots, S_n\}$, and the sliding window size N as input. Internally, it uses a sliding window of N instructions, $\mathcal{T}_{window}$, to generate the SMT formula Φ. For this window, S_{init} is the initial cache state as computed by in-order simulation, and I_{target} is the target instruction. The formula Φ is satisfiable if and only if an out-of-order execution of the instructions within the window leads to divergent cache behaviors at the instruction I_{target}.

Algorithm 1 SymbolicCheck($\mathcal{T}_{ino}, \mathcal{T}_{cache}, N$) for predictive cache analysis.

1: **for** $pos \leftarrow 1$ to $(n - N)$ **do**
2: $first = (pos - N > 0)\ ?\ (pos - N) : 1$
3: $\mathcal{T}_{window} = \mathcal{T}_{ino}[first, pos]$
4: $I_{target} = \mathcal{T}_{ino}[pos]$
5: $S_{init} = \mathcal{T}_{cache}[first - 1]$
6: $\Phi =$ BuildFormula($\mathcal{T}_{window}, I_{target}, S_{init}$)
7: if ($SAT(\Phi) == true$) print LEAK_FOUND

Running Example We use the example code snippet in Fig. 2 to illustrate the symbolic encoding presented in this section. For this example, the in-order execution trace generated by our method is shown in the top half of Fig. 8. Note that `A` is marked as symbolic since `A[i]` is affected by the unknown variable `i`. The logical constraints are shown in the bottom half. Assume that the target instruction is $\mathtt{I}_4$, meaning that we want to construct a formula Φ to check if $\mathtt{I}_4$ has divergent cache behaviors.

The program counter and cache state constraints are shown in Lines 10-12; recall that each program counter variable must have a unique value. The dependency constraints are shown in Line 13. Then, in Line 14, we show the two symbolic variables used to check divergent cache behaviors; their values are in the range of the symbolic store in Line 5.

The update function for Instruction $\mathtt{I}_4$ starts from Line 15. If $\mathtt{v}_1$`==0x77ef5bd0`, which means `0x77ef5bd0` is used, the age after executing $\mathtt{I}_4$ is set to 0. The dependency relations indicate that $\mathtt{I}_5$ is allowed to execute before $\mathtt{I}_4$. From Line 16 to 18, we show an example update age constraints with program counter constraint and the condition which `Age_0x77ef5bd4_`$\mathtt{I}_4$ would increase by 1 from its predecessor $\mathtt{I}_5$ according to Section 5.1. Similarly, we encode other predecessors of $\mathtt{I}_4$ for the update function in Line 19. Finally, we encode the divergent cache constraint in Line 20.

```
In-Order Execution  Trace                                          Dependency relations:
I1: load  0x77ef5bd0 /*A[0]*/                                             |
I2: load  0x77ef5bd4 /*A[1]*/                                             V
I3: load  0x77ef5bd8 /*A[2]*/
I4: symbolic store 0x77ef5bd0, 0x77ef5bd4, 0x77ef5bd8 /*A[i]*/ <I1,I4> <I2,I4> <I3,I4>
I5: load  0x77ef5bdc /*B   */
Initialize Ages: Age_0x77ef5bd0_init == 0, Age_0x77ef5bd4_init == 0
                 Age_0x77ef5bd8_init == 0, Age_0x77ef5bdc_init == 0
PC Constraints:  1 ≤ PC_I1:5 ≤5, PC_I0 == 0, distinct(PC_Ii)
Age Constraints: -1 ≤ Age_0x77ef5bd0_Ii ≤ 3, -1 ≤ Age_0x77ef5bd4_Ii ≤ 3
                 -1 ≤ Age_0x77ef5bd8_Ii ≤ 3, -1 ≤ Age_0x77ef5bdc_Ii ≤ 3
DEP Constraints:  PC_I1 < PC_I4, PC_I2 < PC_I4, PC_I3 < PC_I4, PC_I0 < PC_I1:5
Symbolic Var:    v1/v2 ∈ {0x77ef5bd0, 0x77ef5bd4, 0x77ef5bd8}, v1 ≠ v2
Update Function: v1 == 0x77ef5bd0  ⟹  Age_0x77ef5bd0_I4 == 0
- I4.Pred is I5: (PC_I5 + 1 == PC_I4 ∧ Age_0x77ef5bd4_I5 > Age_0x77ef5bd0_I5
∧ Age_0x77ef5bd0_I5 ≠ -1 ∧ Age_0x77ef5bd4_I5 ≠ -1)
 ⟹   Age_0x77ef5bd4_I4 = Age_0x77ef5bd4_I5 + 1; ......
- I4.Pred is I1, I2, I3: ......
DivC Constraint:  Age_v1_I4 ≥ 3 ∧ Age_v2_I4 < 3 ∧ Age_v2_I4 ≠ -1
```

Fig. 8. An example encoding where the register variable i holds a secret value .

5.3 Optimizations of the Symbolic Encoding

Without optimization, the size of the formula Φ may be as large as $O(N^2M)$ in the worst case, where N is the number of instructions in the sliding window and M is the number of memory addresses used inside the window. In practice, however, many of the logical constraints can be skipped. Here, we propose two optimization techniques.

Skipping the Infeasible Cache Update Relations While constructing the constraints that update the cache states of the instructions, the default approach is to assume that, for any instruction I_i, any other instruction I_j in the same window may be executed immediately before I_i. This means it must construct N^2 update relations. However, due to the dependencies among instructions captured by the DEP relation, there may be many instruction pairs (I_j, I_i) such that I_j is not allowed to execute before I_i. By leveraging the information, we can skip many of these update relations.

Skipping the Unnecessary Φ_{divc} Constraints In many cases, by checking the initial cache state with respect to the sliding window of N instructions, we may be able to know that divergent cache behaviors are impossible during any of the out-of-order executions. In other words, Φ_{divc} is guaranteed to be unsatisfiable (UNSAT). Thus, we can avoid generating Φ. Toward this end, we check for the following two conditions, each of which is sufficient for Φ_{divc} to be UNSAT:

- *All ages are too young*: Inside the initial cache state (with respect to the window), if all cache line ages are less than $(MAX - M)$, where M is the number of unique addresses used in this window, we skip checking any of the instructions in this window for divergent cache behaviors. This is because the cache is large enough that, regardless of the execution order, none of the cache lines will be evicted.

Table 1. Statistics of the benchmark programs and the execution traces.

Name	Description	SLOC	Logged execution trace				
			Length	# Store	# Load	# Addr	#Cache-line
AES	Advanced Encryption Standard	2,077	32,069	8,753	23,316	3,126	139
DES	Archetypal block cipher	1,090	10,162	3,994	6,168	946	148
SEED	Symmetric key block cipher	720	20,820	6,999	13,821	2,044	130
Camellia	Symmetric key block cipher	555	14,595	5,487	9,108	1,63	130
Chacha20	Pseudorandom function based stream cipher	263	15,739	3,668	12,071	687	134
IDEA	International Data Encryption Algorithm	288	2,920	884	2,036	318	140
ARIA	Symmetric key block cipher	1,265	15,672	5,237	10,435	1,642	128
SM4	Symmetric key block cipher	301	11,362	3,410	7,952	1,412	131
MD5	MD5 message-digest algorithm	312	3,134	878	2,256	361	156
Blake2	Hash based on ChaCha stream cipher	512	4,832	1,363	3,469	309	163
SHA256	Secure Hash Algorithm standard	825	5,900	1,302	4,598	435	164
Whirlpool	Hash designed after Square block cipher	1,100	6,941	1,915	5,026	1,257	172

– *The age of addr accessed by the target instruction is too young*: Inside the initial cache state, if the age of *addr* is less than $(MAX - M)$, we skip checking this particular target instruction for divergent cache behaviors. This is because, regardless of the value of the secret variable, this particular cache line will never be evicted out of the cache.

6 Experiments

We have implemented our method in a tool named Spreca, which builds upon the LLVM compiler [17] and the Z3 SMT solver [19]. Specifically, it uses LLVM to implement the static analysis component, which takes a C program as input and computes the dependencies of memory-related instructions before instrumenting the LLVM bit-code; the instrumented bit-code, after compilation, is used to generate the execution trace at run time. We use Z3 to implement our symbolic analysis component, which takes the logged execution trace as input and generates SMT formulas of the cache states for leakage detection. Overall, our implementation includes 3.6K lines of C++ code inside LLVM for trace generation, SMT encoding and leakage detection, as well as 0.5K lines of Python/Bash script code for processing the trace files and automation. The archive is available at: https://doi.org/10.5281/zenodo.6117196.

6.1 Benchmarks

The benchmarks used to evaluate our tool are a set of C programs from `OpenSSL 1.1.1k` that implement well-known block-ciphers such as AES and DES and cryptographic hashing functions such as SHA256 and Whirlpool. The statistics of these benchmark programs are shown in Table 1, including the name of the program, a short description, the number of lines of C code, and statistics of the logged execution trace, which serves as input of our symbolic analysis method. For each execution trace, we show the trace length, the number of Store (ST) operations, the number of Load (LD) operations, the number of distinct memory locations touched by the execution, and the number of corresponding cache lines.

Our experiments were designed to answer the following questions:

Table 2. Results of our symbolic predictive analysis method for 8K fully associative cache, with LRU replacement policy, and window size set to 10.

Name	Trace length	SMT solver calls made			Leaking sites	Analysis time (s)
		total instances	SAT instances	UNSAT instances		
AES	32,069	0	0	0	0	246.0
DES	10,162	1	0	1	0	620.4
SEED	20,820	593	14	579	5	4,922.1
Camellia	14,595	366	15	351	6	2,475.1
Chacha20	15,739	0	0	0	0	4.9
IDEA	2,920	0	0	0	0	1.0
ARIA	15,672	1,060	0	1,060	0	8,760.2
SM4	11,362	27	0	27	0	788.1
MD5	3,134	0	0	0	0	1.2
Blake2	4,832	0	0	0	0	1.8
SHA256	5,900	0	0	0	0	2.4
Whirlpool	6,941	0	0	0	0	2.8

- Is our method effective in detecting OOE-related cache side-channel leaks?
- Is our method, based on symbolic analysis, more scalable than explicit analysis?

Toward this end, for each benchmark program, we applied our symbolic analysis method to check if it can find OOE-related cache side-channel leaks, i.e., leaks that otherwise would not show up unless out-of-order execution is considered. To evaluate the scalability of our method, we also compared it with a baseline explicit analysis method. Due to space limit, we omit the detailed algorithm of the explicit analysis method, which systematically enumerates the same set of out-of-order executions of instructions considered by our symbolic analysis method. Thus, both our symbolic method and the explicit method examine the same type of secret-dependent divergent cache behaviors, but they differ in efficiency and scalability.

6.2 Leakage Detection Results

Table 2 shows the results of our symbolic analysis method. These results were obtained using the following parameters: the cache has a total of 8K bytes, divided into 128 cache lines, with 64 bytes per cache line. The cache is fully associative, with the LRU replacement policy. The OOE window size is set to 10, meaning the number of Load/Store instructions that will be executed out of order is bounded to 10. Recall that inside the reorder buffer, there can be many non-memory instructions (e.g., arithmetic operations); thus, setting the window size to 10 is a reasonable choice. In this table, Columns 1-2 show the program name and the trace length. Columns 3-5 show the number of SMT solver calls, the number of satisfiable (SAT) instances, and the number of unsatisfiable (UNSAT) instances. Column 6 shows the number of leaking sites detected by our method and Column 7 shows the total analysis time in seconds.

Note that the number of SMT solver calls may be smaller than the number of instructions in the trace and, in many cases, is 0 because of the optimizations implemented during our symbolic encoding: for any instruction, if our simple

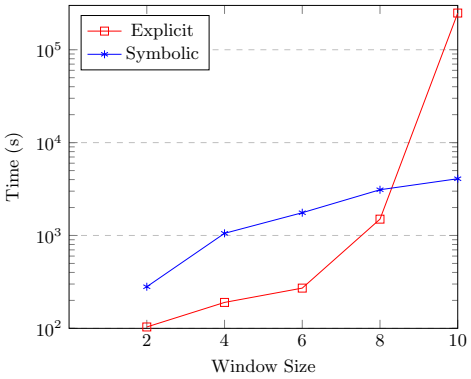


Fig. 9. Comparison of the analysis time: symbolic method versus explicit method.

checks reveal that no OOE-related divergent cache behavior is possible, we skip the more time-consuming SMT solver call. Also note that the number of leaking sites in Column 6, which are locations in the original C program, may be smaller than the number of UNSAT instances in Column 4; this is because multiple UNSAT results may be mapped to the same source code location.

To confirm that the leaking sites reported in Table 2 are indeed feasible (5 for SEED and 6 for Camellia), we manually inspected the source code and the LLVM bit-code of both SEED and Camellia. Our manual inspection shows that the reordered sequences provided by the SMT solver are indeed feasible as we check them against the source code. We also find that the divergent cache behaviors are real in that the two concrete values computed for each symbolic (sensitive) variable can indeed lead to a cache hit in one case but a cache miss in the other case.

6.3 Scalability Results

To evaluate the scalability of our symbolic analysis method, we compared its analysis time to that of the baseline explicit enumeration method. This experiment was conducted on SEED, with the OOE window size set to 2, 4, 6, 8 and 10, respectively. This is because the computational complexity of the problem increases exponentially as the OOE window size increases. The results are shown in Fig. 9, where the x-axis is the OOE window size and the y-axis is the analysis time in seconds. The blue line represents our symbolic method while the red line represents the explicit method.

The results in Fig. 9 show that, while our symbolic method has a higher fixed cost (associated with generating SMT formulas, calling the Z3 solver, and interpreting the results), and thus is slower than the explicit method when the OOE window size is smaller, it becomes significantly more efficient when the window size is larger. The figure also show that, as expected, the explicit method has an exponential blowup – its analysis time is actually worse than exponential

(factorial in the window size) – whereas the scalability of our symbolic method is significantly better.

7 Related Work

As we have mentioned earlier, the most closely related work is that of Guo et al. [10, 11] which relies on KLEE to detect cache side channels. However, their method only treats program input as symbolic, while still explicitly enumerating the out-of-order executions. Unlike their method, we analyze the set of all possible out-of-order executions symbolically by encoding them in a single logical formula to avoid the exponential blowup. In this sense, our method is the only predictive analysis method that can symbolically analyze the cache behaviors of out-of-order executions.

Besides our method and the method of Guo et al. [10, 11], there are many other techniques for analyzing cache side channels. Some of them use symbolic execution as well, e.g., to detect concurrency-related leaks [12] as well as leaks in sequential programs [3, 21, 29, 32]. Others use static analysis techniques including those based on abstract interpretation [6, 28, 30, 31]. In addition to leakage detection, there are techniques for leakage quantification [1, 2, 5, 7, 16] as well. However, none of these prior works considers out-of-order execution.

Beyond side-channel leakage detection and leakage quantification, cache analysis has been used in other applications such as estimating the worst-case execution time (WCET) of real-time software [9, 13, 25]. Beyond cache analysis, the idea of trace-based predictive analysis has been applied to multithreaded programs to detect concurrency bugs [8, 14, 22–24, 26, 27]. However, a crucial difference is that while concurrency bugs are violations of functional properties of a program, our method for side-channel analysis focuses exclusively on non-functional properties.

8 Conclusions

We have presented a symbolic method for analyzing the cache behaviors of out-of-order executions associated with an in-order execution trace. The method uses static analysis to compute dependencies before instrumenting the program to generate the in-order execution trace. Then, it uses an SMT solver based symbolic analysis to analyze the cache behaviors of all out-of-order executions. Our experiments on cryptographic software code show that the symbolic analysis method is effective in detecting OOE-related cache side-channel leaks and is significantly more scalable than explicit analysis. For future work, we plan to extend our method to detect side-channel leaks caused by other CPU-level optimizations.

Acknowledgements This work was partially funded by the U.S. National Science Foundation grants CNS-1722710 and CNS-1702824.

References

1. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: 30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA. pp. 141–153 (2009)
2. Bao, Q., Wang, Z., Li, X., Larus, J.R., Wu, D.: Abacus: Precise side-channel analysis. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. pp. 797–809 (2021)
3. Brotzman, R., Liu, S., Zhang, D., Tan, G., Kandemir, M.T.: CaSym: Cache aware symbolic execution for side channel detection and mitigation. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. pp. 505–521 (2019)
4. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 991–1008 (2018)
5. Chattopadhyay, S., Beck, M., Rezine, A., Zeller, A.: Quantifying the information leak in cache attacks via symbolic execution. In: Talpin, J., Derler, P., Schneider, K. (eds.) Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017. pp. 25–35 (2017)
6. Doychev, G., Feld, D., Köpf, B., Mauborgne, L., Reineke, J.: CacheAudit: A tool for the static analysis of cache side channels. IACR Cryptol. ePrint Arch. **2013**, 253 (2013)
7. Eldib, H., Wang, C., Taha, M.M.I., Schaumont, P.: QMS: evaluating the side-channel resistance of masked software from source code. In: The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014. pp. 209:1–209:6 (2014)
8. Ganai, M.K., Arora, N., Wang, C., Gupta, A., Balakrishnan, G.: BEST: A symbolic testing tool for predicting multi-threaded program failures. In: Alexander, P., Pasareanu, C.S., Hosking, J.G. (eds.) 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011. pp. 596–599 (2011)
9. Guan, N., Yang, X., Lv, M., Yi, W.: FIFO cache analysis for WCET estimation: a quantitative approach. In: Macii, E. (ed.) Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013. pp. 296–301 (2013)
10. Guo, S., Chen, Y., Li, P., Cheng, Y., Wang, H., Wu, M., Zuo, Z.: SpecuSym: speculative symbolic execution for cache timing leak detection. In: Rothermel, G., Bae, D. (eds.) ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020. pp. 1235–1247 (2020)
11. Guo, S., Chen, Y., Yu, J., Wu, M., Zuo, Z., Li, P., Cheng, Y., Wang, H.: Exposing cache timing side-channel leaks through out-of-order symbolic execution. Proc. ACM Program. Lang. **4**(OOPSLA), 147:1–147:32 (2020)
12. Guo, S., Wu, M., Wang, C.: Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (eds.) Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. pp. 377–388 (2018)

13. Huynh, B.K., Ju, L., Roychoudhury, A.: Scope-aware data cache analysis for WCET estimation. In: 17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011. pp. 203–212 (2011)
14. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 398–413 (2009)
15. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. pp. 1–19 (2019)
16. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 564–580 (2012)
17. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: International Symposium on Code Generation and Optimization. pp. 75–88. San Jose, CA, USA (2004)
18. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 973–990 (2018)
19. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340 (2008)
20. Novillo, D.: Memory SSA - a unified approach for sparsely representing memory operations (2007)
21. Pasareanu, C.S., Phan, Q., Malacaria, P.: Multi-run side-channel analysis using symbolic execution and Max-SMT. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016. pp. 387–400 (2016)
22. Roemer, J., Genç, K., Bond, M.D.: Smarttrack: efficient predictive race detection. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 747–762 (2020)
23. Said, M., Wang, C., Yang, Z., Sakallah, K.A.: Generating data race witnesses by an smt-based analysis. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 313–327 (2011)
24. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predicting serializability violations: SMT-based search vs. DPOR-based search. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7261, pp. 95–114 (2011)

25. Theiling, H., Ferdinand, C., Wilhelm, R.: Fast and precise WCET prediction by separated cache and path analyses. Real Time Syst. **18**(2/3), 157–179 (2000)
26. Wang, C., Kundu, S., Ganai, M.K., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 256–272 (2009)
27. Wang, C., Limaye, R., Ganai, M.K., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6015, pp. 328–342 (2010)
28. Wang, S., Bao, Y., Liu, X., Wang, P., Zhang, D., Wu, D.: Identifying cache-based side channels through secret-augmented abstract interpretation. In: Heninger, N., Traynor, P. (eds.) 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019. pp. 657–674 (2019)
29. Wang, S., Wang, P., Liu, X., Zhang, D., Wu, D.: Cached: Identifying cache-based timing channels in production software. In: Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 235–252 (2017)
30. Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: Tip, F., Bodden, E. (eds.) Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. pp. 15–26 (2018)
31. Wu, M., Wang, C.: Abstract interpretation under speculative execution. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 802–815 (2019)
32. Yarom, Y., Genkin, D., Heninger, N.: CacheBleed: a timing attack on openssl constant-time RSA. J. Cryptogr. Eng. **7**(2), 99–112 (2017)

PEQtest: Testing Functional Equivalence*

Marie-Christine Jakobs (✉) and Maik Wiesner

Technical University of Darmstadt, Department of Computer Science, Darmstadt, Germany
jakobs@cs.tu-darmstadt.de

Abstract. Refactoring a program without changing the program's functional behavior is challenging. To prevent that behavioral changes remain undetected, one may apply approaches that compare the functional behavior of original and refactored programs. Difference detection approaches often use dedicated test generators and may be inefficient (i.e., execute (some of) the non-modified code twice). In contrast, proving functional equivalence often requires expensive verification. Therefore, we propose PEQtest, which aims at localized functional equivalence testing thereby relying on existing tests or test generators. To this end, PEQtest derives a test program from the original program by replacing each code segment being refactored with program code that encodes the equivalence of the original and its refactored code segment. The encoding is similar to program encodings used by some verification-based equivalence checkers. Furthermore, we prove that the test program derived by PEQtest indeed checks functional equivalence. Moreover, we implemented PEQtest in a prototype and evaluate it on several examples. Our evaluation shows that PEQtest successfully detects refactored programs that change the program behavior and that it often performs better than the state-of-the-art equivalence checker PEQcheck.

1 Introduction

Developers refactor programs [16] to improve quality attributes like e.g. performance. For instance, a developer may parallelize a program with OpenMP [30] to improve performance. While a refactoring changes the program code, e.g., adds OpenMP pragmas, to improve the program's quality, the changes must not alter the program's functional behavior. To ensure that a refactored program is reliable, we must check that the refactoring preservers the functional behavior.

Various approaches exist that aim to safeguard refactored programs from altered behavior. In practice, developers often perform regression testing [54], but the success of detecting altered behavior depends on the test suite and its test oracle(s). If refactoring rules are applied, one can prove the correctness of the applied refactoring rules [45,22,44]. In contrast, incremental verification techniques, e.g., [53,39,8,35], propose solutions for efficient re-verification of changed programs,

* This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 184–204, 2022.
https://doi.org/10.1007/978-3-030-99429-7_11

Listing 1.1: Original program

```
void sum_seq(unsigned char N)
{
 int a[N+1];
 a[0] = 0;
 for(int i=1; i<=N;i++)
  a[i] = a[i-1] + i;
}
```

Listing 1.2: Refactored program

```
int sum_par(unsigned char N)
{
 int a[N+1];
 a[0] = 0;
#pragma omp parallel for
 for(int i=1; i <= N; i++)
  a[i] = (i*(i+1))/2;
}
```

Listing 1.3: Generated test program

```
int sum_test(unsigned char N)
{
 int a[N+1];
 a[0] = 0;

 store(a, 0);

 for(int i=1; i <= N; i++)
  a[i] = (i*(i+1))/2;

 store(a, 1);
 restore(a,0);

#pragma omp parallel for
 for(int i=1; i <= N; i++)
  a[i] = (i*(i+1))/2;

 store(a, 2);
 eq_store(a, 1, 2);
}
```

Fig. 1: Original, sequential program (top left), which initializes each array entry i with $\sum_{j=0}^{i} j$, the refactored program (bottom left), which parallelizes the array initialization using OpenMP and utilizing that $\sum_{j=0}^{i} j = \frac{i \cdot (i+1)}{2}$, as well as the generated program for testing functional equivalence (right)

but they typically need a specification of the functional behavior, which rarely exists. Another solution, which does not require a specification, is to inspect whether or when the original and the refactored program behave functionally equivalent. Approaches aiming to detect differences in the behavior [26,52,46,20,31,29,36,47] are inefficient, i.e., execute each test case on the original and the refactored program or function, and often use dedicated test generators. Approaches aiming to prove functional equivalence [5,56,40,14,13,43,49,41,34,4,15,17,38,23,42,19] use heavyweight verification techniques, rarely support parallel programs, and often consider all possible variables values.

Our goal is to develop a lightweight, test-based approach for functional equivalence checking, for which we can use existing tests or test generators. Inspired by equivalence checkers [17,38,23,51,42,2,19] that transform the equivalence of two programs into a set of verification tasks (i.e., programs with assertions), our PEQTEST approach transforms the equivalence of two programs into a test program. To restrict equivalence testing to relevant program values and to reduce the duplicate execution of non-modified code, PEQTEST generates a single test program (verification task) that executes the unchanged code only once and individually checks equivalence of each refactored code segment in the context of the original program. The individual checks use a similar idea as UC-KLEE [38], which verifies equivalence of functions. More concretely, PEQTEST derives the test program from the original program by extending each original code segment with (a) the refactored code segment and (b) code to store, restore, and compare

variable values of modified variables. To store, restore, and compare the values of modified variables, PEQTEST relies on checkpoints, which save the values of a given set of modified variables in a given program state.

In our example (Fig. 1), PEQTEST first detects that the original (sequential) code segment (framed, dark blue) and the refactored (parallelized) code segment (frameless, light blue) modify variable `a`[1]. Thereafter, PEQTEST derives the test program (right) from the original program (top left). It adds the parallelized code segment. To provide the same input to the original and refactored code segment, PEQTEST uses checkpoint 0 to store modified variables. The test program calls `store(a, 0);` to save in checkpoint 0 the values of modified variable `a`before the original code segment and calls `restore(a,0);` to restore the values of modified variables before the refactored code segment. To make the result of both code segments available for equivalence checking, the test program stores the values of modified variable `a` after each code segment in checkpoint 1 and 2, respectively. Finally, the equivalence test `eq_store` checks whether the checkpoints 1 and 2 contain equivalent values for the modified variable `a`.

We proved that PEQTEST generates test programs that can indeed detect inequivalence and that if no execution of the test program reveals an inequivalence, original and refactored program are equivalent. As a proof-of-concept, we implemented PEQTEST and used it to check several program parallelizations and a few sequential refactorings. Our evaluation shows that PEQTEST reliably detects inequivalences and typically outperforms the state-of-the-art equivalence checker PEQCHECK [19].

2 Background

Program Syntax. To present our approach, we rely on a simple imperative language on integer variables.[2] Since synchronization issues, e.g., deadlocks, do not affect how our approach works and we want to keep the programming language simple, our language supports parallel execution, but no synchronization operations. Below, we show the grammar of the programming language that we use to present our approach.

$$S := E \mid v :=_\ell aexpr; \mid \textbf{if}_\ell\ bexpr\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \mid \textbf{while}_\ell\ bexpr\ \textbf{do}\ S \mid S_1 S_2 \mid [S_1 \| \ldots \| S_n]$$

We use E to denote the empty program and assume that arithmetic expressions *aexpr* in assignments and Boolean expressions *bexpr* in if and while statements are built with standard operators on integers. To build more complex programs S, several subprograms S_i may be assembled into a sequence or into a parallel statement. To unambiguously identify the original and refactored code segments during test program generation[3] and any subprogram in our proofs,

[1] Both segments also modify variable `i`, but it is a local variable, which can be ignored.
[2] Our implementation supports a subset of C programs, which may use OpenMP pragmas for parallelization.
[3] For our implementation, one only needs to specify the start and end of code segments i.

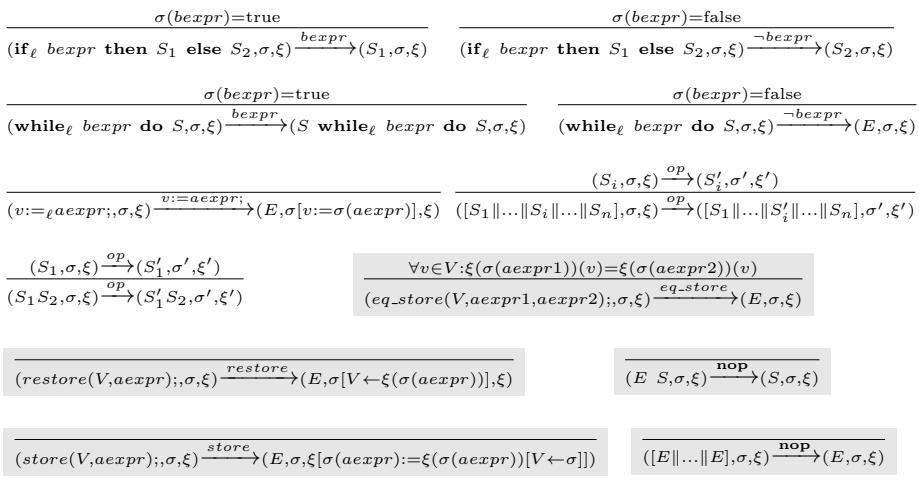


Fig. 2: Rules for operational semantics

we assume that each basic statement is annotated with a label ℓ, which must be unique in the complete program. Moreover, we use the set $\mathcal{V}$ to refer to all program variables and subset $\mathcal{V}(S) \subseteq \mathcal{V}$ to refer to the variables occurring in (sub)program S. Similarly, subset $\mathcal{V}(expr) \subseteq \mathcal{V}$ represents all variables that occur in an arithmetic or Boolean expression $expr$.

While the programming language above is sufficient to represent original and refactored programs, the test programs derived by our approach also use checkpointing to store, restore, and compare relevant parts (e.g., modified variables) of program states. To support checkpointing and checkpoint comparison, we extend the programming language for test programs with the three checkpoint functions `eq_store`, `restore`, and `store`. All three functions get as input a subset $V \subseteq \mathcal{V}$ of relevant variables and one or two arithmetic expressions (typically an integer constant) to refer to the relevant checkpoints.

$$S := eq\_store(V, aexpr1, aexpr2); \mid restore(V, aexpr); \mid store(V, aexpr);$$

Program Semantics We formalize the program semantics using a fairly standard operational semantics that defines how a program executes. A program execution is a sequence of transitions between execution states. An execution state is a triple of a program, a data state, and an additional checkpoint state. A data state is a function $\sigma : \mathcal{V} \to \mathbb{Z}$ that provides an integer value for each program variable. We denote the set of all data states by Σ. A checkpoint state is a function $\xi : \mathbb{N} \to \Sigma$ that maps checkpoints i to data states σ. The set Ξ denotes all checkpoint states.

The 12 rules shown in Fig. 2, which consists of 7 standard rules plus 5 newly introduced rules highlighted in light gray, define the possible transitions. As usual,

we write $\sigma(expr)$ for the evaluation of `expr` in data state $\sigma \in \Sigma$.[4] The state update $\sigma[v := \sigma(aexpr)]$, which is used in the rule for the assignment, returns a new data state σ_n with $\sigma_\mathrm{n}(w) = \sigma(w)$ for all $w \in \mathcal{V} \setminus \{v\}$ and $\sigma_\mathrm{n}(v) = \sigma(aexpr)$. Similarly, the multi state update $\sigma[V \leftarrow \sigma']$, which is used by the new store and restore rules, returns a new data state σ_n with $\sigma_\mathrm{n}(w) = \sigma(w)$ for all $w \in \mathcal{V} \setminus V$ and $\sigma_\mathrm{n}(v) = \sigma'(v)$ for all $v \in V$. In addition, the checkpoint update $\xi[c := \sigma_\mathrm{u}]$, which is used in the store rule, returns a new checkpoint state ξ_n with $\xi_\mathrm{n}(i) = \xi(i)$ for all $i \in \mathbb{N} \setminus \{c\}$ and $\xi_\mathrm{n}(c) = \sigma_\mathrm{u}$.[5] Also, note that instead of assuming that $E\ S$ and $[E\|\ldots\|E]S$ are equivalent to S, we introduce two nop rules, which make our proofs simpler. After we formalized the transitions, we now inductively define the executions $ex(S)$ of a program S with two inference rules:

1. $\frac{\sigma\in\Sigma,\xi\in\Xi}{(S,\sigma,\xi)\in ex(S)}$ and
2. $\frac{(S_0,\sigma_0,\xi_0)\xrightarrow{op_1}\ldots\xrightarrow{op_n}(S_n,\sigma_n,\xi_n)\in ex(S),\quad (S_n,\sigma_n,\xi_n)\xrightarrow{op_{n+1}}(S_{n+1},\sigma_{n+1},\xi_{n+1})}{(S_0,\sigma_0)\xrightarrow{op_1}\ldots\xrightarrow{op_n}(S_n,\sigma_n,\xi_n)\xrightarrow{op_{n+1}}(S_{n+1},\sigma_{n+1},\xi_{n+1})\in ex(S)}$.

We write $(S,\sigma,\xi) \to^* (S',\sigma',\xi')$ if the intermediate steps of the execution are unimportant. Furthermore, we say that execution $(S,\sigma,\xi) \to^* (S',\sigma',\xi')$ (i) terminates normally if $S' = E$ and (ii) violates a checkpoint equivalence if S' violates a checkpoint equivalence in (σ',ξ'). In general, a program S' violates a checkpoint equivalence in (σ',ξ') if either (a) there exists a statement $S_{eq} = eq\_store(V, aexpr1, aexpr2)$; such that $\exists v \in V : \xi'(\sigma'(aexpr1))(v) \neq \xi'(\sigma'(aexpr2))(v)$ and $S = S_{eq}$ or $S = S_{eq}S'$ or (b) $S = [S_1\|\ldots\|S_i\|\ldots\|S_n]$ or $S = [S_1\|\ldots\|S_i\|\ldots\|S_n]S'$ and there exists at least one subprogram S_i that violates a checkpoint equivalence in (σ',ξ'). In general, a program S violates a checkpoint equivalence if there exists an execution $(S,\sigma,\xi) \to^* (S',\sigma',\xi') \in ex(S)$ such that S' violates a checkpoint in (σ',ξ').

Partial Equivalence. We are interested whether two (sub)programs behave functionally equivalent, i.e., compute the same output when given the same input. Like many other approaches on equivalence checking, we focus on *partial equivalence*, i.e., we limit equivalence to executions that terminate normally.[6] In addition, we utilize that checkpoint functions are not used in programs, but are only introduced to test functional equivalence. Therefore, our definition of partial equivalence focuses on data states and ignores checkpoint states.

Definition 1. *(Sub)programs* $S1$ *and* $S2$ *are* partially equivalent *(*$S1 \equiv S2$*) if*

$$\forall\sigma,\sigma',\sigma'' \in \Sigma,\xi,\xi',\xi'',\xi''' \in \Xi : ((S1,\sigma,\xi) \to^* (E,\sigma',\xi') \in ex(S1) \\ \wedge (S2,\sigma,\xi'') \to^* (E,\sigma'',\xi''') \in ex(S2)) \implies \sigma' = \sigma'' \ .$$

[4] Note that we do not specify the expression evaluation in detail because we have not fixed the expression syntax. However, we assume that the result of evaluating integer constant c in data state σ is the constant c (i.e., $\sigma(c) = c$) and that expression evaluation is deterministic (i.e., $\sigma(expr) = x \wedge \sigma(expr) = y \implies x = y$).

[5] The store rule determines the state σ_u using a multi state update and the index c evaluating an arithmetic expression (often a constant) in the current data state.

[6] Note that we still may detect that a refactoring introduces non-termination because if a refactoring introduces non-termination, our test program either detects inequivalence or does not terminate for some inputs.

Variable Modification. To make equivalence testing more efficient, we only want to checkpoint modified variables, i.e., the checkpoint should only store the value of those variables whose value may change. The following definition formalizes the set of variables modified by a (sub)program.

Definition 2. *Let S be a (sub)program. The variables* modified *by S are:*

$$\mathcal{M}(S) := \{v \in \mathcal{V} \mid \exists \sigma, \sigma' \in \Sigma, \xi, \xi' \in \Xi : (S, \sigma, \xi) \rightarrow^* (\cdot, \sigma', \xi') \wedge \sigma(v) \neq \sigma'(v)\}.$$

For instance, in programs written in our programming language that do not use restore statements variables can only be modified by assignments. For those programs, the set $\mathcal{M}(S)$ of modified variables can be overapproximated by the set of variables that occur in S on the left-hand side of an assignment. In the following, we describe any overapproximation of the modified variables, e.g. the one sketched above, by $M_{\approx} : S \rightarrow 2^{\mathcal{V}}$ and assume that $\mathcal{M}(S) \subseteq M_{\approx}(S)$.

3 Generating Test Programs with PEQtest

Our goal is to test equivalence between an original and refactored program, which both do not use checkpoint functions. As explained earlier, checkpoint functions are supposed to be used by test programs only. In this section, we describe how PEQtest generates the test program for equivalence testing, prove soundness of the generated test program, i.e., show that the generated test program checks functional equivalence, and discuss limitations of PEQtest's program generation as well as our implementation.

Sound Test Program Generation. To test functional equivalence of two subprograms, the idea of our PEQtest approach is to execute both subprograms with the same input and compare their outputs. The test program generated by PEQtest will execute the two subprograms sequentially to avoid that their executions can interfere with each other. Furthermore, it will ensure that both subprograms get equal inputs, which may be produced by the (original) program, and that their outputs can be compared. Many verification approaches for functional equivalence [17,38,23,51,42,2,19] use a similar setup, but do not restrict the inputs. To ensure equal inputs and make outputs available, these approaches either (1) duplicate (shared, modified) variables, replace the variables in one of the subprograms by the duplicated ones, and assign equal values to the original and duplicated inputs [17,42,2,19], (2) add additional variables to store the input and output values and restore the input after the execution of the first subprogram [51,23], or (3) use dedicated functions, e.g., checkpoint functions, to store and restore inputs and outputs [38]. For our test program, we choose option (3) because it does not change the subprograms and, thus, it simplifies test program generation as well as it eases the comprehensibility of the test program.

Next, we discuss how we implement option (3). To lower the test effort, we decide to only store and compare values of variables that may be modified by one of the two subprograms. Since this set cannot always be determined precisely and different overapproximations are imaginable, we use parameter V to provide

this set to the test program generator. Moreover, we aim at localized equivalence testing. Thus, our test program likely includes more than one functional equivalence test, namely one for each pair of original and refactored subprogram. While the output must be stored directly after the execution of each subprogram, the output comparison can be done at the end of the test program or after the execution of original and refactored subprogram. We choose the second option because it allows us to reuse checkpoints and lets the test program stop at the first difference of outputs, which makes it easier to detect which pair of subprograms is responsible for the failure of the test program, i.e., which pair of subprograms is inequivalent. We stop at the first difference instead of e.g. logging the difference because test execution becomes faster, but we address the logging alternative when discussing the limitations. The following definition shows how we encode the functional equivalence test for an original subprogram $S1$ and the refactored subprogram $S2$ for a given overapproximation V of the set of modified variables.

$$\begin{aligned}&\mathit{test\_eq}(V, S1, S2) :=\\ &\mathit{store}(V,0);\ S1\ \mathit{store}(V,1); \mathit{restore}(V,0);\ S2\ \mathit{store}(V,2); \mathit{eq\_store}(V,1,2);\end{aligned}$$

Next, we show that our test encoding is sound, i.e., it may detect inequivalences if the two subprograms $S1$ and $S2$ are inequivalent. Our encoding uses checkpoint equivalence to detect whether two subprograms $S1$ and $S2$ are inequivalent, i.e., differ in their outputs. Hence, it must violate a checkpoint equivalence if $S1$ and $S2$ are inequivalent. We can ensure even more and show that the test encoding is also complete. As shown by the following theorem our test encoding violates a checkpoint equivalence if and only if $S1$ and $S2$ are inequivalent.

Theorem 1. *Let $S1$ and $S2$ be (sub)programs without calls to checkpoint functions and $M_{\approx}$ be an overapproximation of the modified variables. Then, $S1 \equiv S2$ iff $\mathit{test\_eq}(M_{\approx}(S1) \cup M_{\approx}(S2), S1, S2)$ does not violate a checkpoint equivalence.*

Proof (Sketch). Let $M := M_{\approx}(S1) \cup M_{\approx}(S2)$.

$\Rightarrow$ Let $(\mathit{test\_eq}(M, S1, S2), \sigma, \xi) \rightarrow^* (\mathit{eq\_store}(M,1,2);, \sigma_5, \xi_5)$ be arbitrary. Show with semantics that there exists an execution
$(\mathit{test\_eq}(M, S1, S2), \sigma, \xi)$
$\rightarrow (S1\ \mathit{store}(M,1); \mathit{restore}(M,0);\ S2\ \mathit{store}(M,2); \mathit{eq\_store}(M,1,2);, \sigma_1, \xi_1)$
$\rightarrow^* (\mathit{store}(M,1); \mathit{restore}(M,0);\ S2\ \mathit{store}(M,2); \mathit{eq\_store}(M,1,2);, \sigma_2, \xi_2)$
$\rightarrow^* (S2\ \mathit{store}(M,2); \mathit{eq\_store}(M,1,2);, \sigma_3, \xi_3)$
$\rightarrow^* (\mathit{store}(M,2); \mathit{eq\_store}(M,1,2);, \sigma_4, \xi_4)$
$\rightarrow (\mathit{eq\_store}(M,1,2);, \sigma_5, \xi_5)$
with $\sigma = \sigma_1 = \sigma_3$, for all $v \in \mathcal{V} \setminus M$ also $\sigma(v) = \sigma_5(v)$, and for all $v \in M$ we have $\xi_5(1)(v) = \sigma_2(v)$ and $\xi_5(2)(v) = \sigma_4(v)$.
Conclude that exists $(S1, \sigma_1, \xi_1) \rightarrow^* (E, \sigma_2, \xi_2)$ and $(S2, \sigma_3, \xi_3) \rightarrow^* (E, \sigma_4, \xi_4)$ with $\sigma = \sigma_1 = \sigma_3$ and for all $v \in M$ we have $\xi_5(1)(v) = \sigma_2$ and $\xi_5(2)(v) = \sigma_4$. By assumption ($S1 \equiv S2$), $\sigma_2 = \sigma_4$ and, thus, $\xi_5(1)(v) = \sigma_2(v) = \sigma_4(v) = \xi_5(2)(v)$. By semantics, $(\mathit{test\_eq}(M, S1, S2), \sigma, \xi) \rightarrow^* (\mathit{eq\_store}(M,1,2);, \sigma_5, \xi_5)$ does not violate a checkpoint equivalence.

$\Leftarrow$ Let $(S1, \sigma_1, \xi_1) \rightarrow^* (E, \sigma_2, \xi_2)$ and $(S2, \sigma_3, \xi_3) \rightarrow^* (E, \sigma_4, \xi_4)$ be arbitrary with $\sigma_1 = \sigma_3$. Show with semantics that there exists an execution

$(test\_eq(M, S1, S2), \sigma, \xi)$
$\rightarrow (S1\ store(M,1); restore(M,0);\ S2\ store(M,2); eq\_store(M,1,2);, \sigma_1, \xi_1)$
$\rightarrow^* (store(M,1); restore(M,0);\ S2\ store(M,2); eq\_store(M,1,2);, \sigma_2, \xi_2)$
$\rightarrow^* (S2\ store(M,2); eq\_store(M,1,2);, \sigma_3, \xi_3)$
$\rightarrow^* (store(M,2); eq\_store(M,1,2);, \sigma_4, \xi_4)$
$\rightarrow (eq\_store(M,1,2);, \sigma_5, \xi_5)$
with $\sigma = \sigma_1 = \sigma_3$, for all $v \in \mathcal{V} \setminus M$ also $\sigma(v) = \sigma_5(v)$, and for all $v \in M$ we have $\xi_5(1)(v) = \sigma_2(v)$ and $\xi_5(2)(v) = \sigma_4(v)$.
Since the test program does not violate a checkpoint equivalence, for all $v \in M$ we know $\sigma_2(v) = \xi_5(1)(v) = \xi_5(2)(v) = \sigma_4(v)$. We conclude that $\sigma_2 = \sigma_4$. □

So far, we can use the test encoding to test or even verify functional equivalence of complete programs. Following the idea of PEQCHECK [19], which checks equivalence on the level of subprograms rather than on the level of functions or programs, our goal is to split testing of equivalence into multiple subtests, namely one subtest per pair of original and refactored subprogram. While PEQCHECK builds one equivalence task per pair and verifies all tasks on every input, our PEQTEST approach generates one single test program that only provides inputs produced by the original program[7]. More concretely, PEQTEST derives the test program from the original program by replacing the subprograms being refactored with the test encoding *test_eq* of the original and refactored subprogram.

Currently, we assume that PEQTEST is informed about the refactored subprograms. More concretely, given original program S and refactored program S', we assume that there exists a partial, injective *replacement function* $\gamma : 2^S \rightharpoonup 2^S$[8] such that S' can be derived from S by replacing all subprograms S_1 of S with $S_1 \in preImg(\gamma)$ by $\gamma(S_1)$. Generally, we write $S2 = \Gamma(S1, \gamma)$ to denote that $S2$ is derivable from $S1$ by replacing all subprograms S_s of $S1$ by $\gamma(S_s)$. For the PEQTEST approach, we assume that the replacement function γ only describes the refactoring of the original program S, i.e., $preImg(\gamma)$ only contains subprograms of S. In addition, the replacement must be unambiguous. Hence, we do not allow $S_1, S_2 \in preImg(\gamma)$ such that S_2 is a subprogram of S_1 nor $S_1, S_1S_2 \in img(\gamma)$ such that S_1 is a subprogram of S and $S_1 \notin preImg(\gamma)$.[9] We also require that $E, [E\| \ldots \|E] \notin (preImg(\gamma) \cup img(\gamma))$ and $\neg\exists S : E\ S, [E\| \ldots \|E]S \in (preImg(\gamma) \cup img(\gamma))$ because they are no proper programs. To avoid that interference of parallel statements can invalidate the result of a test, all subprograms in $preImg(\gamma)$ ($img(\gamma)$) must not occur in a parallel statement of the original (refactored) program. Thus, a refactoring in a parallel statement must be described by a refactoring of the parallel statement. Note that for proper programs one can always use $\gamma = \{(S, S')\}$.[10]

To generate our test program, PEQTEST requires a replacement function γ_{test} that maps the subprograms being refactored to their test encodings. PEQTEST

[7] If all original and refactored subprograms are equivalent (which we aim to inspect), the original and refactored program will provide the same inputs.
[8] If γ is not injective, one can make it injective by properly changing statement labels.
[9] One can achieve this by proper choices of code segments and statement labels.
[10] However, one may need to adapt some of the labels in S'.

derives γ_{test} from the replacement function γ, which describes the refactoring. For each subprogram in the domain, PEQTEST replaces its image (the refactored subprogram) by the test encoding of that subprogram and its refactored subprogram thereby using an $M_{\approx}$ to determine the set of modified variables.

$$\gamma_{\text{test}}(\gamma, M_{\approx}) := \{(S1, \mathit{test\_eq}(M_{\approx}(S1) \cup M_{\approx}(\gamma(S1)), S1, \gamma(S1))) \mid S1 \in \mathit{preImg}(\gamma)\}$$

Let us briefly discuss why γ_{test} fulfills the requirements on a replacement function. Since the test encoding contains $\gamma(S1)$, function γ_{test} inherits injectivity from γ. By construction, test encodings are unequal to E, $E\ S$, $[E\| \ldots \|E]$, and $[E\| \ldots \|E]S$ and start with checkpoint functions, which we assume that the original program does not contain. The remaining requirements are fulfilled because we only replace refactored subprograms by the corresponding test encoding.

Now, we have everything at hand to generate the test program, which can then be used to detect inequivalences with an existing test approach, e.g., [12,1]. As explained, we derive the test program from the original program by replacing the subprograms being refactored with the test encoding *test_eq* of the original and refactored subprogram. To achieve this, we use the replacement function γ_{test}.

$$\mathit{test\_prog}(S, \gamma, M_{\approx}) := \Gamma(S, \gamma_{\text{test}}(\gamma, M_{\approx}))$$

Again, let us consider soundness, but now for the test program. Our goal is to detect inequivalences caused by a refactoring. Thus, we do not give any guarantees if the original program is non-deterministic, i.e., not equivalent to itself, which can only occur if it contains non-deterministic parallel statements or checkpoint functions. We already assumed that checkpoint functions are only used by the test program, but not by the original or refactored program. For our soundness discussion, we also exclude programs that contain non-replaced, non-deterministic parallel statements. More concretely, we assume that all parallel statements S_p that are not replaced, i.e., for whom there does not exist a subprogram $S_s \in \mathit{preImg}(\gamma)$ such that $S_p = S_s$ or S_p is a subprogram of S_s, are deterministic ($S_p \equiv S_p$). In this case, the following theorem ensures that our PEQTEST approach can soundly detect inequivalences, i.e., the test program generated by PEQTEST is able to detect a violation of a checkpoint equivalence if original and refactored program are inequivalent.

Theorem 2. *Let S and S' be programs without calls to checkpoint functions, $M_{\approx}$ an overapproximation of the modified variables, γ be a replacement function such that $S' = \Gamma(S, \gamma)$, and all non-replaced parallel statements S_p of S are deterministic ($S_p \equiv S_p$). If $S \not\equiv S'$, then there exists $(S_0, \sigma_0, \xi_0) \rightarrow^* (S_n, \sigma_n, \xi_n) \in \mathit{ex}(\mathit{test\_prog}(S, \gamma, M_{\approx}))$ that violates a checkpoint equivalence.*

Finally, let us look at the contraposition of the above theorem. While our intention for PEQTEST is testing and detection of equivalence violations, the corollary below states that we can alternatively verify the test program generated by PEQTEST to show functional equivalence.

Listing 1.4: Original program

```
void swapi_orig(int x, int y)
{
  tmp=y+1;
  y=x;
  x=tmp;
}
```

Listing 1.5: Refactored program

```
void swapi_mod(int x, int y)
{
  tmp=y;
  y=x;
  x=tmp+1;
}
```

Fig. 3: Behaviorally equivalent original and refactored program whose code segments are not equivalent

Corollary 1. *Let S and S' be programs without calls to checkpoint functions, $M_{\approx}$ an overapproximation of the modified variables, γ be a replacement function such that $S' = \Gamma(S, \gamma)$, and all non-replaced parallel statements S_p of S are deterministic ($S_p \equiv S_p$). If no execution $(S_0, \sigma_0, \xi_0) \rightarrow^* (S_n, \sigma_n, \xi_n) \in ex(test\_prog(S, \gamma, M_{\approx}))$ violates a checkpoint equivalence, then $S \equiv S'$.*

Discussion of Limitations. Functional equivalence of two programs is undecidable [17]. While our PEQTEST approach is sound under certain assumptions. PEQTEST may report violations of checkpoint equivalences, although original and refactored program are equivalent. Hence, it may be incomplete. One reason is the wrong choice of code segments. For example, consider Fig. 3. Although the two code segments of original and refactored program (highlighted in blue and green, respectively) are inequivalent, the programs are equivalent. For our experiments, we ensured that we do not make the wrong choice for the code segments. In practice, one may check whether a reported violation is a false alarm caused by a wrong choice of code segments by reusing the test input causing the violation to execute one or more test programs generated by PEQTEST that use the same original and refactored program but larger segments, e.g., using segments on function or program level, or iteratively merging segments until the violation is disproved or the segments become the programs.

Next, let us discuss the assumption used in Theorem 2. One can easily get rid of the assumption that non-replaced parallel statements must be deterministic. Basically, PEQTEST needs to extend γ with pairs (S_p, S_p) for all non-replaced parallel statements S_p. Supporting checkpoint functions is more challenging because PEQTEST must be able to store and restore checkpoints and it must ensure that its checkpoints and the program's checkpoints do not interfere. While one may find such an encoding, our definition of partial equivalence does not cover checkpoint states. Also, it does not support non-deterministic programs since our main motivation for PEQTEST is refactoring or parallelization of sequential programs not the refactoring of non-deterministic, parallel programs. To properly support checkpointing and all kinds of parallel programs, our definition of equivalence and PEQTEST need to be adapted significantly.

Also, the requirements on the replacement function restrict our PEQTEST approach. While many assumptions can be met by adapting labels of statements,

the requirement that code segments must be subprograms and they must not occur in a parallel statement are major restrictions. However, note that this only limits the granularity of code segments, but not the applicability of the approach.

Finally, we want to mention that in our above formalization we chose to stop as soon as PEQTEST finds a violation because it simplified our proofs. To always inspect all refactored code segments, one can either move PEQTEST's checks at the end of the test program and use different checkpoints per test encoding, or only write a log but do not stop when detecting a difference. To ensure that one still tests on values of the original program, one must restore the output of the original program at the end of each test encoding or swap $S1$ and $S2$ in the test encoding *test_eq*, i.e., execute the refactored subprogram $S2$ before the original subprogram $S1$. Our current implementation postpones PEQTEST's checks to the end of the test program and restores the output of the original program at the end of each test encoding.

Implementation. We support test program generation for a subset of C programs with or without OpenMP directives. So far, we do not support programs with pointer aliasing (except for parameter passing). While we allow pointers and dynamic memory allocation, we do not support the modification of dynamic data structures in original or refactored code segments. The reason is that we checkpoint arrays and structs by recursively checkpointing their elements and checkpoint pointers by dereferencing them and then checkpoint the dereferenced non-pointer element. Thus, our current implementation only works correctly in case that pointers that need to be checkpointed are non-null and do not change in original or refactored code segments.[11]

Our test program generation relies on the ROSE compiler framework [37]. To store and restore checkpoints, we use a minicpr library, but we built our own library to compare checkpoints. Our implementation assumes that the start and end of a code segment i is specified by pragma statements `#pragma scope_i` and `#pragma epocs_i`. Currently, we insert them manually. For OpenMP parallelization (our main field of application), insertion is mostly straightforward. Often, choosing the code blocks associated with the outermost OpenMP directives is a good choice. This can easily be automated, but has not been implemented yet.

For each code segment, our implementation runs ROSE's definition-use analysis to detect the modified variables $M_{\approx}$ that are visible after the code segment. If a code segment contains procedure calls, we also add all global variables and all variables occurring in the parameter expression of a pointer or array argument to the modified variables $M_{\approx}$. Based on the computed set $M_{\approx}$ of modified variables, we then extend the sequential code segment with the refactored code segment and the calls to the checkpoint library necessary to store and restore checkpoints. In contrast to our formalization, the store and restore operation only get the checkpoint name, while additional calls are used to inform the checkpoint library which variables V to consider. Also note that the test program generated by our implementation stores the output of the original and refactored code segments

[11] Due to internals of the used checkpoint library, pointers must not change after they are first checkpointed.

in checkpoints that differ for each execution of a test encoding and performs output comparison at the end of the test program, which allows us to inspect all checkpoints at once and to possibly find multiple violations.

Next, we describe the checkpoint comparison. For each variable in the two checkpoints[12], we check whether their content is equivalent. Except for floating point values, we rely on C's byte level comparison function `memcmp`. Often, implementations of floating point operations like + are not associative, but small differences of floating point values are tolerable. Thus, our comparison of floating point values succeeds when the difference of the values is within a tolerance ε[13].

4 Evaluation

The goals of our experiments are to (a) study how effective and efficient is PEQTEST's detection of inequivalences and to (b) compare PEQTEST to an existing equivalence checker. For our comparison, we choose PEQCHECK because it also supports localized checking for OpenMP programs.

4.1 Experimental Setup

Benchmark. To check equivalence of sequential and parallelized programs, we use the tasks from the DataRaceBench (DRB) benchmark suite [24,50] (version 1.3.2), which addresses common mistakes in OpenMP parallelization and contains OpenMP programs with and without data races. From the DataRaceBench, we exclude all tasks with thread private directives, which we cannot cover with our segments and all tasks that require at least an OpenMP 4.5 compiler or that offload computation to a different device (i.e., use the `target` construct) because they are neither supported by PEQTEST nor PEQCHECK. In total, we get 132 tasks (26 equivalent and 106 inequivalent tasks). We manually selected the code segments following the idea discussed in the implementation paragraph and use the DataRaceBench programs without OpenMP constructs for the sequential (original) programs. To execute the generated test programs, we use the inputs provided by DataRaceBench.

To check equivalence of two sequential program versions, we consider all non-recursive programs from Rêve [15]. However, we exclude loop4 and loop5, which were not available, as well as digits10, digits!10, and barthe2, which declare different sets of output variables in original and refactored program and, thus, are detected inequivalent during test program generation. To make the programs executable, we remove the mark annotations, which have no implementation, and extend each of the programs with a test driver that generates random inputs. The code segments are the same as in the evaluation of PEQCHECK [19]. In total, we get 15 sequential tasks (5 equivalent and 10 inequivalent tasks).

Tool Configurations. To study the trade-off between effectiveness and efficiency, we examine three PEQTEST configurations, which differ in the resources

[12] By construction, checkpoints that are compared store the same variables.
[13] In our evaluation, we use $\varepsilon = 10^{-8}$.

used during test program execution. The low effort configuration uses one thread and runs the test program once. The other two configurations use two threads for the DRB tasks and one thread for the sequential tasks while running the test program 10- and 100-times. For the competitor PEQcheck [19], we use a setup similar to [19]. For the DRB tasks, PEQcheck combines the PEQcheck encoding[14] (revision 9dc36b) and verifier CIVL [42] (version 1.20_5259) using the theorem prover Z3 [27] (version 4.8.7). We restrict CIVL to two threads, set its timeout to 5 min, and disable the division by zero and memory leakage checks. For the sequential tasks, PEQcheck combines the PEQcheck encoding with verifier CPAchecker [7] (version 2.0). For verification, we use CPAchecker's default analysis, which is also limited to 5 min.

Environment. We use a time limit of 5 min per task and run our experiments on an Ubuntu 20.04 machine with an Intel Core i7 (1.8 GHz) and 32 GB of RAM.

4.2 Experiments

RQ 1: How effective is PEQtest with minimal resources? To answer this research question, we look at PEQtest's results for the low effort configuration (1 thread, 1 run). For the DataRaceBench (DRB) tasks (left) and the sequential (SEQ) tasks (right), Tab. 1 shows for all three PEQtest configurations the absolute and relative number[15] of correctly detected inequivalences, the number of missed inequivalences, i.e., inequivalences that are not detected, the number of equivalent tasks for which an inequivalence is incorrectly detected (i.e., the false alarms), and the number of equivalent tasks for which no inequivalence is detected. For the two classes in which no inequivalence is detected (missed inequivalence or correctly detected no inequivalence), we also distinguish between the two reasons for not detecting inequivalences: (1) no inequivalences are reported during test program execution and (2) task not completed, e.g., test program generation failed or a timeout occurred during test program generation or execution.

Looking at the first two columns of the DRB tasks and the two columns of the SEQ tasks in Tab. 1, which show the results of the low effort configuration, we observe that for our examples PEQtest does not report any false alarms, i.e., the number of incorrectly detected inequivalences is zero. Thus, we have 100 % precision for inequivalence detection. More surprisingly, PEQtest detects more than half of the inequivalences (i.e., recall $> 50\,\%$) with its low effort configuration and, thus, without parallel execution in case of the parallelized DRB tasks. Studying the detected inequivalences, we observe that almost all the detected inequivalent DRB tasks use a variable to which data-sharing attribute (first)private is assigned and that is visible, but typically not live after the parallelized code segment. The data-sharing attribute makes the variable thread-local during execution of the parallelized code segment and prevents that the thread-local variable values become available after the parallelized code segment.

[14] https://git.rwth-aachen.de/svpsys-sw/FECheck

[15] The relative numbers are the absolute numbers divided by the total number of equivalent and inequivalent tasks, respectively.

Table 1: For each of the three PEQtest configurations, shows for the DRB and sequential (SEQ) tasks the absolute and relative number of tasks for which inequivalence is detected correctly, is missed, is detected incorrectly, and is correctly not detected. If no inequivalence is detected, the table also distinguishes between no inequivalence reported (i.e., no inequivalence observed in runs) and task is not completed due to a timeout or failure.

	DRB tasks						SEQ tasks	
	1 thread		2 threads				1 thread	
	1 run		10 runs		100 runs		1/10/100 runs	
correctly detected inequivalence	58	(55 %)	72	(68 %)	**74**	**(70 %)**	6	(60 %)
missed inequivalence	48	(45 %)	34	(32 %)	**32**	**(20 %)**	4	(40 %)
no inequivalence reported	38	(35 %)	24	(22 %)	**22**	**(17 %)**	3	(30 %)
task not completed	10	(10 %)	10	(10 %)	10	(10 %)	1	(10 %)
incorrectly reported inequivalence	0	(0 %)	0	(0 %)	0	(0 %)	0	(0 %)
correctly detected no inequivalence	26	(100 %)	26	(100 %)	26	(100 %)	5	(100 %)
no inequivalence reported	22	(85 %)	22	(85 %)	22	(85 %)	4	(80 %)
task not completed	4	(15 %)	4	(15 %)	4	(15 %)	1	(20 %)

Furthermore, many of the detected inequivalent sequential tasks are inequivalent for many different input values. We conclude that inequivalences caused by the discussed data-sharing attributes or input-insensitive inequivalences can easily be detected with a single run and thread.

RQ 2: Does PEQtest's effectiveness increase when given more resources and what are the costs? First, we examine whether PEQtest performs better if we increase the resources for testing, i.e., the number of runs and for parallelized programs also the number of threads used during test program execution. Comparing the results of our three PEQtest configurations (Tab. 1), we observe that there is no difference for the sequential tasks. The reason is that one can only detect the missed inequivalences with particular inputs whose random generation is unlikely. For the DRB tasks, however, the number of correctly detected inequivalences increases and the number of missed inequivalences decreases when providing more resources. All other entries stay the same. Hence, PEQtest's effectiveness may increase (i.e., its recall increases) when we allow it to use more resources. Especially, using more than one thread for parallelized programs increases the effectiveness significantly, as one could have expected. For our examples, using 100 instead of 10 runs hardly improves PEQtest's effectiveness. In general, PEQtest misses inequivalences in the DRB tasks if the generation of the test program fails (10 tasks). In addition, it misses inequivalences for SIMD constructs (2 tasks), inequivalences depending on thread scheduling (13 tasks), and inequivalences in I/O behavior (7 tasks), e.g., values written via `printf`, which our implementation does not support yet[16].

[16] Support for I/O can be added by writing all outputs to the checkpoint.

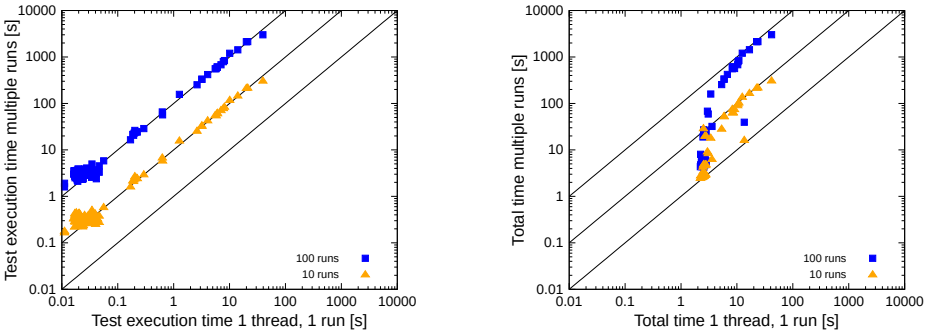


Fig. 4: Per task compare execution time of all test program runs (left) and total runtime of PEQTEST (right) in low effort configuration (1 thread, 1 run) against the other two configurations (2 threads for DRB tasks and 1 thread for sequential tasks, and 10 (▲) or 100 (■) runs)

Second, we examine the costs for increasing PEQTEST's resources for test program execution. To this end, we look at the execution times PEQTEST consumes for all test program runs and the total execution time (test program generation and execution). Figure 4 compares for each task that does not belong to one of the task not completed categories the times for the low effort configuration (1 thread, 1 run, x-axis) with the other two configurations of PEQTEST. As one could have expected, the scatter plot on the left-hand side of Fig. 4 shows that the execution times for the test programs scale linearly with the number of runs. A similar behavior can often be observed when the total time of the low effort configuration is not dominated by the test program generation ($> 3\,s$).

In summary, providing more resources often increases PEQTEST's effectiveness while causing at most a linear increase of runtime costs. In particular for parallelized tasks, using more than one thread is beneficial. However, we require many runs of the generated test program to find schedule-dependent or input-sensitive inequivalences.

RQ 3: How does PEQtest compare against state-of-the-art? We compare PEQTEST's configuration using 100 runs with equivalence checker PEQCHECK [19], which also performs localized checks, but relies on verification. Since PEQTEST's and PEQCHECK's definition of functional equivalence differ (PEQTEST considers all variables, while PEQCHECK only considers live variables), we restrict the comparison of PEQTEST and PEQCHECK to those 72 DRB tasks and 8 sequential tasks that (1) are either equivalent or inequivalent for both notions of equivalence and (2) in which the code segments affect at least one variable that is live afterwards.

Table 2 shows the results of PEQTEST and PEQCHECK on the restricted benchmark. The structure of Tab. 2 is the same as Tab. 1. Looking at Tab. 2, we first observe that both approaches do not incorrectly detect an inequivalence, i.e., they do not report false alarms. Hence, the precision for inequivalence detection is

Table 2: For PEQtest and PEQcheck, shows for the DRB and sequential (SEQ) tasks the absolute and relative number of tasks for which inequivalence is detected correctly, is missed, is detected incorrectly, and is correctly not detected. If no inequivalence is detected, also distinguishes between no inequivalence reported and task is not completed due to a timeout or failure.

	PEQtest 100 runs				PEQcheck			
	DRB		SEQ		DRB		SEQ	
correctly detected inequivalence	**39**	**(74 %)**	2	(67 %)	3	(6 %)	**3**	**(100 %)**
missed inequivalence	**14**	**(26 %)**	1	(%)	50	(94 %)	**0**	**(0 %)**
no inequivalence reported	11	(21 %)	0	(33 %)	**1**	**(2 %)**	0	(0 %)
task not completed	**3**	**(5 %)**	1	(33 %)	49	(92 %)	**0**	**(0 %)**
incorrectly detected inequivalence	0	(0 %)	0	(0 %)	0	(0 %)	0	(0 %)
correctly detected no inequivalence	19	(100 %)	5	(100 %)	19	(100 %)	5	(100 %)
no inequivalence reported	15	(79 %)	4	(80 %)	**5**	(26 %)	4	(80 %)
task not completed	**4**	(21 %)	1	(20 %)	14	(74 %)	1	(20 %)

100 %. For the sequential tasks, PEQcheck detects one additional inequivalent task, for which PEQtest times out. In contrast, PEQtest detects significantly more inequivalent DRB tasks (i.e., has a higher recall) and, thus, misses less inequivalent DRB tasks. An important reason for the lower recall of PEQcheck is that PEQcheck's inspection fails in 87.5 % of the DRB tasks. The major failure causes are timeouts (30 %), missing support for OpenMP constructs in the verifier CIVL (31 %), and the detection of violations that are unrelated to functional equivalence, e.g., array out of bounds accesses in a verification task, which is generated by PEQcheck to check functional equivalence. Despite PEQcheck's worse performance, it can verify the task `DRB076-flush-orig-no.c`, for which PEQtest failed. Finally, we remark that although PEQtest has a higher time limit than PEQtest (namely, 5 min per run instead of 5 min per verification task), there exist only two tasks in which PEQtest requires more than 5 min in total and PEQcheck could have profited from a higher time limit.

Summing up, PEQtest is typically a better choice than PEQcheck when aiming to find inequivalences. In particular, PEQtest profits from relying on compiler support of OpenMP constructs and from checking equivalence only for the test inputs. Thus, PEQtest is well-suited for inequivalence detection, but in contrast to PEQcheck, which considers all inputs, it rarely proves equivalence.

5 Related Work

Approaches inspecting functional equivalence aim at proving equivalence or detecting behavioral differences. Alternatively, they characterize for which inputs equivalence is ensured.

Proving Functional Equivalence. Approaches proving functional equivalence may use relational verification [6,5],(bi)simulation relations [56,40,14,13], or domain-specific checks [55,9,10,18,25]. Other approaches transform the programs into models and check model equivalence [43,49,41]. ARDiff [4] compares symbolic summaries and Rêve [15] translates the equivalence into Horn constraints. Several approaches [17,38,23,51,42,2,19] encode equivalence checking into programs. Their encoding idea is similar to PEQTEST's encoding of the functional equivalence tests. The closest encoding is the encoding of UCKLEE [38], which also use checkpointing, while the other approaches duplicate variables. Despite similar encodings, these approaches do not test, but verify the generated programs. A further difference is that they typically generate more than one program, namely one per changed unit (program [42], function [17,38,23,51], or refactored code segment [2,19]). Each generated program only consists of the functional equivalence check of the respective unit and typically considers all possible inputs. In contrast, PEQTEST embeds the equivalence tests into the original program and only considers inputs produced by the original program.

Difference Detection. Relative debugging [3] executes the original and refactored program in parallel and compares the values of user-defined variables or data structure at user-defined program locations, which is more fine-grained than functional equivalence. Nevertheless, several techniques focus on detecting differences of the functional behavior. Differential monitoring [28] applies runtime monitoring that runs two programs, e.g., original and refactored program, in parallel, distributes any input to both programs, compares their outputs, and forwards equivalent outputs to the environment, while aborting in case of inequivalence. Following the idea of differential testing [26], BERT [20], shadow symbolic execution [31], and HyDiff [29] generate tests and execute the generated tests on original and refactored program to detect differences in the behavior. BERT [20] generates inputs to cover the changed code parts. Shadow symbolic execution [31] uses a more advanced test generation that is steered towards internal behavioral differences. HyDiff [29] combines shadow symbolic execution with fuzzing, using the tests from the shadow symbolic execution to steer the fuzzer AFL. In contrast, Qi et al. [36] and eXpress [47] directly aim at generating difference revealing tests. To this end, they steer the test generation to find test inputs that reach a change that affects the output. While the previous techniques use special test generators, Diffut [52] and DiffGen [46] rely on standard test generators. Diffut keeps shadow variables for the original program in the refactored program, wraps the method of the original program to extend it with equivalence checks, and uses JML annotations to force the execution of the wrapped method of the original program while testing the refactored program. DiffGen [46] generates one test driver per changed method that copies the input, executes original and refactored method with original and copied input, respectively, and contains one check per output. DiffGen's encoding idea is similar to PEQTEST's encoding of functional equivalence tests, but PEQTEST focuses on refactored code segments.

Semantic Characterization of Differences. To provide more information in case of non-equivalence, a few approaches compute or (under)approximate the

condition when original and refactored program are equivalent. To this end, they use symbolic execution [34,48], abstract interpretation [21,32,33], or testing [11].

6 Conclusion

While refactorings are necessary to improve software quality, correct refactoring, i.e., a refactoring that does not change the functional behavior of the software, is challenging. Several solutions have been proposed to detect that refactored programs alter the behavior, some of them compare the functional behavior of original and refactored programs.

Approaches checking functional equivalence often use heavyweight (formal) verification. Furthermore, difference detection approaches frequently use dedicated test generators and execute (some of) the non-modified code twice, once for the original and once for the refactored program (function). To overcome these restrictions, we propose PEQtest, which can be used to test (the intended application) or to formally verify functional equivalence. The test program generated by PEQtest—for which we proved that it checks functional equivalence—allows us to rely on compiler support, e.g., for OpenMP, to reuse existing tests or test generators, and at the same time to utilize that refactorings are often local, thus, avoiding to execute non-modified code more than once in each test program execution. To this end, PEQtest replaces each refactored code segment in the program, e.g., a parallelized code segment, by a local check that inspects the equivalence of the corresponding original and refactored code segment.

We implemented PEQtest and evaluated it with the DataRaceBench benchmark suite and sequential refactorings already used to evaluate other functional equivalence checkers. Our experiments show that PEQtest detects many of the inequivalent tasks, e.g., incorrectly parallelized tasks, using a limited amount of resources, while reporting no false alarm. A comparison with the state-of-the-art equivalence checker PEQcheck reveals that PEQtest often performs better.

References

1. Technical "whitepaper" for afl-fuzz, http://lcamtuf.coredump.cx/afl/technical_details.txt (last accessed 2022-01-19)
2. Abadi, M., Keidar-Barner, S., Pidan, D., Veksler, T.: Verifying parallel code after refactoring using equivalence checking. Int. J. Parallel Program. **47**(1), 59–73 (2019)
3. Abramson, D., Foster, I.T., Michalakes, J., Sosič, R.: Relative debugging and its application to the development of large numerical models. In: Proc. SC. p. 51. ACM (1995)
4. Badihi, S., Akinotcho, F., Li, Y., Rubin, J.: ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In: Proc. FSE. pp. 13–24. ACM (2020)
5. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Proc. FM. pp. 200–214. LNCS 6664, Springer (2011)
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proc. POPL. pp. 14–25. ACM (2004)

7. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011)
8. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013)
9. Blom, S., Darabi, S., Huisman, M.: Verification of loop parallelisations. In: Proc. FASE. pp. 202–217. LNCS 9033, Springer, Berlin (2015)
10. Blom, S., Darabi, S., Huisman, M., Safari, M.: Correct program parallelisations. STTT (2021)
11. Böhme, M., d. S. Oliveira, B.C., Roychoudhury, A.: Partition-based regression verification. In: Proc. ICSE. pp. 302–311. IEEE (2013)
12. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
13. Churchill, B.R., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proc. PLDI. pp. 1027–1040. ACM (2019)
14. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Proc. APLAS. pp. 127–147. LNCS 10695, Springer (2017)
15. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proc. ASE. pp. 349–360. ACM (2014)
16. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison-Wesley (1999)
17. Godlin, B., Strichman, O.: Regression verification. In: Proc. DAC. pp. 466–471. ACM (2009)
18. Jakobs, M.C.: PatEC: Pattern-based equivalence checking. In: Proc. SPIN. LNCS 12864, Springer, Cham (2021)
19. Jakobs, M.C.: PEQcheck: Localized and context-aware checking of functional equivalence. In: Proc. FormaliSE. pp. 130–140. IEEE (2021)
20. Jin, W., Orso, A., Xie, T.: Automated behavioral regression testing. In: Proc. ICST. pp. 137–146. IEEE (2010)
21. Kawaguchi, M., Lahiri, S., Rebelo, H.: Conditional equivalence. Tech. rep., Microsoft Research (2010)
22. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Proc. PLDI. pp. 327–337. ACM (2009)
23. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In: Proc. CAV. pp. 712–717. LNCS 7358, Springer (2012)
24. Liao, C., Lin, P.H., Asplund, J., Schordan, M., Karlin, I.: DataRaceBench: A benchmark suite for systematic evaluation of data race detection tools. In: Proc. SC. pp. 11:1–11:14. ACM (2017)
25. Malík, V., Vojnar, T.: Automatically checking semantic equivalence between versions of large-scale C projects. In: Proc. ICST. pp. 329–339. IEEE (2021)
26. McKeeman, W.M.: Differential testing for software. Digital Technical Journal **10**(1), 100–107 (1998)
27. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. TACAS. pp. 337–340. LNCS 4963, Springer (2008)
28. Muehlboeck, F., Henzinger, T.A.: Differential monitoring. In: Proc. RV. pp. 231–243. LNCS 12974, Springer (2021)
29. Noller, Y., Pasareanu, C.S., Böhme, M., Sun, Y., Nguyen, H.L., Grunske, L.: HyDiff: Hybrid differential software analysis. In: Proc. ICSE. pp. 1273–1285. ACM (2020)
30. OpenMP: OpenMP application programming interface (version 5.1). Tech. rep., OpenMP Architecture Review Board (2020)

31. Palikareva, H., Kuchta, T., Cadar, C.: Shadow of a doubt: Testing for divergences between software versions. In: Proc. ICSE. pp. 1181–1192. ACM (2016)
32. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: Proc. SAS. pp. 238–258. LNCS 7935, Springer (2013)
33. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: Proc. OOPSLA. pp. 811–828. ACM (2014)
34. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: Proc. FSE. pp. 226–237. ACM (2008)
35. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: Proc. PLDI. pp. 504–515. ACM (2011)
36. Qi, D., Roychoudhury, A., Liang, Z.: Test generation to expose changes in evolving programs. In: Proc. ASE. pp. 397–406. ACM (2010)
37. Quinlan, D., Liao, C.: The ROSE source-to-source compiler infrastructure. In: Cetus users and compiler infrastructure workshop, in conjunction with PACT. vol. 2011, pp. 1–3. Citeseer (2011)
38. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Proc. CAV. pp. 669–685. LNCS 6806, Springer (2011)
39. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: Proc. FMCAD. pp. 114–121. IEEE (2012)
40. Sharma, R., Schkufza, E., Churchill, B.R., Aiken, A.: Data-driven equivalence checking. In: Proc. OOPSLA. pp. 391–406. ACM (2013)
41. Shashidhar, K.C., Bruynooghe, M., Catthoor, F., Janssens, G.: Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In: Proc. DATE. pp. 1310–1315. IEEE (2005)
42. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: the concurrency intermediate verification language. In: Proc. SC. pp. 61:1–61:12. ACM (2015)
43. Siegel, S.F., Zirkel, T.K.: TASS: the toolkit for accurate scientific software. Mathematics in Computer Science **5**(4), 395–426 (2011)
44. Steinhöfel, D.: REFINITY to model and prove program transformation rules. In: Proc. APLAS. pp. 311–319. LNCS 12470, Springer (2020)
45. Sultana, N., Thompson, S.J.: Mechanical verification of refactorings. In: Proc. PEPM. pp. 51–60. ACM (2008)
46. Taneja, K., Xie, T.: DiffGen: Automated regression unit-test generation. In: Proc. ASE. pp. 407–410. IEEE (2008)
47. Taneja, K., Xie, T., Tillmann, N., de Halleux, J.: eXpress: Guided path exploration for efficient regression test generation. In: Proc. ISSTA. pp. 1–11. ACM (2011)
48. Trostanetski, A., Grumberg, O., Kroening, D.: Modular demand-driven analysis of semantic difference for program versions. In: Proc. SAS. pp. 405–427. LNCS 10422, Springer (2017)
49. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. In: Proc. CAV. pp. 599–613. LNCS 5643, Springer (2009)
50. Verma, G., Shi, Y., Liao, C., Chapman, B.M., Yan, Y.: Enhancing DataRaceBench for evaluating data race detection tools. In: Proc. Correctness@SC. pp. 20–30. IEEE (2020)
51. Wood, T., Drossopoulou, S., Lahiri, S.K., Eisenbach, S.: Modular verification of procedure equivalence in the presence of memory allocation. In: Proc. ESOP. pp. 937–963. LNCS 10201, Springer (2017)

52. Xie, T., Taneja, K., Kale, S., Marinov, D.: Towards a framework for differential unit testing of object-oriented programs. In: Proc. AST. pp. 17–23. IEEE (2007)
53. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: Proc. ICSM. pp. 115–124. IEEE (2009)
54. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verification Reliab. **22**(2), 67–120 (2012)
55. Yu, F., Yang, S., Wang, F., Chen, G., Chan, C.: Symbolic consistency checking of OpenMP parallel programs. In: Proc. LCTES. pp. 139–148. ACM (2012)
56. Zaks, A., Pnueli, A.: CoVaC: Compiler validation by program analysis of the cross-product. In: Proc. FM. pp. 35–51. LNCS 5014, Springer (2008)

An Institutional Approach to Communicating UML State Machines

Tobias Rosenberger[1,2], Alexander Knapp[3](✉), and Markus Roggenbach[1]

[1] Swansea University, Swansea, U. K.
{t.rosenberger.971978, m.roggenbach}@swansea.ac.uk
[2] VERIMAG, Université Grenoble Alpes, Grenoble, France
[3] Universität Augsburg, Augsburg, Germany
knapp@informatik.uni-augsburg.de

Abstract We present a new approach on how to provide institution-based semantics for communicating UML state machines in form of a hybrid modal logic $\mathcal{M}_{\mathcal{D}}^{\downarrow}$. A theoroidal comorphism maps $\mathcal{M}_{\mathcal{D}}^{\downarrow}$ into the CASL institution. This allows for symbolic reasoning on communicating UML state machines.

1 Introduction

In line with a long-standing line of research [5,6,15,4], we set out on a general programme to bring together multi-view system specification with UML diagrams and heterogeneous specification and verification based on institution theory, giving the different system views both a joint semantics and richer tool support. Institutions, a formal notion of a logic, are a principled way of creating such joint semantics. They make moderate assumptions about the data constituting a logic, give uniform notions of well-behaved translations between logics and, given a graph of such translations, automatically give rise to a joint institution.

UML state machines are an object-based variant of Harel statecharts. Within the UML, state machines are a central means to specify system behaviour. In previous work [16], an institutional semantics for UML state machines was provided that allowed for symbolic reasoning. Such symbolic reasoning can be of advantage as, in principle, it allows to verify properties of UML state machines with large or infinite state spaces. Here, we extend this work in order to cater for communication.

A typical scenario for such communication is the interaction between a `User`, an `ATM`, and a `Bank` in order to authenticate the `User` as legitimate owner of a bank card by checking an entered PIN. Figure 1 depicts a UML modelling for this scenario. In brief, the system consists of the `ATM` and the `Bank`, where we consider `User` interaction as an external communication. The scenario begins with the `User` entering a bank card and a PIN. The `ATM` requests their verification by the `Bank`. The `Bank` checks validity of the card/PIN combination and communicates the result to the `ATM`. We model the validity check as internal, non-deterministic choice made by the `Bank`. In case of a positive result, the `ATM` will return the card to the `User`. In case of a negative result, the `User` is given a

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 205–224, 2022.
https://doi.org/10.1007/978-3-030-99429-7_12

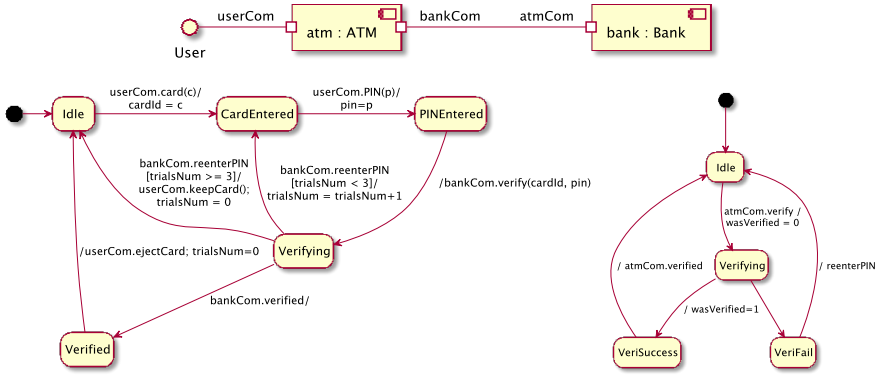


Figure 1. UML diagrams for the `ATM` example (implicit completion events omitted): Composite structure diagram: top; state machine: left `ATM`, right `Bank`.

second and third chance to enter a correct PIN. After the third verification failure, the `ATM` will keep the card. A typical question on this model is whether the `ATM` will consider the verification successful only if the `Bank` has already come to the same conclusion. To answer such questions, one needs to take into account the behaviour of all state machines involved as well as how they can communicate via the ports and connectors as specified by a composite structure diagram.

Closest to our approach are the works [6,4]. Both these papers address the topic of communicating state machines, however, both fail to provide institutions of state machines as reported in [15,16]. Learning from the reason for this shortcoming, rather than capturing UML state machines directly as an institution, [16] builds up a new logic in which UML state machines can be embedded. Here, we extend this logic for communication. In particular, we treat UML event pools as part of composite structure diagrams rather than of state machines. State machines are seen as a completely open system, which is (partially) closed by 'wiring up' in a communication structure. Overall, this leads to a separation of concerns: event pools and transitions can be analysed independently.

A number of authors give formal semantics to communicating state machines, however with a purpose different from symbolic analysis of UML. The Object Management Group provides an executable semantics of UML Composite Structures [14]. Their objective is to provide an interpreter for the executable subset fUML of the UML. Dragomir [12] define transformations from composite structure diagrams to communicating extended timed automata for the purpose of simulation, static analysis and model-checking. Mazzanti et al. [8] provide a UML model checker that also covers composite structure diagrams. A quite comprehensive formal semantics has been provided by Liu et al. [7], again with the main purpose of supporting model checking.

In Section 2, we recall the notion of an institution and sketch the CFOL$^{=}$ institution of CASL, which we use for specifying data. In Section 3, we extend the hybrid modal logic $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ [16] to cater also for output by adding the notion of messages (in [16] with input only). For structures and formulae this requires us to introduce relativisations with

regards to a set of outputs. We show that the extended logic $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ is an institution, can be embedded into CASL via a theoroidal comorphism, and allows for "borrowing" of CASL theorem proving support. In Section 4, we show how to embed simple UML state machines with output into the extended logic $\mathcal{M}^{\downarrow}_{\mathcal{D}}$. In Section 5, we provide an institution for simple UML composite structures by enriching our extended logic $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ with elements capturing connectors and event queues. Again, "borrowing" of CASL theorem proving support is possible. Finally, in Section 6, we demonstrate that our approach allows for automated theorem proving.

2 Background on Institutions and CASL

Institutions are an abstract formalisation of the notion of logical systems combining signatures, structures, sentences, and satisfaction under the slogan "truth is invariant under change of notation" [3]. Institutions can be related in different ways by institution (forward) (co-)morphisms, where a so-called theoroidal institution comorphism covers a particular case of encoding a "poorer" logic into a "richer" one. The algebraic specification language CASL [11] uses an institution of first-order logic at its basic specification level, where mainly signature items and axiom sentences are listed. On its structured specifications level, CASL offers institution-independent combination mechanisms to build larger specifications in a hierarchical and modular fashion. We use CASL's basic institution CFOL$^{=}$ of first-order logic with equality and sort generation constraints [9] and construct a theoroidal institution comorphism from our hybrid modal logic institution $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ to CFOL$^{=}$.

2.1 Institutions and Theoroidal Institution Comorphisms

An *institution* $\mathcal{I} = (\mathbb{S}^{\mathcal{I}}, \mathit{Str}^{\mathcal{I}}, \mathrm{Sen}^{\mathcal{I}}, \models^{\mathcal{I}})$ consists of (i) a category of *signatures* $\mathbb{S}^{\mathcal{I}}$; (ii) a contravariant *structures functor* $\mathit{Str}^{\mathcal{I}} \colon (\mathbb{S}^{\mathcal{I}})^{\mathrm{op}} \to \mathrm{Cat}$, where Cat is the category of (small) categories; (iii) a *sentence functor* $\mathrm{Sen}^{\mathcal{I}} \colon \mathbb{S}^{\mathcal{I}} \to \mathrm{Set}$, where Set is the category of sets; and (iv) a family of *satisfaction relations* $\models^{\mathcal{I}}_{\Sigma} \subseteq |\mathit{Str}^{\mathcal{I}}(\Sigma)| \times \mathrm{Sen}^{\mathcal{I}}(\Sigma)$ indexed over $\Sigma \in |\mathbb{S}^{\mathcal{I}}|$, such that the following *satisfaction condition* holds for all $\sigma \colon \Sigma \to \Sigma'$ in $\mathbb{S}^{\mathcal{I}}$, $\varphi \in \mathrm{Sen}^{\mathcal{I}}(\Sigma)$, and $M' \in |\mathit{Str}^{\mathcal{I}}(\Sigma')|$:

$$\mathit{Str}^{\mathcal{I}}(\sigma)(M') \models^{\mathcal{I}}_{\Sigma} \varphi \iff M' \models^{\mathcal{I}}_{\Sigma'} \mathrm{Sen}^{\mathcal{I}}(\sigma)(\varphi) \,.$$

$\mathit{Str}^{\mathcal{I}}(\sigma)$ is called the *reduct* functor, $\mathrm{Sen}^{\mathcal{I}}(\sigma)$ the *translation* function.

A *theory presentation* $T = (\Sigma, \Phi)$ in the institution $\mathcal{I}$ consists of a signature $\Sigma \in |\mathbb{S}^{\mathcal{I}}|$, also denoted by $\mathit{Sig}(T)$, and a set of sentences $\Phi \subseteq \mathrm{Sen}^{\mathcal{I}}(\Sigma)$. Its *model class* $\mathrm{Mod}^{\mathcal{I}}(T)$ is the class $\{M \in \mathit{Str}^{\mathcal{I}}(\Sigma) \mid M \models^{\mathcal{I}}_{\Sigma} \varphi \text{ f. a. } \varphi \in \Phi\}$ of the Σ-structures satisfying the sentences in Φ. A *theory presentation morphism* $\sigma \colon (\Sigma, \Phi) \to (\Sigma', \Phi')$ is given by a signature morphism $\sigma \colon \Sigma \to \Sigma'$ such that $M' \models^{\mathcal{I}}_{\Sigma'} \mathrm{Sen}^{\mathcal{I}}(\sigma)(\varphi)$ for all $\varphi \in \Phi$ and $M' \in \mathrm{Mod}^{\mathcal{I}}(\Sigma', \Phi')$. Theory presentations in $\mathcal{I}$ and their morphisms form the category $\mathrm{Pres}^{\mathcal{I}}$.

A *theoroidal institution comorphism* $\nu = (\nu^{\mathit{Sig}}, \nu^{\mathrm{Mod}}, \nu^{\mathrm{Sen}}) \colon \mathcal{I} \to \mathcal{I}'$ consists of a functor $\nu^{\mathit{Sig}} \colon \mathbb{S}^{\mathcal{I}} \to \mathrm{Pres}^{\mathcal{I}'}$ inducing the functor $\nu^{\mathbb{S}} = \nu^{\mathit{Sig}}; \mathit{Sig} \colon \mathbb{S}^{\mathcal{I}} \to \mathbb{S}^{\mathcal{I}'}$ on

signatures, a natural transformation $\nu^{\mathrm{Mod}} \colon (\nu^{Sig})^{\mathrm{op}}; \mathrm{Mod}^{\mathcal{I}'} \dot{\to} Str^{\mathcal{I}}$ on models and structures, and a natural transformation $\nu^{\mathrm{Sen}} \colon \mathrm{Sen}^{\mathcal{I}} \to \nu^{\mathbb{S}}; \mathrm{Sen}^{\mathcal{I}'}$ on sentences, such that for all $\Sigma \in |\mathbb{S}^{\mathcal{I}}|$, $M' \in |\mathrm{Mod}^{\mathcal{I}'}(\nu^{Sig}(\Sigma))|$, and $\varphi \in \mathrm{Sen}^{\mathcal{I}}(\Sigma)$ the following *satisfaction condition* holds:

$$\nu^{\mathrm{Mod}}_{\Sigma}(M') \models^{\mathcal{I}}_{\Sigma} \varphi \iff M' \models^{\mathcal{I}'}_{\nu^{\mathbb{S}}(\Sigma)} \nu^{\mathrm{Sen}}(\Sigma)(\varphi) \,.$$

A theory presentation (Σ, Φ) over the institution $\mathcal{I}$ is *translated* via a theoroidal institution comorphism $\nu \colon \mathcal{I} \to \mathcal{I}'$ into the theory presentation $\nu^{\mathrm{Pres}}(\Sigma, \Phi) = (\Sigma_\nu, \Phi_\nu \cup \nu^{\mathrm{Sen}}_{\Sigma}(\Phi))$ over $\mathcal{I}'$ where $\nu^{Sig}(\Sigma) = (\Sigma_\nu, \Phi_\nu)$ and $\nu^{\mathrm{Sen}}_{\Sigma}(\Phi) = \{\nu^{\mathrm{Sen}}_{\Sigma}(\varphi) \mid \varphi \in \Phi\}$.

2.2 CASL and the Institution CFOL$^{=}$

At the level of basic CASL specifications, CFOL$^{=}$ offers declarations of *sorts*, *operations*, and *predicates* with given argument and result sorts. Formally, this defines a *many-sorted signature* $\Sigma = (S, F, P)$ with a set S of sorts, a $S^* \times S$-sorted families $F = (F_{w,s})_{w\, s \in S^+}$ of *total function symbols*, and family $P = (P_w)_{w \in S^*}$ of *predicate symbols*. Using these symbols, one may then write axioms in first-order logic with equality. Moreover, one can specify *data types*, each given by a list of data constructors and, optionally, selectors. Data types may be declared to be *generated* or *free*. Generatedness amounts to an implicit higher-order induction axiom and intuitively states that all elements of the data types are reachable by constructor terms ("no junk"); freeness additionally requires that all these constructor terms are distinct ("no confusion"). Basic CASL specifications denote the class of all algebras which fulfil the declared axioms, i.e., CASL has loose semantics. More formally, for CFOL$^{=}$ a *many-sorted Σ-structure* M consists of a non-empty carrier set s^M for each $s \in S$, a total function $f^M \colon w^M \to s^M$ for each function symbol $f \in F_{w,s}$ and a predicate p^M for each predicate symbol $p \in P_w$. A *many-sorted Σ-sentence* is a closed many-sorted first-order formula over Σ or a sort generation constraint.

3 The Hybrid Modal Logic $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ for Event/Data Systems

The logic $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ is a hybrid modal logic for specifying and reasoning about event/data-based reactive systems. The modal part of the logic allows to handle transitions between system configurations where the modalities describe guarded configuration moves based on input and output events with arguments, i.e., messages, and the corresponding effects on data. The hybrid part of the logic allows to bind control states of system configurations and to jump to configurations with such control states explicitly. $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ with its signatures, sentences, and structures forms an institution. Furthermore, $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ can be translated into CASL via a theoroidal institution comorphism.

We extend the logic and the comorphism of [16] by including output. A modal formula $\langle\!| i : \phi \mathbin{/\!/} [O]_N : \psi |\!\rangle \varrho$ now says that in the current configuration an input message according to i can be accepted if precondition state predicate ϕ holds and that, in response, output messages according to $[O]_N$ and satisfying the transition predicate ψ can be produced such that ϱ holds afterwards. The messages frame $[O]_N$ tells that besides outputs from O also additional messages according to N can be sent. This relativisation allows $\mathcal{M}^{\downarrow}_{\mathcal{D}}$

to specify the "cone of messages above O" in a finite and, in particular, institution-compatible way that also is extensible into a theoroidal institution comorphism from $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ to CASL. We furthermore demonstrate that for pure $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-invariants the comorphism leads to simpler CASL proof obligations that are easier to automate in theorem proving.

For the inclusion of *data* in $\mathcal{M}^{\downarrow}_{\mathcal{D}}$, we assume given a consistent, monomorphic CASL specification Dt. The interpretation of the sorts $S(Dt)$ of Dt represents the different kinds of data, like the integers or lists of integers. Requiring Dt to be monomorphic fixes these carrier sets as there is, up to isomorphism, a single model $\mathcal{D}$ of Dt. We also use open formulæ $\mathscr{F}^{\text{CASL}}_{Sig(Dt),X}$ over sorted variables $X = (X_s)_{s\in S(Dt)}$ and their satisfaction relation $\mathcal{D}, \beta \models^{\text{CASL}}_{Sig(Dt),X} \varphi$ for a variable valuation $\beta\colon X \to \mathcal{D}$, i.e., $\beta = (\beta_s\colon X_s \to s^{\mathcal{D}})_{s\in S(Dt)}$.

3.1 Data States and Transitions

A *data signature* A consists of a finite set of *attributes* $|A|$ and a sorting $s(A)\colon |A| \to S(Dt)$. A *data signature morphism* from a data signature A to a data signature A' is a function $\alpha\colon |A| \to |A'|$ such that $s(A)(a) = s(A')(\alpha(a))$ for all $a \in |A|$. We sometimes identify A with the $S(Dt)$-sorted family $(s(A)^{-1}(s))_{s\in S(Dt)}$.

A *data state* ω for a data signature A is given by an attribute valuation $\omega\colon A \to \mathcal{D}$, i.e., $\omega(a) \in s(A)(a)^{\mathcal{D}}$ for $a \in |A|$; in particular, $\Omega(A) = \mathcal{D}^A$ is the set of A-data states. The *state predicates* $\mathscr{F}^{\mathcal{D}}_{A,X}$ are the formulæ in $\mathscr{F}^{\text{CASL}}_{Sig(Dt),A\cup X}$, taking A as well as an additional $S(Dt)$-indexed family X as variables. A state predicate $\phi \in \mathscr{F}^{\mathcal{D}}_{A,X}$ is to be interpreted over an A-data state ω and variable valuation $\beta\colon X \to \mathcal{D}$ and we define the *satisfaction relation* $\models^{\mathcal{D}}$ by

$$\omega, \beta \models^{\mathcal{D}}_{A,X} \phi \iff \mathcal{D}, \omega \cup \beta \models^{\text{CASL}}_{Sig(Dt),A\cup X} \phi\,.$$

The α-*reduct* of an A'-data state $\omega'\colon A' \to \mathcal{D}$ along a data signature morphism $\alpha\colon A \to A'$ is given by the A-data state $\omega'|\alpha\colon A \to \mathcal{D}$ with $(\omega'|\alpha)(a) = \omega'(\alpha(a))$ for every $a \in |A|$. The *state predicate translation* $\mathscr{F}^{\mathcal{D}}_{\alpha,X}\colon \mathscr{F}^{\mathcal{D}}_{A,X} \to \mathscr{F}^{\mathcal{D}}_{A',X}$ along $\alpha\colon A \to A'$ is given by the CASL-formula translation $\mathscr{F}^{\text{CASL}}_{Sig(Dt),\alpha\cup 1_X}$ along the substitution $\alpha \cup 1_X$. Reduct and translation fulfil the following *satisfaction condition* due to the general substitution lemma for CASL:

$$\omega'|\alpha, \beta \models^{\mathcal{D}}_{A,X} \phi \iff \omega', \beta \models^{\mathcal{D}}_{A',X} \mathscr{F}^{\mathcal{D}}_{\alpha,X}(\phi)\,.$$

A *data transition* (ω, ω') for a data signature A is a pair of A-data states; in particular, $\Omega^2(A) = (\mathcal{D}^A)^2$ is the set of A-data transitions. It holds that $(\mathcal{D}^A)^2 \cong \mathcal{D}^{2A}$, where $2A = A \uplus A$ and we assume that no attribute in A ends in a prime $\prime$ and all attributes in the second summand are adorned with an additional prime. The *transition predicates* $\mathscr{F}^{2\mathcal{D}}_{A,X}$ are the formulæ $\mathscr{F}^{\mathcal{D}}_{2A,X}$. The satisfaction relation $\models^{2\mathcal{D}}$ for a transition predicate $\psi \in \mathscr{F}^{2\mathcal{D}}_{A,X}$, data transition $(\omega, \omega') \in \Omega^2(A)$, and valuation $\beta\colon X \to \mathcal{D}$ is defined as

$$(\omega, \omega'), \beta \models^{2\mathcal{D}}_{A,X} \psi \iff \omega + \omega', \beta \models^{\mathcal{D}}_{2A,X} \psi$$

where $\omega + \omega' \in \Omega(2A)$ with $(\omega + \omega')(a) = \omega(a)$ and $(\omega + \omega')(a') = \omega'(a)$.

The α-*reduct* of an A'-data transition (ω', ω'') along a data signature morphism $\alpha\colon A \to A'$ is given by the A-data transition $(\omega', \omega'')|\alpha = (\omega'|\alpha, \omega''|\alpha)$. The *transition predicate translation* $\mathscr{F}^{2\mathcal{D}}_{\alpha,X}$ along α is given by $\mathscr{F}^{\mathcal{D}}_{2\alpha,X}$ with $2\alpha\colon 2A \to 2A'$ defined by $2\alpha(a) = \alpha(a)$ and $2\alpha(a') = \alpha(a)'$. Like for data states, reduct and translation fulfil the following *satisfaction condition*:

$$(\omega', \omega'')|\alpha, \beta \models^{2\mathcal{D}}_{A,X} \psi \iff (\omega', \omega''), \beta \models^{2\mathcal{D}}_{A',X} \mathscr{F}^{2\mathcal{D}}_{\alpha,X}(\psi)\,.$$

3.2 Events and Messages

An *event signature* E consists of a finite set of events $|E|$ and a map $\overline{s}(E)\colon |E| \to S(Dt)^*$ assigning to each $e \in |E|$ its list of parameter sorts. An *event signature morphism* $\eta\colon E \to E'$ is a function $\eta\colon |E| \to |E'|$ such that $\overline{s}(E)(e) = \overline{s}(E')(\eta(e))$ for all $e \in |E|$. We write $e(X)$ for $e \in |E|$ and $\overline{s}(E)(e) = s_1, \ldots, s_n$ when choosing n different *parameters* $X = x_1, \ldots, x_n$, and also $e(X) \in E$ in this case; when $f = e(X)$, we write $X(f)$ for X and we furthermore lift this notation to sets and lists of events. We sometimes identify the parameter list X with the $S(Dt)$-sorted family $(\{x_i \mid s_i = s\})_{s \in S(Dt)}$ and write $\overline{s}(E)(e)(x_i)$ for s_i.

A *message* $e(\beta)$ over an event signature E is given by an event $e(X) \in E$ with its parameters X instantiated by a parameter valuation $\beta\colon X \to \mathcal{D}$ such that $\beta(x) \in s^{\mathcal{D}}$ for $\overline{s}(E)(e)(x) = s$; the set of all messages over an event signature E is denoted by $\hat{E}(E)$. When $\hat{e} = e(\beta) \in \hat{E}(E)$, we write $\beta(\hat{e})$ for β, and when $e(X) \in E$ and $\beta\colon Y \to \mathcal{D}$ for $X \subseteq Y$, we write $e(\beta)$ for $e(\beta{\restriction}X)$; both notations are furthermore lifted to sets and lists.

The set of *shufflings* $\hat{F}_1 \parallel \hat{F}_2$ of two message lists $\hat{F}_1$ and $\hat{F}_2$ is inductively given by

$$\hat{F} \parallel \varepsilon = \{\hat{F}\} = \varepsilon \parallel \hat{F}\,,$$
$$(\hat{f} :: \hat{F}_1) \parallel \hat{F}_2 = \{\hat{f} :: \hat{F} \mid \hat{F} \in \hat{F}_1 \parallel \hat{F}_2\} = \hat{F}_1 \parallel (\hat{f} :: \hat{F}_2)\,.$$

An event signature morphism $\eta\colon E \to E'$ is lifted to a message $e(\beta) \in \hat{E}(E)$ by setting $\hat{E}(\eta)(e(\beta)) = \eta(e)(\beta) \in \hat{E}(E')$ and also to sets and lists of messages.

3.3 Event/Data Signatures

An *event/data signature* Σ consists of *input* and *output* event signatures $I(\Sigma)$ and $O(\Sigma)$, and a data signature $A(\Sigma)$. An *event/data signature morphism* $\sigma\colon \Sigma \to \Sigma'$ consists of an input event signature morphism $I(\sigma)\colon I(\Sigma) \to I(\Sigma')$, an output event signature morphism $O(\sigma)\colon O(\Sigma) \to O(\Sigma')$, and a data signature morphism $A(\sigma)\colon A(\Sigma) \to A(\Sigma')$. We lift the event signatures and signature morphisms to messages by writing $\hat{I}(\Sigma)$ for $\hat{E}(I(\Sigma))$, $\hat{O}(\Sigma)$ for $\hat{E}(O(\Sigma))$, $\hat{I}(\sigma)$ for $\hat{E}(I(\sigma))$, and $\hat{O}(\sigma)$ for $\hat{E}(O(\sigma))$.

The category of $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-*signatures* $\mathbb{S}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}$ consists of the event/data signatures and signature morphisms.

3.4 Event/Data Structures

A *configuration* $\gamma = (c, d)$ consists of a *control state* c from some set of control states C and a *data state* d from some set of data states D. Given a data signature A the data state

of γ may be *labelled* by a map ω such that $\omega(d) \in \Omega(A)$. For a set of configurations Γ we write $C(\Gamma)$ for its set of control states.

A Σ*-event/data structure* $M = (\Gamma, R, \Gamma_0, \omega)$ over an event/data signature Σ consists of a set of *configurations* $\Gamma \subseteq C \times D$, a family of *transition relations* $R = (R_{\hat{\imath},\hat{O}} \subseteq \Gamma \times \Gamma)_{\hat{\imath}\in\hat{I}(\Sigma),\hat{O}\in\hat{O}(\Sigma)^*}$, and a non-empty set of *initial configurations* $\Gamma_0 \subseteq \Gamma$ such that Γ is *reachable* from Γ_0 via R, i.e., for all $\gamma \in \Gamma$ there are $\gamma_0 \in \Gamma_0$, $n \geq 0$, $\hat{\imath}_1, \ldots, \hat{\imath}_n \in \hat{I}(\Sigma)$, $\hat{O}_1, \ldots, \hat{O}_n \in \hat{O}(\Sigma)^*$, and $(\gamma_k, \gamma_{k+1}) \in R_{\hat{\imath}_{k+1},\hat{O}_{k+1}}$ for all $0 \leq k < n$ with $\gamma_n = \gamma$; and a *data state labelling* $\omega\colon D \to \Omega(A(\Sigma))$.

We write $c(M)(\gamma) = c$ and $\omega(M)(\gamma) = \omega(d)$ for $\gamma = (c, d) \in \Gamma$, $\Gamma(M)$ for Γ, $C(M)$ for $\{c(M)(\gamma) \mid \gamma \in \Gamma(M)\}$, $R(M)$ for R, $\Gamma_0(M)$ for Γ_0, $C_0(M)$ for $C(\Gamma_0)$, and $\Omega_0(M)$ for $\{\omega(M)(\gamma_0) \mid \gamma_0 \in \Gamma_0\}$.

The above definition restricts structures to reachable ones only. Although an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-sentence will hold in an event/data structure if it is satisfied in all its initial states, the modal and hybrid operators of $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ will allow for expressing that a certain property holds in all (reachable) states of the structure.

The σ*-reduct* of a Σ'-event/data structure M' along the event/data signature morphism $\sigma\colon \Sigma \to \Sigma'$ is the Σ-event/data structure $M'|\sigma$ such that

- $\Gamma(M'|\sigma) \subseteq \Gamma(M')$ as well as $R(M'|\sigma) = (R(M'|\sigma)_{\hat{\imath},\hat{O}})_{\hat{\imath}\in\hat{I}(\Sigma),\hat{O}\in\hat{O}(\Sigma)^*}$ are inductively defined by $\Gamma_0(M') \subseteq \Gamma(M'|\sigma)$ and, for all $\gamma', \gamma'' \in \Gamma(M')$, $\hat{\imath} \in \hat{I}(\Sigma)$, and $\hat{O} \in \hat{O}(\Sigma)^*$, if $\gamma' \in \Gamma(M'|\sigma)$ and $(\gamma', \gamma'') \in R(M')_{\hat{I}(\sigma)(\hat{\imath}),\hat{O}(\sigma)(\hat{O})}$, then $\gamma'' \in \Gamma(M'|\sigma)$ and $(\gamma', \gamma'') \in R(M'|\sigma)_{\hat{\imath},\hat{O}}$;
- $\Gamma_0(M'|\sigma) = \Gamma_0(M')$; and
- $\omega(M'|\sigma)(\gamma') = (\omega(M')(\gamma'))|A(\sigma)$ for all $\gamma' \in \Gamma(M'|\sigma)$.

This σ-reduct keeps exactly those transitions that are a direct image along σ. It would also be possible to additionally keep transitions that show a super-list of the outputs that can be reached by σ. When moving to $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-sentences, however, it turns out to be impossible to fix a particular list of outputs.

Given sets of input events $J \subseteq I(\Sigma)$ and output events $N \subseteq O(\Sigma)$, we denote by $\Gamma^{J,N}(M, \gamma)$ and $\Gamma^{J,N}(M)$, respectively, the set of configurations of a Σ-event/data structure M that are J, N-reachable from a configuration $\gamma \in \Gamma(M)$ and from an initial configuration $\gamma_0 \in \Gamma_0(M)$, respectively. Here a $\gamma_n \in \Gamma(M)$ is J, N*-reachable in* M *from* a $\gamma_1 \in \Gamma(M)$ if there are $n \geq 1$, $\hat{\imath}_2, \ldots, \hat{\imath}_n \in \hat{I}(J)$, $\hat{O}_2, \ldots, \hat{O}_n \in \hat{O}(N)^*$, and $(\gamma_i, \gamma_{i+1}) \in R(M)_{\hat{\imath}_{k+1},\hat{O}_{k+1}}$ for all $1 \leq k < n$.

The Σ-event/data structures form the discrete category $Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma)$ of $\mathcal{M}^{\downarrow}_{\mathcal{D}}$*-structures* over Σ. For each $\sigma\colon \Sigma \to \Sigma'$ in $\mathbb{S}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}$ the σ*-reduct functor* $Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\sigma)\colon Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma') \to Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma)$ is given by $Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\sigma)(M') = M'|\sigma$.

3.5 Event/Data Formulæ and Sentences

The Σ*-event/data formulæ* $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S}$ over an event/data signature Σ and a set of *state variables* S are inductively defined by

- φ — data state sentence $\varphi \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma),\emptyset}$ holds in the current configuration;

- s — the control state of the current configuration is $s \in S$;
- $\downarrow s \,.\, \varrho$ — calling the current control state s, formula $\varrho \in \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma, S \uplus \{s\}}$ holds (s is turned into a fresh variable by adding to by disjoint union to the set of state variables);
- $(@^{J,N} s)\varrho$ — in all configurations with control state $s \in S$ that are J, N-reachable, formula $\varrho \in \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma, S}$ holds (relativised "jump");
- $\Box^{J,N} \varrho$ — in all configurations that are J, N-reachable from the current configuration formula $\varrho \in \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma, S}$ holds (relativised "globally");
- $\langle i \mathbin{/\!\!/} [O]_N : \psi \rangle \varrho$ — in the current configuration there are valuations $\beta \colon X(i) \to \mathcal{D}$, $\beta' \colon X(O) \to \mathcal{D}$, and a transition for the incoming message $i(\beta) \in \hat{I}(\Sigma)$ and the outgoing messages $\hat{O}' \in O(\beta') \parallel \hat{N}$ for $O(\beta') \in \hat{O}(\Sigma)^*$, $\hat{N} \in \hat{O}(N)^*$ such that $\beta \cup \beta'$ satisfies transition formula $\psi \in \mathscr{F}^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O)}$ and $\varrho \in \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma, S}$ holds afterwards;
- $\langle\!\langle i : \phi \mathbin{/\!\!/} [O]_N : \psi \rangle\!\rangle \varrho$ — in the current configuration for all valuations $\beta \colon X(i) \to \mathcal{D}$ satisfying state formula $\phi \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma), X(i)}$ there are a valuation $\beta' \colon X(O) \to \mathcal{D}$ and a transition for the incoming message $i(\beta) \in \hat{I}(\Sigma)$ and the outgoing messages $\hat{O}' \in O(\beta') \parallel \hat{N}$ for $O(\beta') \in \hat{O}(\Sigma)^*$, $\hat{N} \in \hat{O}(N)^*$ such that $\beta \cup \beta'$ satisfies transition formula $\psi \in \mathscr{F}^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O)}$ and $\varrho \in \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma, S}$ holds afterwards;
- $\neg \varrho$ — in the current configuration $\varrho \in \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma, S}$ does not hold;
- $\varrho_1 \vee \varrho_2$ — in the current configuration $\varrho_1 \in \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma, S}$ or $\varrho_2 \in \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma, S}$ hold.

We write $(@s)\varrho$ for $(@^{I(\Sigma), O(\Sigma)} s)\varrho$, $\Box \varrho$ for $\Box^{I(\Sigma), O(\Sigma)} \varrho$, $[i \mathbin{/\!\!/} [O]_N : \psi]\varrho$ for $\neg\langle i \mathbin{/\!\!/} [O]_N : \psi \rangle \neg \varrho$, and true for $\downarrow s \,.\, s$; we write O for $[O]_\emptyset$.

Two different kinds of relativisations are used in $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-formulæ: For the jump operator $(@^{J,N} s)\varrho$ and the globally operator $\Box^{J,N} \varrho$ the subsets of input events $J \subseteq I(\Sigma)$ and output events $N \subseteq O(\Sigma)$ restrict the referable states in an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-structure to those that are J, N-reachable. On the other hand, $[O]_N$ specifies that besides messages from O additional messages for events in $N \subseteq O(\Sigma)$ can be mixed into the output, such that, in particular, $[O]_\emptyset$ requires exactly O. Since the set of output events is assumed to be finite, $[O]_N$ can be used to specify message lists of arbitrary length with finitely many formulæ. Moreover, the syntactic information in both kinds of relativisations is kept through a translation to another $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signature.

Let $\sigma \colon \Sigma \to \Sigma'$ be an event/data signature morphism. The *event/data formulæ translation* $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S} \colon \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma, S} \to \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma', S}$ along σ is recursively given by

- $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S}(\varphi) = \mathscr{F}^{\mathcal{D}}_{A(\sigma), \emptyset}(\varphi)$;
- $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S}(s) = s$;
- $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S}(\downarrow s \,.\, \varrho) = \downarrow s \,.\, \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S \uplus \{s\}}(\varrho)$;
- $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S}((@^{J,N} s)\varrho) = (@^{I(\sigma)(J), O(\sigma)(N)} s)\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S}(\varrho)$;
- $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S}(\Box^{J,N} \varrho) = \Box^{I(\sigma)(J), O(\sigma)(N)} \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S}(\varrho)$;
- $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S}(\langle i \mathbin{/\!\!/} [O]_N : \psi \rangle \varrho) =$
  $\langle I(\sigma)(i) \mathbin{/\!\!/} [O(\sigma)(O)]_{O(\sigma)(N)} : \mathscr{F}^{2\mathcal{D}}_{A(\sigma), X(i) \cup X(O)}(\psi) \rangle \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma, S}(\varrho)$;

- $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma,S}(\langle\!|i : \phi /\!\!/ [O]_N : \psi|\!\rangle \varrho) =$
  $\langle\!|I(\sigma)(i) : \mathscr{F}^{\mathcal{D}}_{A(\sigma),X(i)}(\phi) /\!\!/ [O(\sigma)(O)]_{O(\sigma)(N)} : \mathscr{F}^{2\mathcal{D}}_{A(\sigma),X(i)\cup X(O)}(\psi)|\!\rangle \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma,S}(\varrho)$;
- $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma,S}(\neg\varrho) = \neg\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma,S}(\varrho)$;
- $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma,S}(\varrho_1 \vee \varrho_2) = \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma,S}(\varrho_1) \vee \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma,S}(\varrho_2)$.

The set $\mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma)$ of *Σ-event/data sentences* is given by $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,\emptyset}$, the *event/data sentence translation* $\mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\sigma)\colon \mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma) \to \mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma')$ by $\mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\sigma,\emptyset}$.

3.6 Satisfaction Relation for $\mathcal{M}^{\downarrow}_{\mathcal{D}}$

Let Σ be an event/data signature, M a Σ-event/data structure, S a set of state variables, $v\colon S \to C(M)$ a state variable assignment, and $\gamma \in \Gamma(M)$. The *satisfaction relation* for event/data formulæ is inductively given by

- $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \varphi$ iff $\omega(M)(\gamma), \emptyset \models^{\mathcal{D}}_{A(\Sigma),\emptyset} \varphi$;
- $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} s$ iff $v(s) = c(M)(\gamma)$;
- $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \downarrow s\,.\,\varrho$ iff $M, v\{s \mapsto c(M)(\gamma)\}, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S\uplus\{s\}} \varrho$;
- $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} (@^{J,N}s)\varrho$ iff $M, v, \gamma' \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \varrho$ for all $\gamma' \in \Gamma^{J,N}(M)$ with $c(M)(\gamma') = v(s)$;
- $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \Box^{J,N}\varrho$ iff $M, v, \gamma' \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \varrho$ for all $\gamma' \in \Gamma^{J,N}(M, \gamma)$;
- $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \langle i /\!\!/ [O]_N : \psi\rangle\varrho$ iff there are valuations $\beta\colon X(i) \to \mathcal{D}, \beta'\colon X(O) \to \mathcal{D}$, output messages $\hat{O}' \in O(\beta') \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$, and a configuration $\gamma' \in \Gamma(M)$ such that $(\gamma, \gamma') \in R(M)_{i(\beta),\hat{O}'}$, $(\omega(M)(\gamma), \omega(M)(\gamma')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma),X(i)\cup X(O)} \psi$, and $M, v, \gamma' \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \varrho$;
- $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \langle\!|i : \phi /\!\!/ [O]_N : \psi|\!\rangle\varrho$ iff for all valuations $\beta\colon X(i) \to \mathcal{D}$ that satisfy $\omega(M)(\gamma), \beta \models^{\mathcal{D}}_{A(\Sigma),X(i)} \phi$ there are a valuation $\beta'\colon X(O) \to \mathcal{D}$, output messages $\hat{O}' \in O(\beta') \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$, and a configuration $\gamma' \in \Gamma(M)$ such that $(\gamma, \gamma') \in R(M)_{i(\beta),\hat{O}'}$, $(\omega(M)(\gamma), \omega(M)(\gamma')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma),X(i)\cup X(O)} \psi$, and $M, v, \gamma' \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \varrho$;
- $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \neg\varrho$ iff $M, v, \gamma \not\models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \varrho$;
- $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \varrho_1 \vee \varrho_2$ iff $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \varrho_1$ or $M, v, \gamma \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \varrho_2$.

For a $\Sigma \in |\mathbb{S}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}|$, an $M \in |Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma)|$, and a $\rho \in \mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma)$ the *satisfaction relation* $M \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma} \rho$ holds if, and only if, $M, \emptyset, \gamma_0 \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,\emptyset} \rho$ for all $\gamma_0 \in \Gamma_0(M)$.

Theorem 1. $(\mathbb{S}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}, Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}, \mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}, \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}})$ *is an institution.*

from *Basic/StructuredDatatypes* **get** LIST, SET % import finite lists and sets
spec $\text{TRANS}_\Sigma = Dt$
then free type $\text{InEvt} ::= I(\Sigma)$
free type $\text{OutEvt} ::= O(\Sigma)$
then LIST[sort OutEvt] **and** SET[sort InEvt] **and** SET[sort OutEvt]
then sort Ctrl
free type $\text{Conf} ::= \text{conf}(\text{c} : \text{Ctrl}; A(\Sigma))$
preds $\text{init} : \text{Conf};$
$\text{trans} : \text{Conf} \times \text{InEvt} \times \mathit{List}[\text{OutEvt}] \times \text{Conf}$
$\cdot \exists g : \text{Conf} \cdot \text{init}(g)$ % there is some initial configuration
then free { **pred** $\text{reachable} : \mathit{Set}[\text{InEvt}] \times \mathit{Set}[\text{OutEvt}] \times \text{Conf} \times \text{Conf}$
$\forall g, g', g'' : \text{Conf}; J : \mathit{Set}[\text{InEvt}]; N : \mathit{Set}[\text{OutEvt}]; i : \text{InEvt}; O : \mathit{List}[\text{OutEvt}]$
$\cdot \text{reachable}(J, N, g, g)$
$\cdot \text{reachable}(J, N, g, g') \wedge i \in J \wedge O \subseteq N \wedge \text{trans}(g', i, O, g'') \Rightarrow$
$\text{reachable}(J, N, g, g'')$ }
then preds $\text{reachable}(J : \mathit{Set}[\text{InEvt}], N : \mathit{Set}[\text{OutEvt}], g : \text{Conf}) \Leftrightarrow$
$\exists g_0 : \text{Conf} \cdot \text{init}(g_0) \wedge \text{reachable}(J, N, g_0, g);$
$\text{reachable}(g : \text{Conf}) \Leftrightarrow \text{reachable}(I(\Sigma), O(\Sigma), g)$
then pred $\text{mixed} : \mathit{List}[\text{OutEvt}] \times \mathit{Set}[\text{OutEvt}] \times \mathit{List}[\text{OutEvt}]$
$\forall o, o' : \text{OutEvt}; O, O' : \mathit{List}[\text{OutEvt}]; N : \mathit{Set}[\text{OutEvt}]$
$\cdot \text{mixed}(O, N, O)$
$\cdot \text{mixed}(o :: O, N, o :: O')$ **if** $\text{mixed}(O, N, O')$
$\cdot \text{mixed}(O, N, o' :: O')$ **if** $\text{mixed}(O, N, O') \wedge o' \in N$
end

Figure 2. Frame for translating $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ into CASL.

3.7 A Theoroidal Comorphism from $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ to CASL

We define a theoroidal comorphism from $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ to CASL. The construction mainly follows the standard translation of modal logics to first-order logic [1] and extends the scheme of [16] by outputs.

The basis is a representation of $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signatures and the frame given by $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-structures as a CASL-specification as shown in Fig. 2. The signature translation

$$\nu^{Sig} : \mathbb{S}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}} \to \text{Pres}^{\text{CASL}}$$

maps an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signature Σ to the CASL-theory presentation given by TRANS_Σ and an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signature morphism to the corresponding theory presentation morphism. TRANS_Σ first of all covers the events according to $I(\Sigma)$ and $O(\Sigma)$ with types InEvt and OutEvt, and the configurations with type Conf showing a single constructor conf for the control state from Ctrl and a data state given by assignments to the attributes from $A(\Sigma)$. Furthermore, TRANS_Σ sets the frame for describing reachable transition systems with a set of initial configurations, a transition relation, and reachability predicates, where the specification of reachable uses CASL's "structured free" construct to ensure reachability to be inductively defined. Finally, a predicate mixed is included for representing the shufflings of a list of outputs with some additional output events.

The model translation

$$\nu^{\mathrm{Mod}}_{\Sigma} \colon \mathrm{Mod}^{\textsc{Casl}}(\nu^{Sig}(\Sigma)) \to Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma)$$

then can rely on this encoding. In particular, for a model $M' \in \mathrm{Mod}^{\textsc{Casl}}(\nu^{Sig}(\Sigma))$, there are bijective maps $\iota_{M',\mathrm{Conf}} \colon \mathrm{Conf}^{M'} \cong \mathrm{Ctrl}^{M'} \times \Omega(A(\Sigma))$ for the configurations as well as $\iota_{M',\mathrm{InEvt}} \colon \mathrm{InEvt}^{M'} \cong \hat{I}(\Sigma)$ and $\iota_{M',\mathrm{OutEvt}} \colon \mathrm{OutEvt}^{M'} \cong \hat{O}(\Sigma)$ for the messages. Moreover, $\mathrm{mixed}^{M'}(\iota^{-1}_{M',\mathrm{OutEvt}}(\hat{O}), \iota^{-1}_{M',\mathrm{OutEvt}}(N), \iota^{-1}_{M',\mathrm{OutEvt}}(\hat{O}'))$ if, and only if, $\hat{O}' \in \hat{O} \parallel \hat{N}'$ with $\hat{N}' \in N^*$. The $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-structure resulting from a CASL-model M' of $\textsc{Trans}_{\Sigma}$ can thus be defined by

- $\Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) = \iota^{-1}_{M',\mathrm{Conf}}(\{g^{M'} \in \mathrm{Conf}^{M'} \mid \mathrm{reachable}^{M'}(g^{M'})\})$
- $R(\nu^{\mathrm{Mod}}_{\Sigma}(M'))_{\hat{\imath},\hat{O}} = \{(\gamma,\gamma') \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \times \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \mid \mathrm{trans}^{M'}(\iota_{M',\mathrm{Conf}}(\gamma), \iota^{-1}_{M',\mathrm{InEvt}}(\hat{\imath}), \iota^{-1}_{M',\mathrm{OutEvt}}(\hat{O}), \iota_{M',\mathrm{Conf}}(\gamma'))\}$
- $\Gamma_0(\nu^{\mathrm{Mod}}_{\Sigma}(M')) = \{\gamma \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \mid \mathrm{init}^{M'}(\iota_{M',\mathrm{Conf}}(\gamma))\}$
- $\omega(\nu^{\mathrm{Mod}}_{\Sigma}(M')) = \{(c,\omega) \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \mapsto \omega\}$

For $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-sentences, we first define a formula translation

$$\nu^{\mathscr{F}}_{\Sigma,S,g} \colon \mathscr{F}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma,S} \to \mathscr{F}^{\textsc{Casl}}_{\nu^{\mathbb{S}}(\Sigma),S\cup\{g\}}$$

which, mimicking the standard translation, takes a variable $g : \mathrm{Conf}$ as a parameter that records the "current configuration" and also uses a set S of state names for the control states. The translation embeds the data state and 2-data state formulæ using the substitution $A(\Sigma)(g) = \{a \mapsto a(g) \mid a \in A(\Sigma)\}$ for replacing the attributes $a \in A(\Sigma)$ by the accessors $a(g)$. The translation of $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-formulæ then reads

- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\varphi) = \mathscr{F}^{\textsc{Casl}}_{\nu^{\mathbb{S}}(\Sigma),A(\Sigma)(g)}(\varphi)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(s) = (s = \mathrm{c}(g))$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\downarrow s \,.\, \varrho) = \exists s : \mathrm{Ctrl} \,.\, s = \mathrm{c}(g) \wedge \nu^{\mathscr{F}}_{\Sigma,S\uplus\{s\},g}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}((@^{J,N}s)\varrho) = \forall g' : \mathrm{Conf} \,.\, (\mathrm{c}(g') = s \wedge \mathrm{reachable}(J,N,g')) \Rightarrow \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\Box^{J,N}\varrho) = \forall g' : \mathrm{Conf} \,.\, \mathrm{reachable}(J,N,g,g') \Rightarrow \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\langle i \mathbin{/\!\!/} [O]_N : \psi\rangle\varrho) = \exists X : \overline{s}(I(\Sigma))(i); X' : \overline{s}(O(\Sigma))(O); O' : \mathit{List}[\mathrm{OutEvt}]; g' : \mathrm{Conf} \,.\, \mathrm{mixed}(O(X'),N,O') \wedge \mathrm{trans}(g,i(X),O',g') \wedge \mathscr{F}^{\textsc{Casl}}_{\nu^{\mathbb{S}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_{X\cup X'}}(\psi) \wedge \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\langle\!|i : \phi \mathbin{/\!\!/} [O]_N : \psi|\!\rangle\varrho) = \forall X : \overline{s}(I(\Sigma))(i) \,.\, \mathscr{F}^{\textsc{Casl}}_{\nu^{\mathbb{S}}(\Sigma),A(\Sigma)(g)\cup 1_X}(\phi) \Rightarrow \exists X' : \overline{s}(O(\Sigma))(O); O' : \mathit{List}[\mathrm{OutEvt}]; g' : \mathrm{Conf} \,.\, \mathrm{mixed}(O(X'),N,O') \wedge \mathrm{trans}(g,i(X),O',g') \wedge \mathscr{F}^{\textsc{Casl}}_{\nu^{\mathbb{S}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_{X\cup X'}}(\psi) \wedge \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\neg\varrho) = \neg\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_1 \vee \varrho_2) = \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_1) \vee \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_2)$

Building on the translation of formulæ, the sentence translation

$$\nu^{\mathrm{Sen}}_{\Sigma} \colon \mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma) \to \mathrm{Sen}^{\textsc{Casl}}(\nu^{\mathbb{S}}(\Sigma))$$

only has to require additionally that evaluation starts in an initial state:

– $\nu^{\mathrm{Sen}}_{\Sigma}(\rho) = \forall g : \mathrm{Conf} \,.\, \mathrm{init}(g) \Rightarrow \nu^{\mathscr{F}}_{\Sigma,\emptyset,g}(\rho)$

Theorem 2. *$(\nu^{Sig}, \nu^{\mathrm{Mod}}, \nu^{\mathrm{Sen}})$ is a theoroidal comorphism from $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ to* CASL.

For a CASL-proof of an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$*-invariant* $\Box\varphi$ such that φ has to hold in every reachable configuration, the full generality of the reachable predicate can sometimes be avoided by replacing the proof obligation $\forall g : \mathrm{Conf} \,.\, \mathrm{reachable}(g) \Rightarrow \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathbb{S}}(\Sigma),A(\Sigma)(g)}(\varphi)$ by the usual stepwise induction scheme that only requires to demonstrate the invariant to hold in all initial configurations and that it is preserved by every transition. Moreover, the $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-state formula φ can be generalised into a CASL-invariant.

Proposition 1. *Let (Σ, P) be a theory presentation in $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ and $(\nu^{\mathbb{S}}(\Sigma), \Phi)$ a theory presentation in* CASL *such that $\mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathrm{Pres}}(\Sigma, P)) \subseteq \mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathbb{S}}(\Sigma), \Phi)$. Let $inv^{\mathrm{CASL}}(g) \in \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathbb{S}}(\Sigma),\{g\}}$ be a* CASL*-formula with a single free variable g and $inv^{\mathcal{M}^{\downarrow}_{\mathcal{D}}} \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma),\emptyset}$ an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-state formula, such that*

(I0) $\forall g : \mathrm{Conf} \,.\, inv^{\mathrm{CASL}}(g) \Rightarrow \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathbb{S}}(\Sigma),A(\Sigma)(g)}(inv^{\mathcal{M}^{\downarrow}_{\mathcal{D}}})$

(I1) $\forall g : \mathrm{Conf} \,.\, \mathrm{init}(g) \Rightarrow inv^{\mathrm{CASL}}(g)$

(I2) $\forall g, g' : \mathrm{Conf}; i \in \mathrm{InEvt}; O \in \mathit{List}[\mathrm{OutEvt}] \,.$
$inv^{\mathrm{CASL}}(g) \wedge \mathrm{trans}(g, i, O, g') \Rightarrow inv^{\mathrm{CASL}}(g')$

hold in every model $M' \in \mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathbb{S}}(\Sigma), \Phi)$. Then $\nu^{\mathrm{Mod}}_{\Sigma}(M') \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma} \Box inv^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}$ for all models $M' \in \mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathrm{Pres}}(\Sigma, P))$.

4 Simple UML State Machines with Outputs

UML state machines [13, Ch. 14] provide means to specify the reactive behaviour of objects or component instances. These entities hold an internal data state, typically given by a set of attributes or properties as specified in a static structure, and shall react to event occurrences like incoming messages by firing different transitions in different control states. Such transitions may have a guard depending on event arguments and the internal state and may change, as an effect, the internal control and data state of the entity as well as send out messages on their own. Beyond such "simple" means for specifying reactive entities, UML state machines offer also more advanced modelling constructs, like hierarchical states or compound transitions, which, however, we defer to future work.

In our formal account, extending again [16], a *simple UML state machine with outputs* U uses an event/data signature $\Sigma(U)$ for its input and output events as well as its attributes and consists of a finite set of *control states* $C(U)$; a finite set of *transition specifications* $T(U)$ of the form $(c, \phi, i(X), o_1(X_1), \ldots, o_m(X_m), \psi, c')$ with

– *source* and *target* control states $c, c' \in C(U)$,
– *input* event $i(X) \in I(\Sigma(U))$ and *output* events $o_1(X_1), \ldots, o_m(X_m) \in O(\Sigma(U))$ such that $X \cap \bigcup_{1 \leq k \leq m} X_m = \emptyset$,
– *precondition* state predicate $\phi \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma(U)),X}$, and

– *postcondition* transition predicate $\psi \in \mathscr{F}^{2\mathcal{D}}_{A(\Sigma(U)),X\cup\bigcup_{1\leq k\leq m} X_k}$;

an *initial control state* $c_0(U) \in C(U)$; and an *initial state predicate* $\varphi_0(U) \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma(U)),\emptyset}$, such that $C(U)$ is *syntactically reachable*, i.e., for every $c \in C(U) \setminus \{c_0(U)\}$ there are $(c_0(U), \phi_1, i_1, O_1, \psi_1, c_1), \ldots, (c_{n-1}, i_n, O_n, \psi_n, c_n) \in T(U)$ with $n > 0$ and $c_n = c$. The constraint of syntactic reachability is only introduced to simplify semantic and algorithmic constructions on simple UML state machines with output.

A $\Sigma(U)$-event/data structure M is a *model* of a simple UML state machine U with output, $M \in \mathrm{Mod}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(U)$, if $C(U) \subseteq C(M)$ up to a bijective renaming, $C_0(M) = \{c_0(U)\}$, $\Omega_0(M) \subseteq \{\omega \in |\Omega(A(\Sigma(U)))| \mid \omega \models^{\mathcal{D}}_{A(\Sigma(U)),\emptyset} \varphi_0(U)\}$, and if the following holds for all $(c, d) \in \Gamma(M)$:

– for all transition specifications $(c, \phi, i, O, \psi, c') \in T(U)$ and $\beta\colon X(i) \to \mathcal{D}$ with $\omega(M)(d), \beta \models^{\mathcal{D}}_{A(\Sigma(U)),X(i)} \phi$, there is a $\beta'\colon X(O) \to \mathcal{D}$ and a pair $((c, d), (c', d')) \in R(M)_{i(\beta),O(\beta')}$ such that $(\omega(M)(d), \omega(M)(d')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma(U)),X(i)\cup X(O)} \psi$;
– for all pairs $((c, d), (c', d')) \in R(M)_{i(\beta),O(\beta')}$ there is some transition specification $(c, \phi, i, O, \psi, c') \in T(U)$ such that $\omega(M)(d), \beta \models^{\mathcal{D}}_{A(\Sigma(U)),X(i)} \phi$ and also $(\omega(M)(d), \omega(M)(d')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma(U)),X(\{i\}\cup O)} \psi$.

The last requirement that all transitions in a model are due to transition specifications does not cover the requirement of *input enabledness* for UML state machines: An event for which currently no transition can fire is discarded. This behaviour can be added by a syntactical transformation extending the set of transition specifications by self-loops with empty outputs for all situations where some event is not accepted.

In UML, completion events are produced whenever a state completes its internal behaviour and such events have always to be prioritised in event processing; the reaction to a completion event is indicated by a transition without a triggering event. For the simple machines with output described here, where states do not show internal behaviour, the only use of completion events is to let a machine make progress autonomously without external input. For using this feature, the machine's event/data signature has to be extended by such events and the transition specifications have to take completions into account. Still, the prioritisation cannot be covered by a single state machine alone, as it has no event processing discipline of its own.

Extending the characterisation algorithm in [16] with outputs, it can be shown that $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ is expressive enough to capture the model class of a simple UML state machine with output U by a single sentence ϱ_U such that $M \in \mathrm{Mod}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(U)$ if, and only if, $M \models^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}_{\Sigma(U)} \varrho_U$. The simplest case is a single transition specification $(c, \phi, i, O, \psi, c')$: By requiring $(@c)\langle\!|i : \phi /\!/ O : \psi|\!\rangle c'$ it can be ensured that a model indeed shows a transition from control state c to the control state c' for the input event i with precondition ϕ satisfied which outputs O with ψ satisfied. For requiring that such a transition for input i and output O is only offered when the precondition ϕ and the transition condition ψ hold, a formula $(@c)[i /\!/ O : \neg\phi \vee \neg\psi]\mathrm{false}$ has to be added. For ensuring that no other output than O can be produced, on the one hand $(@c)[i /\!/ O' : \mathrm{true}]\mathrm{false}$ for every $O' \neq O$ that is at most the length of O has to be added and on the other hand $(@c)[i /\!/ [O']_{O(\Sigma)} : \mathrm{true}]\mathrm{false}$ for every O' with length one more than O.

Reasoning over a simple UML state machine with output U in CASL via the translation of U's characterising sentence along the theoroidal comorphism of Thm. 2 will involve some not fully transpicuous axioms due to the necessary exclusion of some behaviour using formulæ like $(@c)[i /\!\!/ [O']_{O(\Sigma)} : \mathrm{true}]\mathrm{false}$. It is therefore sometimes advantageous to directly use the requirements for M being a model of U to obtain another characterisation of the trans predicate in the CASL presentation for the comorphism, which then can be favourably combined with Prop. 1 for proving invariants:

Proposition 2. *Let U be a simple UML state machine with output and let $M' \in \mathrm{Mod}^{\mathrm{CASL}}(\nu^{Sig}(\Sigma(U)))$ such that $\nu^{\mathrm{Mod}}_{\Sigma(U)}(M') \in \mathrm{Mod}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(U)$. Then*

$$
\begin{aligned}
M' \models^{\mathrm{CASL}}_{\nu^{\mathbb{S}}(\Sigma)} \forall g : \mathrm{Conf} \,.\, \mathrm{reachable}(g) \Rightarrow \\
\Big(\forall g' : \mathrm{Conf}; i_* : \mathrm{InEvt}; O_* : \mathit{List}[\mathrm{OutEvt}] \,.\, \mathrm{trans}(g, i_*, O_*, g') \iff \\
\bigvee_{(c,\phi,i,O,\psi,c') \in T(U)} \exists X : \overline{s}(I(\Sigma))(i); X' : \overline{s}(O(\Sigma))(O) \,. \\
\mathrm{c}(g) = c \wedge \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathbb{S}}(\Sigma), A(\Sigma)(g) \cup 1_X}(\phi) \wedge i_* = i(X) \wedge O_* = O(X') \wedge \\
\mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathbb{S}}(\Sigma), A(\Sigma)(g) \cup A(\Sigma)(g') \cup 1_{X \cup X'}}(\psi) \wedge \mathrm{c}(g') = c' \Big) \,.
\end{aligned}
$$

5 Simple UML Composite Structures

A UML composite structure [13, Ch. 11] specifies the internal structure of a class or component and its collaborations. For our purposes, a composite structure is given by class or component instances, its so-called *parts*, that can communicate through their attached *ports* specifying provided and required interfaces and being linked by *connectors*. All connectors are assumed to be binary and each part to be equipped with a state machine for describing its behaviour.

A *composite structure signature* Δ over $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ consists of a set $\mathit{Cmp}(\Delta)$ of *parts* c each equipped with an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signature $\Sigma(\Delta, c)$ for its input and output events and internal attributes; a set $\mathit{Prt}(\Delta)$ of *ports* p each showing a part $\mathit{cmp}(\Delta)(p) \in \mathit{Cmp}(\Delta)$ as well as an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signature $\Sigma(\Delta, p)$ without attributes (i.e., $A(\Sigma(\Delta, p)) = \emptyset$) for its *provided* (input) and *required* (output) events; and a symmetric binary relation $\mathit{Con}(\Delta) \subseteq \mathit{Prt}(\Delta) \times \mathit{Prt}(\Delta)$ of *connectors* such that

- for each part $c \in \mathit{Cmp}(\Delta)$, the input and output events of $\Sigma(\Delta, c)$ are the provided and required events of c's ports prefixed with the port name, i.e., for $F \in \{I, O\}$, $F(\Sigma(\Delta, c)) = \bigcup_{p \in \mathit{cmp}(\Delta)^{-1}(c)} \{p.f \mid f \in F(\Sigma(\Delta, p))\}$;
- for each part $c \in \mathit{Cmp}(\Delta)$, the attributes of $\Sigma(\Delta, c)$ are all prefixed with c, i.e., if $a \in A(\Sigma(\Delta, c))$, then $a = c.a_*$;
- for each connection $(p, p') \in \mathit{Con}(\Delta)$, the required events of port p are provided by p', i.e., $O(\Sigma(\Delta, p)) \subseteq I(\Sigma(\Delta, p'))$.

We say that port $p \in \mathit{Prt}(\Delta)$ is *open* in Δ if there is no $p' \in \mathit{Prt}(\Delta)$ such that $(p, p') \in \mathit{Con}(\Delta)$; otherwise p is *connected*.

A *composite structure signature morphism* $\delta\colon \Delta \to \Delta'$ over $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ consists of a function $Cmp(\delta)\colon Cmp(\Delta) \to Cmp(\Delta')$ mapping parts, together with an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signature morphism $\Sigma(\delta, c)\colon \Sigma(\Delta, c) \to \Sigma(\Delta', Cmp(\delta)(c))$ for each $c \in Cmp(\Sigma)$; a function $Prt(\delta)\colon Prt(\Delta) \to Prt(\Delta')$ mapping ports, together with an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signature morphism $Prt(\delta)(p)\colon \Sigma(\Delta, p) \to \Sigma(\Delta', Prt(\delta)(p))$, preserving

- the part owning each port p, i.e., $Cmp(\delta)(cmp(\Delta)(p)) = cmp(\Delta')(Prt(\delta)(p))$;
- the connections, i.e., if $(p, p') \in Con(\Delta)$, then $(Prt(\delta)(p), Prt(\delta)(p')) \in Con(\Delta')$.

The category of $\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})$-*signatures* $\mathbb{S}^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}$ consists of the composite structure signatures and signature morphisms over $\mathcal{M}^{\downarrow}_{\mathcal{D}}$.

For an $\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})$-signature Δ, a Δ-*composite structure structure* (sic!) over $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ is a family $\mathscr{C} \in (\mathscr{C}(c) \in |Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\Sigma(\Delta, c))|)_{c \in Cmp(\Delta)}$ consisting of an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-structure for each part c. The δ-*reduct* $\mathscr{C}'|\delta$ of a Δ'-composite structure structure $\mathscr{C}'$ over $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ along a composite structure signature morphism $\delta\colon \Delta \to \Delta'$ is computed component-wise as $(\mathscr{C}'(Cmp(\delta)(c))|\Sigma(\delta, c))_{c \in Cmp(\Delta)}$. The Δ-composite structure structures form the discrete category $Str^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}(\Delta)$ of $\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})$-*structures* over Δ. For each signature morphism $\delta\colon \Delta \to \Delta'$ in $\mathbb{S}^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}$ the δ-*reduct functor* $Str^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}(\delta)\colon Str^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}(\Delta') \to Str^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}(\Delta)$ is given by $Str^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}(\delta)(\mathscr{C}') = \mathscr{C}'|\delta$.

In UML, state machines organised in a composite structure communicate with each other by sending messages which are stored in event pools. A state machine draws a message from its event pool, which is typically implemented as an event queue, and reacts to this message by firing one of its enabled transitions or by discarding it when no transition is enabled. This communication scheme is obtained for a Δ-composite structure structure $\mathscr{C}$ over $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ by constructing an overall $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-structure over an $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signature that reflects the parts, the ports, and the connections in its events and attributes, but includes explicit event queues as additional attributes. The overall $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-structure over this queue-based $\mathcal{M}^{\downarrow}_{\mathcal{D}}$-signature then implements the selection of an event from a part's event queue, the reactions of this part to this event, and the distribution of the produced messages to the connected parts.

Formally, we construct a functor $\Sigma_{\mathrm{q}}\colon \mathbb{S}^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})} \to \mathbb{S}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}$ on signatures that assigns to a composite structure signature Δ the *queue-based* event/data signature $\Sigma_{\mathrm{q}}(\Delta) = \bigcup_{c \in Cmp(\Delta)}(\Sigma(\Delta, c) \cup \{\mathrm{q}_c : \hat{I}(\Sigma(\Delta, c))^*\})$ and to a composite structure signature morphism the canonically corresponding event/data signature morphism. For a composite structure signature Δ and a part $c \in Cmp(\Delta)$ there is a natural signature embedding $\eta^{\mathrm{q}}_{\Delta,c}\colon \Sigma(\Delta, c) \to \Sigma_{\mathrm{q}}(\Delta)$.

For a Δ-composite structure structure $\mathscr{C}$ we construct an overall $\Sigma_{\mathrm{q}}(\Delta)$-event/data structure $M_{\mathscr{C}}$ as follows: An overall configuration of $M_{\mathscr{C}}$ consists, for each part $c \in Cmp(\Delta)$, of an *event queue* $q(c) \in \hat{I}(\Sigma(\Delta, c))^*$ stored in the attribute q_c and a part configuration $\gamma(c) \in \Gamma(\mathscr{C}(c))$; initially, all parts are in some of their initial configurations and all event queues are empty. In an overall configuration $(q(c), \gamma(c))_{c \in Cmp(\Delta)}$ an overall transition to another overall configuration $(q'(c), \gamma'(c))_{c \in Cmp(\Delta)}$ *reacts* to some $\hat{\imath} \in \hat{I}(\Sigma_{\mathrm{q}}(\Delta))$ and *outputs* some $\hat{O} \in \hat{O}(\Sigma_{\mathrm{q}}(\Delta))^*$. This $\hat{\imath}$ can either instantiate some provided event $i \in I(\Sigma(\Delta, p_*))$ of some of the open ports $p_* \in Prt(\Delta)$ with

$c_* = cmp(\Delta)(p)$, or it is the head of the event queue of some $c_* \in Cmp(\Delta)$ such that $i \in I(\Sigma(\Delta, c_*))$. In the latter case, $\hat{\imath}$ is removed from the event queue of c_*. In both cases, the reaction of part c_* is any transition $(\gamma(c_*), \gamma'_*) \in R(\mathscr{C}(c))_{\hat{\imath},\hat{O}}$ and overall $\gamma' = \gamma\{c_* \mapsto \gamma'_*\}$. Finally, all outputs $p.\hat{o} \in \hat{O}$ such that $(p, p') \in Con(\Delta)$ and $cmp(\Delta)(p') = c'$ are appended to the respective event queue of part c'. This defines a natural transformation $Str_{\mathrm{q}}^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})} \colon Str^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})} \dot{\to} \Sigma_{\mathrm{q}}; Str^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}$ with $Str_{\mathrm{q},\Delta}^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}(\mathscr{C}) = M_{\mathscr{C}}$.

Theorem 3. $(\mathbb{S}^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}, Str^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}, \mathrm{Sen}^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}, \models^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})})$ *with* $\mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}} = \Sigma_{\mathrm{q}}; \mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}$ *and* $\mathscr{C} \models_{\Delta}^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})} \varrho$ *if, and only if,* $Str_{\mathrm{q},\Delta}^{\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})}(\mathscr{C}) \models_{\Sigma_{\mathrm{q}}(\Delta)}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}} \varrho$ *is an institution.*

$\mathrm{cs}(\mathcal{M}^{\downarrow}_{\mathcal{D}})$ inherits the event/data formulæ of $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ and the underlying $\mathcal{D}$, though extended by queue attributes. In particular, we have for a part $c \in Cmp(\Delta)$ that a transition sentence $\langle\!|i : \phi \mathbin{/\!\!/} O : \psi|\!\rangle \varrho$ (in the current configuration there are valuations and a transition for the incoming message and the outgoing messages such that these valuations satisfy transition formula ψ and ϱ holds afterwards) locally formulated for this part can be faithfully transferred to the global composite structure, abbreviating the embedding $\eta^{\mathrm{q}}_{\Delta,c}$ to η,

$$\begin{aligned}\langle\!|\eta(i) : \mathscr{F}^{\mathcal{D}}_{A(\eta),X(i)}(\phi) \wedge (\mathrm{hd}(\mathrm{q}_c) = I(\eta)(i) \vee \mathrm{open}_{\Delta,c}(I(\eta)(i))) \mathbin{/\!\!/}\\ O(\eta)(O) : \mathscr{F}^{2\mathcal{D}}_{A(\eta),X(i)\cup X(O)}(\psi) \wedge\\ \textstyle\bigwedge_{a\in A(\Sigma_{\mathrm{q}}(\Delta))\setminus(A(\Sigma(\Delta,c))\cup\{\mathrm{q}_c \mid c\in Cmp(\Delta)\})} a = a' \wedge\\ \mathrm{dist}_{\Delta,c}(I(\eta)(i), O(\eta)(O), (\mathrm{q}_c, \mathrm{q}'_c)_{c\in Cmp(\Delta)})|\!\rangle \mathrm{Sen}^{\mathcal{M}^{\downarrow}_{\mathcal{D}}}(\eta)(\varrho)\,,\end{aligned}$$

where hd yields the head of a queue, open checks whether the part's port for the event is open, the frame condition $a = a'$ ranges over all attributes not pertaining to c or the queues, dist removes the input and distributes the outputs to the queues.

6 Verification Example: Communication between `User`, `ATM` and `Bank`

We applied[4] the technique set out in this paper to the example from the introduction concerning a typical interaction between a `User`, an `ATM` component and a `Bank` component.

We formalised the state machines for the `Bank` and the `ATM` as well as their communication in CASL. We then set out to show a safety property (by means of a stronger invariant) on this system by inductive verification, as justified by Prop. 1. We first tried to show the preservation of said invariant using fully automatic provers connected to Hets [10], the main tool suite for verification based on CASL and institution theory. However, no inductive automated provers are currently connected to Hets. Therefore, handling freely generated datatype would require manual intervention to add suitable induction schemes — defeating our goal of automation. Instead we utilised the interactive theorem prover

[4] Full specifications and proofs accessible at: https://rosento.github.io/2021-paper-composite/

KIV [2]. This prover supports algebraic specifications similar to CASL and offers extensive heuristics for inductive proofs. KIV's heuristics fully automatically discharged all proof obligations in our experiments. The translation of the CASL specifications into KIV is straightforward.

With our process clarified, we can now state the safety property we will prove:

```
safe-def: safe(g) ↔ (ctrl(caConf(g)) = Verified → wasVerified(cbConf(g)) = 1);
used for: s, ls;
```

The above introduces an axiom `safe-def` defining the predicate `safe` and marks the axiom for use as a simplifier rule (`s`) and a local simplifier rule (`ls`) for the KIV system.

The predicate `safe` ranges over a type of system configurations, each consisting of the `ATM` configuration (`caConf`) and queue, as well as the bank configuration (`cbConf`) and queue. The machine configurations in turn consist of the control state and attributes. The safety predicate holds in a configuration *iff* should the `ATM` be in control state `Verified`, the bank attribute `wasVerified` has the value 1.

The behaviours of `Bank` and `ATM` are defined in the form of an initial state predicate and a transition predicate. For space reasons we show only one transition:

```
atmTrans-def: atmTrans(atmConf(sa1, c1, p1, t1), in, out, atmConf(sa2, c2, p2, t2))
  ↔ ∃ c : CardId, p : Pin . ...
       ∨ (   sa1 = CardEntered
          ∧ in = msg(userCom, PIN(p)) ∧ out = (msg(atmCompl, PINEnteredCompl) +l [])
          ∧ p2 = p ∧ sa2 = PINEntered ∧ c2 = c1 ∧ t2 = t1)
       ∨ ...; used for: s, ls;
```

The `ATM` transitions from one configuration to another, receiving an input event and sending out a list of messages. Each `ATM` configuration consists of (in that order) the control state, the card id to be verified, the PIN to be verified and the counter for the number of verification attempts. We give the definition of the transition predicate by a disjunction of the conditions of all syntactic transitions, including the control state before, the input event, the output list, variables to be set, the control state after and variables to remain unchanged. Given these machine predicates and a predicate `dist` to encode connectors, we can then define the transition predicate for the overall system:

```
trans-def: trans(conf(ca1, qa1, cb1, qb1), in, out, conf(ca2, qa2, cb2, qb2))
  ↔   dist(out, qa1, qa2, qb1, qb2)
    ∧ (   (atmTrans(ca1, in, out, ca2) ∧ cb2 = cb1 )
        ∨ (bankTrans(cb1, in, out, cb2) ∧ ca2 = ca1)); used for: s, ls;
```

Initially, the queues are empty and the machines are in their initial configurations.

Having thus defined the machines, we turn to verification and define an invariant strong enough to show both its own preservation and our safety property. The idea is to control the queues' status that allows us to enter the `Verified` state on the `ATM` or to reset the `wasVerified` attribute. In essence the invariant can be syntactically read off from the composite structure.

```
invar-def: invar(conf(ca, qa, cb, qb)) ↔ ∃ x.
    (ctrl(ca) = Idle        ∧ ctrl(cb) = Idle        ∧ qa = empty ∧ qb = empty)
  ∨ (ctrl(ca) = CardEntered ∧ ctrl(cb) = Idle        ∧ qa = empty ∧ qb = empty)
  ∨ (ctrl(ca) = PINEntered  ∧ ctrl(cb) = Idle        ∧ qa = enq(x, empty) ∧ qb = empty)
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = Idle        ∧ qa = empty ∧ qb = enq(x, empty))
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = Verifying   ∧ qa = empty ∧ qb = enq(x, empty))
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = VeriSuccess ∧
                qa = empty ∧ qb = enq(x, empty) ∧ wasVerified(cb) = 1)
```

```
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = VeriFail    ∧ qa = empty ∧ qb = enq(x,empty))
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = Idle        ∧
                qa = enq(msg(bankCom, reenterPIN), empty) ∧ qb = empty)
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = Idle        ∧
                qa = enq(msg(bankCom, verified), empty) ∧ qb = empty ∧ wasVerified(cb) = 1)
  ∨ (ctrl(ca) = Verified    ∧ ctrl(cb) = Idle        ∧
                qa = enq(x, empty) ∧ qb = empty ∧ wasVerified(cb) = 1); used for: s, ls;
```

Note that we can mostly ignore attribute values, as well as all distinctions between queue elements unrelated to our verification task. We can then formulate lemmas to the effect that this invariant does in fact imply the safety property, that it is satisfied in any legal initial configurations and that it is preserved by all transitions. These lemmas are as follows, again limited to one example for the transitions:

```
Safe: invar(g) → safe(g);
Init: init(g) → invar(g);
...
Trans6: g1 = conf(atmConf(Verifying, c, p, t), qa, cb, qb)
      ∧ qa ≠ empty ∧ top(qa) = msg(atmCom, verified)
      ∧ g2 = conf(atmConf(Verified, c, p, t),
                  enq(msg(atmCompl, VerifiedCompl), deq(qa)), cb, qb)
      ∧ invar(g1) → invar(g2);
```

Formulating separate lemmas for each transition instead of one lemma using the transition predicate helps us avoid a combinatorial explosion in the theorem prover.

Providing our specification to KIV with all definitions marked as simplifier rules and activating the heuristics mode “PL heuristics + structural induction”, each of our lemmas is proved without noticeable delay, i.e., the verification of the invariant is successful and does not pose any difficulty to the prover.

7 Conclusion

We have developed two new institutions extending the hybrid modal logic $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ [16]. One institution caters for simple UML state machines with outputs, an extension of it captures simple UML composite structure diagrams. Besides providing formal semantics for communicating UML state machines, via comorphisms these institutions provide a bridge towards theorem proving for UML. Through an elementary example we could demonstrate that, thanks to our framework, effective automated theorem proving for communicating UML state machines is possible.

Future work will be on proof automation. In particular we plan to implement the translations from UML into extended $\mathcal{M}^{\downarrow}_{\mathcal{D}}$, the institution comorphisms from extended $\mathcal{M}^{\downarrow}_{\mathcal{D}}$ to CASL, and possibly the link from Hets to KIV. Yet another important aspect is to implement analyses of the composite structure and its state machines with a view to automatically generate lemmas for automated theorem proving. In terms of our general research programme, the next topic to tackle are UML interactions and how they relate or refine to UML state machines. Going beyond the UML, it would be interesting to consider a truly heterogeneous framework, in which composite structure diagrams connect not only UML state machines, but also components specified in languages such as TLA or Event-B.

References

1. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic, Cambridge Tracts in Theoretical Computer Science, vol. 53. Cambridge University Press (2001)
2. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: Overview and VerifyThis Competition. Intl. J. Softw. Tools Technol. Transfer **17**(6), 677–694 (2015)
3. Goguen, J.A., Burstall, R.M.: Institutions: Abstract Model Theory for Specification and Programming. J. ACM **39**, 95–146 (1992)
4. Knapp, A., Mossakowski, T.: UML Interactions Meet State Machines — An Institutional Approach. In: Bonchi, F., König, B. (eds.) Proc. 7th Intl. Conf. Algebra and Coalgebra in Computer Science (CALCO 2017). LIPIcs, vol. 72, pp. 15:1–15:15 (2017)
5. Knapp, A., Mossakowski, T., Roggenbach, M.: Towards an Institutional Framework for Heterogeneous Formal Development in UML — A Position Paper. In: De Nicola, R., Hennicker, R. (eds.) Software, Services, and Systems — Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering, Lect. Notes Comp. Sci., vol. 8950, pp. 215–230. Springer (2015)
6. Knapp, A., Mossakowski, T., Roggenbach, M., Glauer, M.: An Institution for Simple UML State Machines. In: Egyed, A., Schaefer, I. (eds.) Proc. 18th Intl. Conf. Fundamental Approaches to Software Engineering (FASE 2015). Lect. Notes Comp. Sci., vol. 9033, pp. 3–18. Springer (2015)
7. Liu, S., Liu, Y., Étienne André, Choppy, C., Sun, J., Wadhwa, B., Dong, J.S.: A Formal Semantics for Complete UML State Machines with Communications. In: Johnsen, E.B., Petre, L. (eds.) Proc. 10th Intl. Conf. Integrated Formal Methods (IFM 2013). Lect. Notes Comp. Sci., vol. 7940, pp. 331–346. Springer (2013)
8. Mazzanti, F., Ferrari, A., Spagnolo, G.O.: The KandISTI/UMC Online Open-Access Verification Framework. ERCIM News **109** (2017)
9. Mossakowski, T.: Relating CASL with Other Specification Languages: The Institution Level. Theo. Comp. Sci. **286**(2), 367–475 (2002)
10. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In: Grumberg, O., Huth, M. (eds.) Proc. 13th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007). Lect. Notes Comp. Sci., vol. 4424, pp. 519–522. Springer (2007)
11. Mosses, P.D.: CASL Reference Manual — The Complete Documentation of the Common Algebraic Specification Language, Lect. Notes Comp. Sci., vol. 2960. Springer (2004)
12. Ober, I., Dragomir, I.: Unambiguous UML Composite Structures: The OMEGA2 Experience. In: Cerná, I., Gyimóthy, T., Hromkovic, J., Jeffery, K.G., Královic, R., Vukolic, M., Wolf, S. (eds.) Proc. 37th Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM 2011), Lect. Notes Comp. Sci., vol. 6543, pp. 418–430. Springer (2011)
13. Object Management Group: Unified Modeling Language. Standard formal/2017-12-05, OMG (2017), https://www.omg.org/spec/UML/2.5.1
14. Object Management Group: Precise Semantics of UML Composite Structures. Standard formal/2019-02-01, OMG (2019), https://www.omg.org/spec/PSCS/1.2
15. Rosenberger, T.: Relating UML State Machines and Interactions in an Institutional Framework. Master's thesis, Universität Augsburg, Ludwig-Maximilians-Universität München, Technische Universität München (2017)
16. Rosenberger, T., Bensalem, S., Knapp, A., Roggenbach, M.: Institution-based Encoding and Verification of Simple UML State Machines in CASL/SPASS. In: Roggenbach, M. (ed.) Rev. Sel. Papers 25th Intl. Ws. Recent Trends in Algebraic Development Techniques (WADT 2020). Lect. Notes Comp. Sci., vol. 12669, pp. 120–141. Springer (2020)

Semantic Code Search in Software Repositories using Neural Machine Translation

Evangelos Papathomas(✉), Themistoklis Diamantopoulos, and Andreas Symeonidis

Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
Thessaloniki, Greece
epapathom@ece.auth.gr, thdiaman@issel.ee.auth.gr, symeonid@ece.auth.gr

Abstract. Nowadays, software development is accelerated through the reuse of code snippets found online in question-answering platforms and software repositories. In order to be efficient, this process requires forming an appropriate query and identifying the most suitable code snippet, which can sometimes be challenging and particularly time-consuming. Over the last years, several code recommendation systems have been developed to offer a solution to this problem. Nevertheless, most of them recommend API calls or sequences instead of reusable code snippets. Furthermore, they do not employ architectures advanced enough to exploit the semantics of natural language and code in order to form the optimal query from the question posed. To overcome these issues, we propose CodeTransformer, a code recommendation system that provides useful, reusable code snippets extracted from open-source GitHub repositories. By employing a neural network architecture that comprises advanced attention mechanisms, our system effectively understands and models natural language queries and code snippets in a joint vector space. Upon evaluating CodeTransformer quantitatively against a similar system and qualitatively using a dataset from Stack Overflow, we conclude that our approach can recommend useful and reusable snippets to developers.

Keywords: code reuse · semantic analysis · neural transformers.

1 Introduction

The wide uptake of open-source software in the last few decades has accelerated software development through code reuse. Nowadays, developers search online for ways to solve issues that arise during the development process, such as writing code for complex tasks, integrating APIs, or fixing bugs. The popularity of this paradigm has been further boosted from the introduction of online repositories (e.g. GitHub) and programming communities (e.g. Stack Overflow).

As code reuse has become a vital aspect of today's software development, the challenge of finding appropriate answers to programming-related questions in the vastness of the Internet led to the development of code recommendation systems. While the majority focus on providing API calls and sequences (e.g.

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 225–244, 2022.
https://doi.org/10.1007/978-3-030-99429-7_13

DeepAPI [10]), a selected few have the advantage of recommending reusable code snippets (e.g. DeepCS [9]). Such systems that employ whole snippet extraction mechanisms are greatly valued, as they significantly reduce development time.

However, they are also prone to important limitations. Many accept queries in specialized query languages instead of natural language. In addition, most systems do not employ mechanisms advanced enough to extract the semantics found both in the queries and the source code. And even though some systems engage in semantic analysis (e.g. DeepCS [9], CodeSearchNet [12]), crucial information, such as the control flow of a code snippet, is discarded. Finally, the aforementioned systems typically employ non-annotated datasets and, by extension, lack in terms of training and quantitative evaluation, as ground truth data are essential for the training of a system and the assessment of its performance.

Acknowledging the need for advancing code reuse, GitHub initiated the CodeSearchNet challenge [12], a public competition for code search, specifically aiming to improve on four baseline models using an annotated dataset. These models receive queries in natural language and employ different neural network architectures to return high-quality code snippets. The CodeSearchNet challenge overall provides an interesting testbed due to the variety of programming languages and code snippets in the dataset and the evaluation tools offered.

Given influence by this challenge, in this paper we present CodeTransformer, a system that receives natural language queries and provides reusable code snippets. CodeTransformer uses state-of-the-art neural network and language understanding techniques, while it also employs a custom similarity metric and a custom loss function. Our system does not require some specialized query language; instead, it receives queries in natural language and employs neural machine translation to offer reusable snippets in the form of methods. We train our system on a state-of-the-practice annotated dataset and evaluate its effectiveness against the baseline CodeSearchNet systems [12]. Finally, we assess its applicability in a question-answering context using data from Stack Overflow.

2 Related Work

Code search systems can be distinguished into two categories, those producing sequences of API calls and those producing reusable code. The first category includes systems such as SWIM [21] and T2API [19], which translate text queries to API calls and then synthesize their usage code, i.e. code that uses the calls. SWIM extracts API calls related to a query using Bing and forms their usage code, including the control flow. A limitation is that it cannot handle the semantics of queries (e.g. "convert int to string" and "convert string to int"). T2API is trained on Stack Overflow posts and uses the GraLan language model [17] to model dependencies between API calls and synthesize their usage code.

A different approach to API call recommendation is taken by MULAPI [24]. Apart from usage examples, MULAPI also analyzes the source code and API libraries of a project to provide an implementation of the requested feature. The system also maps the repository of the code to recommend files as locations for

the provided API usage code. The architecture of MULAPI comprises a Stanford Word Segmenter for text preprocessing and a Vector Space Model to assess the similarity between texts. FOCUS [18] is a similar system that analyzes a project's repository and other open source repositories using Abstract Syntax Trees and assesses their similarity using Context-Aware Correlation Filter. Next, it mines API calls from the most similar repositories and presents them to the developers.

Other systems treat code recommendation as a machine translation problem. One of them is DeepAPI [10], which utilizes a Neural Network architecture to transform natural language queries to API sequences. It consists of a recurrent neural network (RNN) encoder that processes natural language using attention mechanisms and an RNN decoder using an Inverse Document Frequency (IDF)-based weighting mechanism to output API sequences. BIKER [4] is a similar system that receives natural language queries and assesses their similarity to Stack Overflow question posts and API documentation. Post texts and code snippets are handled as text and are used to train an embedding model that takes into account IDF weights, and recommends relevant API calls.

Word2API [15] also bridges the semantic gap between natural language and code to provide API recommendations. The system creates tuples of method descriptions and API sequences that are used to train a word embedding model for vector generation. A more advanced approach was implemented by DeepAPIRec [6]. Its architecture consists of Tree-LSTMs, a long short-term memory (LSTM) unit variant that organizes information in an inverse tree structure. DeepAPIRec also utilizes a statistical parameter model of data dependency that allows recommending parameter values for the APIs suggested by the Tree-LSTM.

The second category of systems comprises the ones that recommend reusable code snippets instead of API calls. One of them is Seahawk [20], an Eclipse plugin that, given a query, returns a ranked list of relevant Stack Overflow posts. The posts are retrieved using Apache Solr and ranked using tf-idf. The snippets found in the posts can be integrated into the code of a project. Like Seahawk, NLP2Code [5] is an Eclipse plugin that retrieves code snippets from Stack Overflow posts. NLP2Code processes natural language text and snippets using the TaskNav algorithm and measures their grammatical correlation with the Stanford CoreNLP Toolkit. The system receives natural language queries and employs a customized version of Google Search Engine for search. StackSearch [8] also extracts information from Stack Overlow posts and recommends code snippets using a hybrid language model that combines Tf-Idf and fastText [3]. Its results are also accompanied with labels extracted using named entity recognition.

An interesting alternative is DeepCS [9], which recommends reusable code snippets given a natural language query. DeepCS employs two RNN encoders, one that receives natural language descriptions of methods and one that receives a fusion of method names, API sequences and code tokens. Then the system max pools the embeddings generated by the two encoders and assesses their similarity using cosine similarity. DeepCS can understand the semantics of natural language and code to a specific extent, however it relies on the generated vectors to rank its results without considering more code features such as context.

In contrast to systems that utilize raw data dumps from Stack Overflow or code repositories, CodeSearchNet [12] introduced a well curated dataset specifically designed for semantic code search, as it consists of docstring and code tokens which highlight their semantics while also facilitating the preprocessing. Moreover, it introduced four different baselines, each using a different architecture for its encoders (Neural Bag-of-Words, Bidirectional RNNs, 1D Convolutional Neural Networks and Self-Attention). CodeSearchNet outperforms most systems due to the quality of its dataset and its powerful neural architectures. However, it ignores certain semantics, such as the control flow of the code, so it favors keyword-based methods instead of those using semantic information.

Although the aforementioned systems are effective in certain scenarios, they have important limitations. Most of them handle natural language input as keywords, i.e. measuring token frequency instead of analyzing semantics and context. Also, most systems output API calls or API usage code instead of reusable snippets. Deep learning systems often do not employ custom similarity metrics and loss functions. CodeTransformer, is trained on high-quality annotated data from the CodeSearchNet corpus. It analyzes the query and code semantics using word embeddings, generated with state-of-the-art attention mechanisms. We employ a hybrid similarity metric and build a custom loss function that are suited to the challenge at hand. Thus, our system is able to comprehend relations between similar queries (e.g. "how to write to command line" and "how to output to terminal") and distinguish queries with lexically minor, yet semantically major differences (e.g. "convert int to string" and "convert string to int").

3 Semantic Code Search using Machine Translation

The architecture of our system, shown in Figure 1, comprises four modules: the Dataset Builder, the Neural Network, the Index Builder, and the Search Engine. The Dataset Builder preprocesses the natural language and code data to produce a clean dataset, including the vocabularies of the input and target languages. The Neural Network module generates word embeddings and extracts the most important features per language using attention mechanisms.

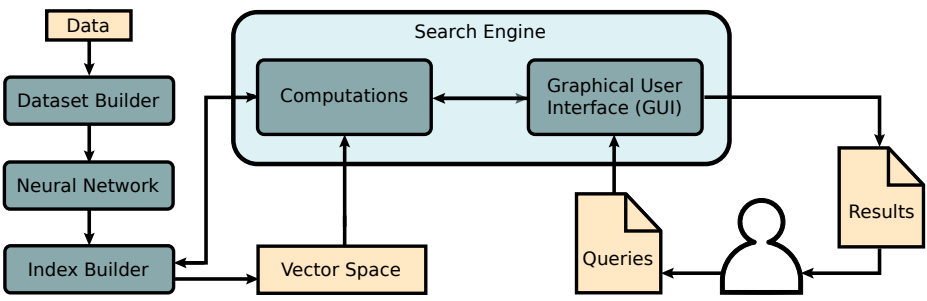


Fig. 1. The architecture of CodeTransformer

Max pooling is used on the word embeddings to generate a single embedding for each natural language and code sequence. The Index Builder builds a vector space containing the sequence embeddings. Each code vector is assigned to an index to allow nearest neighbor search when a natural language vector is received. The Search Engine receives an input query in the GUI and forwards it to the Computations submodule, where the Neural Network analyzes it and generates a natural language sequence embedding. This vector representation of the query is inserted in the vector space to search for its nearest code vectors. The results are forwarded back to the GUI and presented to the user. These modules are further analyzed in the following subsections.

3.1 Data Preprocessor

Dataset Overview The CodeSearchNet corpus comprises over 6.4 million code snippets written in 6 languages, with over 2.3 million of them annotated using docstrings [12]. The snippets were extracted from GitHub repositories, and filtered to remove test functions/constructors, trim long docstrings, and apply de-duplication [16,1]. CodeTransformer was implemented using the Java dataset of the corpus that contains over 1.5 million snippets, of which over 0.54 million come with docstrings. Although we use Java as a proof of concept, it is important to note that our system is mostly language agnostic. Our methodology can be applied to other languages, e.g. Python or JavaScript, with minimal changes.

For each snippet, the dataset contains fields about its origin (repo, path, url, sha) and fields concerning its data (original/full string, method name, extracted code and docstring). The code and the documentation of the snippet (docstring) are also provided as tokens. Table 1 depicts a sample entry of the dataset.

Table 1. An example entry of the dataset

Features	Data
func_name	JsonObjectDeserializer.getRequiredNode
docstring	/** * Helper method to return a {@link JsonNode} from the tree. * @param tree the source tree * @param fieldName the field name to extract * @return the {@link JsonNode} */

code	protected JsonNode getReqNode(JsonNode tree, String fieldName){
	Assert.notNull(tree, "Tree must not be null");
	JsonNode node = tree.get(fieldName);
	Assert.state(node != null && !(node instanceof NullNode), () ->
	"Missing JSON field '" + fieldName + "'");
	return node;
	}

After manual inspection, we concluded that the majority of the dataset entries contain valid natural language docstrings, extracted from each function. However, in certain entries the snippets are not properly annotated and in others the automated natural language text extractor has failed to extract the docstring correctly. For instance, in the docstring of Table 1, the extracted docstring tokens are ['helper', 'method', 'to', 'return', 'a', '{']. To avoid having docstrings that are incorrect or are not properly tokenized, we first preprocess the dataset.

Data Preprocessing We create two separate preprocessing pipelines to effectively target the docstrings and the code data. The regular expressions of Table 2 enable modifications in the tokens of the dataset.

Table 2. Regular expressions for preprocessing

Regex Name	Regular Expression
remove_non_ascii	`[^\x00-\x7f]`
remove_special	`[^A-Za-z0-9]+`
seperate_strings	`[A-Z][a-z][^A-Z]*`
fill_empty	`[A-Z][a-z][^A-Z]*|[A-Z]*(?![a-z])|[A-Z][a-z][^A-Z]*`
remove_unnecessary	`(\s)|(")|(^//)|(^/\*)|(^/\*\*)`
replace_symbols	`^[()[\]{}<>+\-*/^%=&|!?@\.,:;]`

For the removal of noisy natural language data, we designed a pipeline of preprocessing steps, as described below:

1. We remove all the tokens of the docstring located after the first dot symbol encounter, thus reducing their size to that of typical natural language queries.
2. The *remove_non_ascii* and *remove_special* expressions are used to replace all non-ASCII characters and all special characters, respectively, in the tokens of the docstring list with empty characters.
3. The *separate_strings* expression is used to separate all the camelCase tokens of the docstring list and thus augment the data for the neural network.
4. We empty all docstring lists that contain less than 6 or more than 30 tokens[1] as inefficient or lengthy, respectively. The lists are filled with the corresponding camelCase function names and separated using the *fill_empty* expression.
5. All uppercase characters in the docstring tokens are converted to the corresponding lowercase characters, to achieve structural uniformity between tokens with the same meaning but different writing format.

As an example, the docstring of the snippet shown in Table 1 produces the tokens ['helper', 'method', 'to', 'return', 'json', 'node', 'from', 'the', 'tree'].

[1] The limits were defined after studying the data and concluding that most entries with inefficient docstrings contained less than 6 docstring tokens, while also noting that 30 tokens are adequate for a well-defined description of a function.

Concerning noisy code data, we designed a preprocessing pipeline that slightly differs from those of other systems. Most systems do not sufficiently exploit the control flow information of a code snippet. Instead, they solely focus on function and variable names, as well as control flow words, such as if, else, for, etc. To fully exploit the programming symbols of snippets, we perform the following steps:

1. The *remove_non_ascii* and *separate_strings* expressions are used to remove all non-ASCII characters and split the text to tokens.
2. We remove all the tokens of the code list that contain space, double quotes, or create a comment using the *remove_unnecessary* expression.
3. We encode programming symbols to unique tokens, as shown in Table 3.

Table 3. The encoding of programming symbols to unique tokens

#	Unique Token	#	Unique Token	#	Unique Token
(	openingparen	*	multiplyoperator	<	lessoperator
)	closingparen	/	divideoperator	>=	greaterequaloperator
[	openingbracket	^	poweroperator	<=	lessequaloperator
]	closingbracket	%	modulooperator	++	incrementoperator
{	openingbrace	=	assignoperator	−−	decrementoperator
}	closingbrace	==	equaloperator	!	notoperator
+	addoperator	!=	notequaloperator	@	atsign
−	subtractoperator	>	greateroperator	;	semicolon

4. The *remove_special* regular expression is used to remove all the non-alphanumeric characters in the tokens of the code list with empty characters. This step removes symbols that were not replaced in the previous step.
5. We limit the length of the code lists to their first 100 tokens, trimming methods of great length and thus enhancing the uniformity of the dataset. Also, all uppercase characters in the code tokens are converted to the corresponding lowercase ones, as in the docstrings, to favor structural uniformity.

As an example, the code of the method snippet shown in Table 1 produces the tokens shown in Figure 2.

'protected', 'json', 'node', 'get', 'required', 'node', 'openingparen', 'json', 'node', 'tree', 'string', 'field', 'name', 'closingparen', 'openingbrace', 'assert', 'not', 'null', 'openingparen', 'tree', 'tree', 'must', 'not', 'be', 'null', 'closingparen', 'semicolon', 'json', 'node', 'node', 'assignoperator', 'tree', 'get', 'openingparen', 'field', 'name', 'closingparen', 'semicolon', 'assert', 'state', 'openingparen', 'notequaloperator', 'null', 'notoperator', 'openingparen', 'node', 'instanceof', 'null', 'node', 'closingparen', 'openingparen', 'closingparen', 'missing', 'json', 'field', 'addoperator', 'field', 'name', 'addoperator', 'closingparen', 'semicolon', 'return', 'node', 'semicolon', 'closingbrace'

Fig. 2. Example tokens extracted from the code of the method snippet of Table 1

Our preprocessing pipeline minimizes the loss of information by performing data augmentation on docstrings and code. In docstrings where the information is insufficient, the pipeline replaces them with separated camelCase function names (e.g. 'camelCase' becomes 'camel case') that are representative of the code. The pipeline also encodes most code symbols to words instead of removing them and, thus, reinforces code semantics such as control and data flow.

3.2 Neural Network

In this subsection we present the main module of our system, a neural network that employs transformers to map natural language queries to source code.

Network Architecture The main architecture of CodeTransformer is based on Matching Networks [23], a neural network architecture designed to solve One-Shot Learning problems. Our system, however, follows a slightly different approach, as it uses an improved embedding similarity metric and does not require an external memory to function. As we discuss in the following subsections, our architecture utilizes self-attention encoders and a hybrid geometric similarity metric. In contrast to the original approach, ours does not use a softmax function on its output, as the similarity metric we selected does not natively support it. In Figure 3 we present the architecture of the Neural Network module.

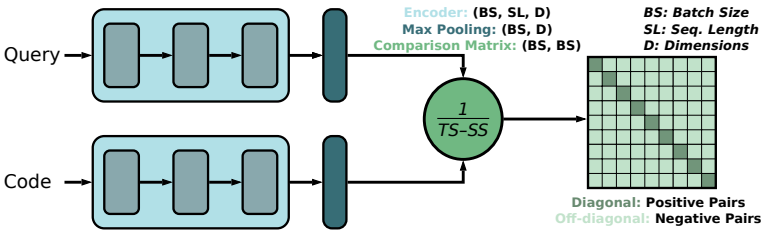


Fig. 3. The main architecture of the Neural Network module

Transformers To maximize the semantic abilities of our system, we employed the state-of-the-art Transformers architecture on both of its encoders [22]. A Transformer consists of two modules, an encoder and a decoder, with minimal architectural differences. Considering the fact that a Matching Network performs feature extraction and not direct translation of language data, our implementation solely requires encoders for its function. The architecture of the Transformer encoder is presented in Figure 4.

The Transformer encoder comprises an embedding layer, a Positional Encoding layer and encoder layers, i.e. consecutive blocks of Multi-Head Attention and Feed-Forward Network layers. In our implementation, we opted for three stacked encoder layers, as they provide sufficient depth for achieving high efficiency.

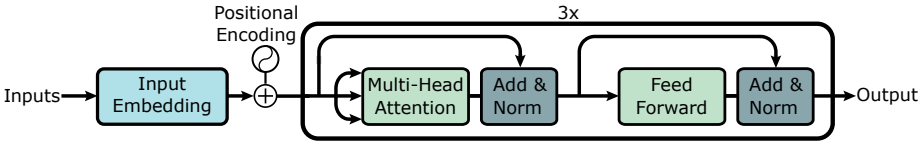


Fig. 4. The architecture of a Transformer encoder

Before inserting a token sequence to an encoder, we create a vocabulary that includes the most frequently occurring words and then encode them to integers. We build two vocabularies, each consisting of 10,000 unique words. After encoding, we pad each entry with zeros to form tensors of equal dimensions. To enhance the generalization capabilities of our system, we reshuffle the dataset at the start of every training iteration and divide it in batches of 128.

When a token sequence is received as input, the encoder embeds the tokens in a high-dimensional vector space. In other words, the encoder generates word embeddings, i.e. vector representations aiming to extract token information. The encoder generates word embeddings of 128 dimensions using an embedding layer. The natural language encoder and the source code encoder have identical parameter values, but each encoder has its own distinct weights and vocabulary. To generate sequence embeddings we use max pooling, as extracts the most essential features of the embeddings outputted from the stacked encoder layers.

Similarity Metric The similarity between natural language and code sequence embeddings is usually quantified using the Euclidean distance or the cosine similarity. However, the computation of the Euclidean distance between two vectors does not contain any information about the angle between the two vectors. On the other hand, cosine similarity does not consider the magnitude of the vectors.

Our system utilizes a hybrid similarity metric, the *Triangle's Area Similarity - Sector's Area Similarity* [11], also known as *TS-SS*, which improves upon the aforementioned metrics by incorporating the Euclidean distance, the magnitude difference and the angle between two vectors to compute their similarity. The Triangle's Area Similarity (TS) comprises the Euclidean distance, the magnitude of each vector and the angle between them, while the Sector's Area Similarity (SS) provides the magnitude difference. The TS of two vectors A and B is:

$$TS\left(A,B\right)=\frac{|A|\cdot|B|\cdot\sin\left(\theta'\right)}{2} \tag{1}$$

where, given θ is the angle between the two vectors, θ' is defined as $\cos^{-1}\left(\theta\right)+10°$. We use θ' instead of θ so that the computation is valid in the case of overlapping vectors (when $\theta=0$). The SS of two vectors A and B is defined as:

$$SS\left(A,B\right)=\pi\left(ED\left(A,B\right)+MD\left(A,B\right)\right)^{2}\cdot\left(\frac{\theta'}{360}\right) \tag{2}$$

where θ' is defined as above, while $ED\left(A,B\right)$ and $MD\left(A,B\right)$ correspond to the Euclidean distance and the magnitude difference between the two vectors,

respectively. Given the dimension of the vectors N, the magnitude difference is:

$$MD\left(A,B\right) = \left| \sqrt{\sum_{n=1}^{N} A_n^2} - \sqrt{\sum_{n=1}^{N} B_n^2} \right| \tag{3}$$

Merging TS and SS via addition is not possible, as they are in different scale. According to Heidarian and Dinneen [11], their multiplication establishes a new scale that sufficiently represents similarity. Consequently, TS-SS is computed as:

$$TS{-}SS\left(A,B\right) = \frac{|A| \cdot |B| \cdot \sin\left(\theta'\right) \cdot \theta' \cdot \pi \cdot \left(ED\left(A,B\right) + MD\left(A,B\right)\right)^2}{720} \tag{4}$$

TS-SS values range from 0 to infinity, with 0 indicating that two vectors are identical. Accordingly, the TS-SS value of two dissimilar vectors is larger than zero, without any limitations. In our implementation, we decided to calculate the reciprocal TS-SS in favor of the custom loss function we use during our network's training process. The final similarity of the two vectors is computed as:

$$Similarity\left(A,B\right) = \frac{1}{TS{-}SS\left(A,B\right)} \tag{5}$$

Loss Function The neural network of CodeTransformer outputs a square similarity matrix, where each row represents a natural language embedding and each column represents a source code embedding. The diagonal matrix cells correspond to the positive pairs of natural language and source code and their values ought to be high. The rest of the matrix cells correspond to the negative pairs, and their values ought to be low. At network initialization, all embeddings contain random values and are scattered throughout the vector space. As a result, in order to bring all similar embeddings closer during training, we need to utilize a loss function that is based on the computations of the reciprocal TS-SS.

A loss function typically used by similar systems (such as CodeSearchNet [12] and DeepCS [9]) is a variation of Hinge loss, computed as follows:

$$Loss = \max\left(0, 1 - positive + negative\right) \tag{6}$$

Upon testing this variation of Hinge loss, we observed that it did not result in successful integration with the vanilla or reciprocal TS-SS. Even after modifying the function's margin to a value larger than 1, due to TS-SS infinite value range, the result was always the same. The embeddings constantly collapsed to a specific point, not allowing distinct sequence embeddings for each positive pair.

This led us to design a custom loss function, based on the squared variation of Hinge loss. We name this loss function *Squared Margin Loss* and define it as:

$$Loss = \left(\max\left(0, margin - positive\right)\right)^2 + negative^2 \tag{7}$$

Furthermore, the derivatives of our loss function are defined as follows:

$$\frac{\partial}{\partial\left(positive\right)} Loss = \begin{cases} 2 \cdot \left(margin - positive\right), & \text{if } positive < margin \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

$$\frac{\partial}{\partial\,(negative)} Loss = 2 \cdot negative \tag{9}$$

The Squared Margin Loss encourages the penalization of larger loss values more, and the penalization of smaller loss values less. Thus, the function ensures the convergence of the network at first epochs and its optimization at later epochs.

By further restricting the function with the function *max*, the positive pairs of similarity value above the margin do not take part in the computation of the loss. In this case, however, the similarity values of the corresponding negative pairs continue to decrease. This allows the similarity values of the diagonal to increase further than the margin. Without the use of the max function, the elements that have crossed the margin would generate useless losses and positive gradients, resulting in the fluctuation of their similarity values around the margin.

Optimizer We train our neural network using the *Adaptive Moment Estimation (Adam)* optimizer [14], which computes adaptive learning rates for each parameter. Adam stores the exponentially decaying average of past gradients and the exponentially decaying average of past squared gradients. Using Adam ensures that the network converges fast through momentum estimation. The convergence also depends on the learning rate; a poor choice of its value can slow down the training process, or even derail the network's weights. To find the ideal learning rate, we examined a range of values generated by the equation:

$$LearningRate = 1.1^{step/100} \cdot 10^{-10} \tag{10}$$

This function generates values starting from 10^{-10} up to a practically infinite value. The learning rate is multiplied by 1.1 once every 100 training steps.

After plotting the accuracy and loss per training step, we noticed a point with a steep increase in accuracy and a steep decrease in loss as well as a point with a steep decrease in accuracy and a steep increase in loss. Next, we isolated the values between these steps and tested those closer to the lower end, where the increase in accuracy and decrease in loss occur. Through trial and error, we selected a learning rate value of $3.2 \cdot 10^{-4}$. We set the margin of our network to 5, the number of heads to 8, and the dff to 512, and trained the network for 40 epochs, as these have been shown to be enough for the efficiency of the results.

3.3 Index Builder

Due to the complexity of our neural network and the number of its parameters, fast response times cannot be guaranteed. To significantly reduce the processing time of our system, we employed Annoy [2], a tool using a Nearest Neighbor Search algorithm. Using Annoy in the Index Builder module allows us to generate a vector space that contains all the source code embedding vectors of the corpus.

Annoy assigns an index on all code embeddings and then assorts them based on their values by building up a forest of trees. The vector dimension of the vector space is set according to the dimension of the output embedding, which is

128. We calculated the Euclidean distance between the vectors and built 10 trees. Regarding the search process in the vector space, we select the first 100 nearest vectors out of the 10.000 nearest forest nodes. Thus, instead of calculating the similarity between a query and the whole corpus using the neural network, Annoy compares the query vector with the nearest 10.000 code vectors. In addition, Annoy's search time does not seem to be hindered by the embedding dimension.

The search process of a query is executed in three stages. Firstly, the query of the user is preprocessed, so that non-alphanumeric symbols are removed, camelCase tokens are separated and uppercase characters are lowercased. Secondly, every query token is encoded as an integer to be passed as input to the neural network. The neural network, in inference mode, generates the sequence embedding of the query to be inserted to the vector space of the Index Builder. Finally, Annoy extracts the indices of the 10 code vectors nearest to the query, and the corresponding code snippets and GitHub URLs are presented to the user.

4 Evaluation

We evaluate our system using two different datasets, the Java corpus of CodeSearchNet [12], and a set of popular Java questions from Stack Overflow[2].

The performance of our system is assessed using the *Precision at K (P@K)*, the *Mean Reciprocal Rank (MRR)* [7] and the *Normalized Discounted Cumulative Gain (NDCG)* [13]. P@K indicates how many out of the first K results are relevant to the query. MRR further incorporates the order of the results, computed as the mean of the reciprocal rank of each query (the reciprocal rank of the i-th query is $1/rank_i$, where $rank_i$ is the rank position of the first relevant document). The NDCG is the normalized DCG, computed for N results as:

$$DCG = \sum_{i=1}^{N} \frac{2^{rel_i} - 1}{log_2\,(i+1)} \tag{11}$$

where rel_i is the graded relevance of the result at position i. Thus, NDCG is computed dividing the result of equation (11) by the ideal DCG, i.e. the one produced if all the results in the list were sorted in the correct order.

4.1 Evaluation using CodeSearchNet Queries

CodeTransformer employs the CodeSearchNet corpus [12] for training and inference, allowing its direct comparison with the implementations of CodeSearchNet. CodeSearchNet comprises four different encoder architectures. One of them is the *Self-Attention (SelfAtt)* architecture, which was examined in the previous section. The *Neural Bag of Words (NBoW)* architecture measures word occurrence within a document, therefore it performs well on keyword-based search operations. The *1D Convolutional Neural Network (1D-CNN)* architecture learns to

[2] The code and details used to reproduce our findings can be found at the repository: https://github.com/AuthEceSoftEng/CodeTransformer

recognize complex, non-linear patterns. In contrast to NBoW and 1D-CNN, the *Bidirectional RNN (biRNN)* architecture further models the word order.

The four implementations are compared to CodeTransformer on the test set of the CodeSearchNet corpus, which includes 15000 docstring and code snippet pairs, for the computation of MRR. Additionally, the four implementations are compared to CodeTransformer using 99 annotated queries provided by CodeSearchNet for computing NDCG. The results are shown in Table 4. Note that, although our system is not directly compared with DeepCS [9] as the systems use different data, we compare it with the *biRNN* implementation of CodeSearchNet that has a similar neural architecture with DeepCS.

Table 4. Evaluation results of CodeTransformer and CodeSearchNet

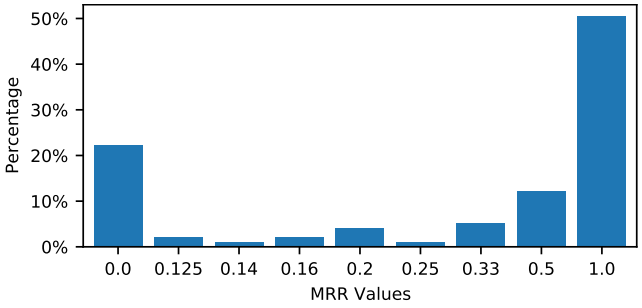
System	MRR	NDCG
CodeSearchNet-NBoW	0.5140	0.1207
CodeSearchNet-1D-CNN	0.5270	0.1282
CodeSearchNet-biRNN	0.2865	0.0623
CodeSearchNet-SelfAtt	0.5866	0.1003
CodeTransformer	0.6263	0.1028

Concerning MRR, our system outperforms CodeSearchNet measurements, indicating that the different strategies followed for our data pipeline are effective. Another factor that may contribute to this result is our preprocessing methodology, as it may be possible that the replacement of insufficient docstrings with function names led to increased MRR values. As a side note, these results were also clear during the validation phase of the algorithms (e.g. the MRR of CodeTransformer for the validation set was the highest at 0.62604, while the second highest was that of CodeSearchNet-SelfAtt at 0.5513).

Concerning NDCG, our system performs slightly better compared to the corresponding Self-Attention implementation of CodeSearchNet, while the NBoW and 1D-CNN implementations perform better than CodeTransformer, possibly because they use docstrings as natural language. However, we note that only a small amount of data was annotated for the computation of NDCG (i.e. only 823 out of 1.5 million Java code snippets). In addition, as the authors of CodeSearchNet note [12], the annotated data were selected using the top 10 results per query, generated by an ensemble of the CodeSearchNet neural models and ElasticSearch, therefore they are what these systems are more likely to produce. Hence, it is possible that correct results are ignored for computing NDCG.

Figure 5 depicts the distribution and the individual MRR values for 99 queries of the test set of CodeSearchNet [12]. As the annotations were not provided, we annotated the first 10 results returned by our system to compute the MRR. The majority of MRR values are equal to 1, indicating that our system returns a relevant result in the first position for more than half of the queries. By examining the results, we found that our system effectively models the semantic informa-

tion of the text and the code snippets. Indicatively, for Q64, CodeTransformer outputs a function that sorts an array using another array's order, even though almost none of the exact words of the query are present in the code (except for the word "sort"). Semantically similar terms are also effectively interpreted. E.g., for Q16 that requests exporting data to an excel file, our system returns an exportXls method, thus modeling the semantic similarity between terms "excel" and "xls". Similarly, given Q91 that requests data extraction from a text file, CodeTransformer returns a method using the term "read" instead of "extract".



Q01: convert int to string	1.000	Q51: how to randomly pick a number	1.000
Q02: priority queue	0.500	Q52: normal distribution	1.000
Q03: string to date	1.000	Q53: nelder mead optimize	0.000
Q04: sort string list	1.000	Q54: hash set for counting distinct elements	0.000
Q05: save list to file	0.500	Q55: how to get database table name	1.000
Q06: postgresql connection	1.000	Q56: deserialize json	0.500
Q07: confusion matrix	1.000	Q57: find int in string	1.000
Q08: set working directory	1.000	Q58: get current process id	1.000
Q09: group by count	0.000	Q59: regex case insensitive	0.160
Q10: binomial distribution	0.200	Q60: custom http error response	1.000
Q11: aes encryption	1.000	Q61: how to determine a string is a valid word	0.200
Q12: linear regression	1.000	Q62: html entities replace	0.330
Q13: socket recv timeout	0.000	Q63: set file attrib hidden	0.330
Q14: write csv	1.000	Q64: sorting arrays based on another arrays order?	1.000
Q15: convert decimal to hex	1.000	Q65: string similarity levenshtein	1.000
Q16: export to excel	1.000	Q66: how to get html of website	0.000
Q17: scatter plot	1.000	Q67: buffered file reader read text	0.500
Q18: convert json to csv	0.160	Q68: encrypt aes ctr mode	0.000
Q19: pretty print json	1.000	Q69: matrix multiply	0.500
Q20: replace in file	0.500	Q70: print model summary	0.000
Q21: k means clustering	1.000	Q71: unique elements	0.500
Q22: connect to sql	1.000	Q72: extract data from html content	0.000
Q23: html encode string	1.000	Q73: heatmap from 3d coordinates	0.000
Q24: finding time elapsed using a timer	0.125	Q74: get all parents of xml node	0.000
Q25: parse binary file to custom class	0.500	Q75: how to extract zip file recursively	1.000
Q26: get current ip address	1.000	Q76: underline text in label widget	0.000
Q27: convert int to bool	0.250	Q77: unzipping large files	0.200
Q28: read text file line by line	1.000	Q78: copying a file to a path	0.500
Q29: get executable path	1.000	Q79: get the description of a http status code	1.000
Q30: httpclient post json	0.500	Q80: randomly extract x items from a list	1.000
Q31: get inner html	0.500	Q81: convert a date string into yyyymmdd	0.330
Q32: convert string to number	0.000	Q82: convert a utc time to epoch	1.000
Q33: format date	1.000	Q83: all permutations of a list	1.000
Q34: readonly array	0.000	Q84: extract latitude and longitude from given input	1.000
Q35: filter array	1.000	Q85: how to check if a checkbox is checked	0.000
Q36: map to json	0.500	Q86: converting uint8 array to image	0.125
Q37: parse json file	0.330	Q87: memoize to disk - persistent memoization	0.000
Q38: get current observable value	0.140	Q88: parse command line argument	1.000
Q39: get name of enumerated value	1.000	Q89: how to read contents of a .gz compressed file?	0.000
Q40: encode url	1.000	Q90: sending binary data over a serial connection	1.000
Q41: create cookie	1.000	Q91: extracting data from a text file	1.000
Q42: how to empty array	1.000	Q92: positions of substrings in string	0.000
Q43: how to get current date	1.000	Q93: reading element from html - <td>	0.000
Q44: how to make the checkbox checked	1.000	Q94: deducting the median from each column	1.000
Q45: initializing array	1.000	Q95: concatenate several file remove header lines	0.000
Q46: how to reverse a string	1.000	Q96: parse query string in url	1.000
Q47: read properties file	1.000	Q97: fuzzy match ranking	0.000
Q48: copy to clipboard	1.000	Q98: output to html file	0.000
Q49: convert html to pdf	0.000	Q99: how to read .csv file in an efficient way?	0.200
Q50: json to xml conversion	0.330		

Fig. 5. MRR values of CodeTransformer for the 99 queries of CodeSearchNet dataset

Concerning queries for which our system did not perform as effectively, some of them are relevant to other programming languages and/or are not included in the corpus. Note that these 99 queries are drawn from 6 languages and thus not all of them are relevant to Java. An example unanswered query is Q34, as read-only arrays do not exist in Java and, therefore, a relevant code snippet is not included in the corpus. After manually inspecting the corpus, we concluded that Java code snippets for queries Q53, Q68, Q70, Q73, Q76 and Q97 are focused on other languages. This is also the case for HTML parsing queries, such as queries Q49, Q66, Q72, Q93 and Q98, for which we could find a few Java methods by manual inspection, however they are mainly targeted at other languages. In any case, considering the results of Table 4 and Figure 5, CodeTransformer seems to provide a relevant answer in the first two positions more often than not.

Finally, as a proof of concept, Table 5 depicts the declarations of the methods returned by our system for query Q91, which refers to "extracting data from a text file". It is clear that the methods respond effectively to the query.

Table 5. Declarations of the methods returned by CodeTransformer for query Q91 "extracting data from a text file"

#	Method Declaration
1	public static String readTextFile(Context context, int resId)
2	public static String readTextFile(Context context, String asset)
3	public DataSource<String> readTextFile(String filePath)
4	public static String readTextFile(String fileName)
5	public static String readTextFile(File file)
6	public DataSource<String> readTextFile(String filePath, String charsetName)
7	public static String readTextFile(Context context, int resourceId)
8	public static String readTextFile(File file) throws IOException
9	private ProjectFile readTextFile(InputStream inputStream) throws MPXJException
10	public DataStreamSource<String> readTextFile(String filePath, String charsetName)

4.2 Evaluation using Stack Overflow Questions

To further evaluate CodeTransformer, we reviewed its performance on real user queries. Although our model uses docstrings instead of real queries, we consider this experiment adequate for assessing its effectiveness as a proof of concept.

We manually selected the first 40 highest-rated Stack Overflow posts at the time of research, in which the posters search for Java code snippets. After querying our system using their titles, we obtained 10 results for each query, sorted by their similarity to the query. Next, we manually annotated the similarity of each result to the query, making sure that the result is a valid answer. To avoid any threats to validity, the annotations were performed without knowledge of the order of the results. Table 6 depicts the questions as well as the rank of the first relevant result and the precision at the first 10 results for each question.

Table 6. Evaluation results of CodeTransformer on the set of the 40 most popular Stack Overflow Java questions

#	Questions	Rank	P@10
S01	How do I read / convert an InputStream into a String in Java?	1	1.0
S02	Create ArrayList from array	2	0.7
S03	How do I generate random integers within a specific range in Java?	2	0.7
S04	Iterate through a HashMap [duplicate]	2	0.2
S05	How do I efficiently iterate over each entry in a Java Map?	2	0.1
S06	How do I convert a String to an int in Java?	1	0.2
S07	Initialization of an ArrayList in one line	–	–
S08	How do I determine whether an array contains a value in Java?	3	0.3
S09	How do I call one constructor from another in Java?	–	–
S10	How do I declare and initialize an array in Java?	–	–
S11	How to get an enum value from a string value in Java?	1	1.0
S12	What's the simplest way to print a Java array?	1	1.0
S13	How to generate a random alpha-numeric string?	1	0.8
S14	How to split a string in Java	1	1.0
S15	Sort a Map<Key. Value> by values	7	0.1
S16	How do I create a Java string from the contents of a file?	7	0.1
S17	How can I convert a stack trace to a string?	1	0.8
S18	Fastest way to determine if an integer's square root is integer	–	–
S19	How do I create a file and write to it in Java?	3	0.3
S20	How can I concatenate two arrays in Java?	1	0.7
S21	How to round a number to n decimal places in Java	1	0.8
S22	Convert ArrayList<String> to String[] array	1	0.7
S23	Sort ArrayList of custom Objects by property	1	0.5
S24	How can I initialise a static Map?	–	–
S25	How to directly initialize a HashMap (in a literal way)?	1	0.9
S26	How to create a generic array in Java?	1	1.0
S27	How to parse JSON in Java	1	0.7
S28	Converting array to list in Java	1	0.4
S29	How to get the current working directory in Java?	1	0.9
S30	Converting 'ArrayList<String>' to 'String[]' in Java	9	0.1
S31	How can I pad an integer with zeros on the left?	1	0.4
S32	How can I get the current stack trace in Java?	1	1.0
S33	Java 8 List<V> into Map<K. V>	–	–
S34	Reading a plain text file in Java	1	1.0
S35	How to check if a String is numeric in Java	3	0.7
S36	Java string to date conversion	1	0.9
S37	A 'for' loop to iterate over an enum in Java	–	–
S38	How do I convert a String to an InputStream in Java?	–	–
S39	Convert InputStream to byte array in Java	2	0.7
S40	How can I read a large text file line by line using Java?	1	0.7

Precision at the first 10 results is relatively high for most queries. Moreover, we may note that CodeTransformer effectively disambiguates among queries with similar context. Consider, e.g., queries S17 and S32 that are both relevant to

stack traces; although these queries are similar, the system was able to comprehend the semantics of each query and return several highly ranked relevant results. Even for queries with low precision in their results, CodeTransformer placed the first relevant result in the first or the second position. Thus, even though for some queries there are not many relevant results, the users typically receive at least one correct answer. An example would be query S06, for which the system returned only two relevant results, but one of them is ranked in the first place. It is also notable, in the same query, that CodeTransformer distinguishes among converting "string to integer" and "integer to string".

The fact that 8 out of 40 questions were not answered at all occurs mostly because a matching function does not exist in the corpus. For example, queries S07, S09, S10, S24, and S37 do not require a whole method for their implementation and, thus, the corpus does not include relevant code snippets. Other queries may be too complex, such as query S18, for which our system returns some relevant code snippets, however these results do not meet the condition of the fastest way to examine if an integer's square root is an integer.

In Table 7 we provide three example Stack Overflow queries and the corresponding relevant answers. For the first two queries, CodeTransformer has placed the answers at the first position, while for the third query the answer was placed at the second position. As shown by these examples, CodeTransformer indeed retrieves and recommends useful snippets in a question-answering scenario.

5 Conclusion

Although there are several approaches for code snippet retrieval, most of them do not consider semantics of natural language and code, ignoring essential information regarding the data. Furthermore, several of them recommend API calls or sequences instead of reusable code snippets, requiring more effort from the developer. Deep learning systems are usually more effective, however most do not employ advanced neural transformer architectures and are limited by the fact that they are not trained on annotated datasets. Our system, CodeTransformer, overcomes these limitations by employing a state-of-the-art neural network architecture. The advanced attention mechanisms of this architecture, including specialized similarity metric and custom loss function, along with the preprocessing pipeline specifically designed to augment natural language and code semantics, allow the system to generate powerful data representations.

Upon evaluating CodeTransformer against the implementations of CodeSearchNet, we found that our system is more effective, especially when the developer would prefer to receive the method most relevant to the query rather than a list of related methods. We further assessed CodeTransformer on a dataset of actual questions from Stack Overflow, with the results indicating that it is capable of retrieving useful code, even for complex natural language queries.

For future work, we consider implementing our network using real-life natural language data, such as Stack Overflow questions, instead of code documentation. In addition, we could train our network using other (less curated) datasets

Table 7. Example Stack Overflow queries and the answers of CodeTransformer

Features	Data
Query S11	How to get an enum value from a string value in Java?

Result	`public static T getEnumFromString(Class c, String string) {`
	`  if (c != null && string != null) {`
	`  return Enum.valueOf(c, string.trim().toUpperCase());`
	`  }`
	`  return null;`
	`}`

Query S20	How can I concatenate two arrays in Java?

Result	`public static String[] concat(String[] array1, String[] array2) {`
	`  int length1 = array1.length;`
	`  int length2 = array2.length;`
	`  int length = length1 + length2;`
	`  String[] dest = new String[length];`
	`  System.arraycopy(array1, 0, dest, 0, length1);`
	`  System.arraycopy(array2, 0, dest, length1, length2);`
	`  return dest;`
	`}`

Query S36	Java string to date conversion

Result	`public static Date serviceStringToDate(String s) {`
	`  if (s == null)`
	`  return null;`
	`  try {`
	`  return new SimpleDateFormat(_serviceDateFormat).parse(s);`
	`  }`
	`  catch (Exception e) {`
	`  return null;`
	`  }`
	`}`

and explore different preprocessing techniques, incorporating the semantics of programming symbols and the information provided by method names to the natural language data. Finally, we could explore whether our system can generate docstrings by providing code snippets as input to the code encoder and comparing their sequence embeddings to docstring sequence embeddings.

Acknowledgements

This research has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE (project code: T1EDK-02347).

References

1. Allamanis, M.: The Adverse Effects of Code Duplication in Machine Learning Models of Code. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. p. 143–153. Onward! 2019, Association for Computing Machinery, New York, NY, USA (2019)
2. Bernhardsson, E.: Annoy: Approximate Nearest Neighbors in C++/Python (2018), https://pypi.org/project/annoy/, Python package version 1.13.0
3. Bojanowski, P., Grave, E., Joulin, A., Mikolov, T.: Enriching Word Vectors with Subword Information. Transactions of the Association for Computational Linguistics **5**, 135–146 (2017)
4. Cai, L., Wang, H., Huang, Q., Xia, X., Xing, Z., Lo, D.: BIKER: A Tool for Bi-Information Source Based API Method Recommendation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1075–1079. ESEC/FSE 2019, ACM, New York, NY, USA (2019)
5. Campbell, B.A., Treude, C.: NLP2Code: Code Snippet Content Assist via Natural Language Tasks. In: Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution. pp. 628–632. ICSME 2017, IEEE Computer Society, Los Alamitos, CA, USA (2017)
6. Chen, C., Peng, X., Sun, J., Xing, Z., Wang, X., Zhao, Y., Zhang, H., Zhao, W.: Generative API Usage Code Recommendation with Parameter Concretization. Science China Information Sciences **62**(9), 192103 (2019)
7. Craswell, N.: Mean Reciprocal Rank, p. 1703. In: Liu, Ling and Özsu, M. Tamer (eds), Encyclopedia of Database Systems, Springer, Boston, MA (2009)
8. Diamantopoulos, T., Oikonomou, N., Symeonidis, A.: Extracting Semantics from Question-Answering Services for Snippet Reuse. In: Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering. pp. 119–139. Dublin, Ireland (2020)
9. Gu, X., Zhang, H., Kim, S.: Deep Code Search. In: Proceedings of the 40th International Conference on Software Engineering. p. 933–944. ICSE '18, Association for Computing Machinery, New York, NY, USA (2018)
10. Gu, X., Zhang, H., Zhang, D., Kim, S.: Deep API Learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 631–642. FSE 2016, ACM, New York, NY, USA (2016)
11. Heidarian, A., Dinneen, M.J.: A Hybrid Geometric Approach for Measuring Similarity Level Among Documents and Document Clustering. In: Proceedings of the 2016 IEEE Second International Conference on Big Data Computing Service and Applications. pp. 142–151. BigDataService 2016, IEEE Computer Society, Los Alamitos, CA, USA (2016)
12. Husain, H., Wu, H.H., Gazit, T., Allamanis, M., Brockschmidt, M.: CodeSearchNet Challenge: Evaluating the State of Semantic Code Search (2019)
13. Järvelin, K., Kekäläinen, J.: Cumulated Gain-Based Evaluation of IR Techniques. ACM Trans. Inf. Syst. **20**(4), 422—-446 (2002)
14. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. In: Proceedings of the 3rd International Conference on Learning Representations. pp. 1–15. ICLR 2015, San Diego, CA, USA (2015)
15. Li, X., Jiang, H., Kamei, Y., Chen, X.: Bridging Semantic Gaps between Natural Languages and APIs with Word Embedding. IEEE Transactions on Software Engineering pp. 1–17 (2018)

16. Lopes, C.V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., Vitek, J.: DéJàVu: A Map of Code Duplicates on GitHub. Proc. ACM Program. Lang. **1**(OOPSLA) (2017)
17. Nguyen, A.T., Nguyen, T.N.: Graph-Based Statistical Language Model for Code. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. p. 858–868. ICSE '15, IEEE Press (2015)
18. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T., Di Penta, M.: FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In: Proceedings of the 41st International Conference on Software Engineering. p. 1050–1060. ICSE '19, IEEE Press (2019)
19. Nguyen, T., Rigby, P.C., Nguyen, A.T., Karanfil, M., Nguyen, T.N.: T2API: Synthesizing API Code Usage Templates from English Texts with Statistical Translation. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 1013–1017. FSE 2016, ACM, New York, NY, USA (2016)
20. Ponzanelli, L., Bacchelli, A., Lanza, M.: Seahawk: Stack Overflow in the IDE. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 1295–1298. ICSE '13, IEEE Press, Piscataway, NJ, USA (2013)
21. Raghothaman, M., Wei, Y., Hamadi, Y.: SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In: Proceedings of the 38th International Conference on Software Engineering. pp. 357–367. ICSE '16, ACM, New York, NY, USA (2016)
22. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, u., Polosukhin, I.: Attention is All You Need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. p. 6000–6010. NIPS'17, Curran Associates Inc., Red Hook, NY, USA (2017)
23. Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., Wierstra, D.: Matching Networks for One Shot Learning. In: Proceedings of the 30th International Conference on Neural Information Processing Systems. p. 3637–3645. NIPS'16, Curran Associates Inc., Red Hook, NY, USA (2016)
24. Xu, C., Sun, X., Li, B., Lu, X., Guo, H.: MULAPI: Improving API method recommendation with API usage location. Journal of Systems and Software **142**, 195 – 205 (2018)

AequeVox: Automated Fairness Testing of Speech Recognition Systems*

Sai Sathiesh Rajan(✉), Sakshi Udeshi, and Sudipta Chattopadhyay

Singapore University of Technology and Design, Singapore 487372, Singapore
{*sai_rajan*,*sakshi_udeshi*}@mymail.sutd.edu.sg
sudipta_chattopadhyay@sutd.edu.sg

Abstract. Automatic Speech Recognition (ASR) systems have become ubiquitous. They can be found in a variety of form factors and are increasingly important in our daily lives. As such, ensuring that these systems are equitable to different subgroups of the population is crucial. In this paper, we introduce, AequeVox, an automated testing framework for evaluating the fairness of ASR systems. AequeVox simulates different environments to assess the effectiveness of ASR systems for different populations. In addition, we investigate whether the chosen simulations are comprehensible to humans. We further propose a fault localization technique capable of identifying words that are not robust to these varying environments. Both components of AequeVox are able to operate in the absence of ground truth data.

We evaluate AequeVox on speech from four different datasets using three different commercial ASRs. Our experiments reveal that non-native English, female and Nigerian English speakers generate *109%*, *528.5%* and *156.9%* more errors, on average than native English, male and UK Midlands speakers, respectively. Our user study also reveals that 82.9% of the simulations (employed through speech transformations) had a comprehensibility rating above seven (out of ten), with the lowest rating being 6.78. This further validates the fairness violations discovered by AequeVox. Finally, we show that the non-robust words, as predicted by the fault localization technique embodied in AequeVox, show *223.8%* more errors than the predicted robust words across all ASRs.

1 Introduction

Automated speech recognition (ASR) systems have made great strides in a variety of application areas e.g. smart home devices, robotics and handheld devices, among others. The wide variety of applications have made ASR systems serve increasingly diverse groups of people. Consequently, it is crucial that such systems behave in a non-discriminatory fashion. This is particularly important because assistive technologies powered by ASR systems are often the primary mode of

* This work is partially supported by Singapore Ministry of Education (MOE) grant number MOE2018-T2-1-098 and OneConnect Financial grant number RGOCFT2001.

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 245–267, 2022.
https://doi.org/10.1007/978-3-030-99429-7_14

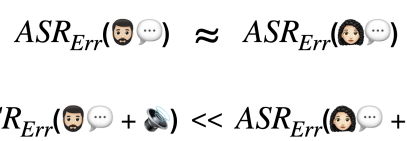


Fig. 1: Fairness Testing in AequeVox

interaction for users with certain disabilities [20]. Consequently, it is critical that an ASR system employed in such systems is effective in diverse environments and across a wide variety of speakers (e.g. male, female, native English speakers, non-native English speakers) since they are often deployed in safety-critical scenarios [18].

In this paper, we are broadly concerned with the *fairness properties in ASR systems*. Specifically, *we investigate whether speech from one group is more robustly recognised as compared to another group*. For instance, consider the example shown in Figure 1 for a system *ASR*. The metric ASR_{Err} captures the error rate induced by *ASR*. Consider speech from two groups of speakers i.e. *male* and *female*. We assume that the *ASR* has similar error rates for both the groups of speakers, as illustrated in the upper half of Figure 1. We now apply a small, constant perturbation on the speech provided by the two groups. Such a perturbation can be, for instance, addition of small noise, exemplifying the natural conditions that the ASR systems may need to work in (e.g. a noisy environment). If we observe that the ASR_{Err} increases disproportionately for one of the speaker groups, as compared to the other, then we consider such a behaviour a violation of fairness (see the second half of Figure 1). Intuitively, Figure 1 exemplifies the violations of *Equality of Outcomes* [36] in the context of ASR systems, where the male group is provided with a higher quality of service in a noisy environment as compared to the female group. Automatically discovering such scenarios of unfairness via simulating the ASR service in diverse environments is the main contribution of our AequeVox framework.

AequeVox facilitates fairness testing without having any access to ground truth transcription data. Although, text-to-speech (TTS) can be used for generating speech, we argue that it is not suitable for accurately identifying the bias towards speech coming from a certain group. Specifically, speakers may intentionally use enunciation, intonation, different degrees of loudness or other aspects of vocalization to articulate their message. Additionally, speakers unintentionally communicate their social characteristics such as their place of origin (through their accent), gender, age and education. This is unique to human speech and TTS systems cannot faithfully capture all the complexities inherent to human speech. Therefore, we believe that fairness testing of ASR systems should involve speech data from human speakers.

We note that human speech (and the ASRs) may be subject to adverse environments (e.g. noise) and it is critical that the fairness evaluation considers such adverse environments. To facilitate the testing of ASR systems in adverse environments, we model the speech signal as a sinusoidal wave and subject it to eight different metamorphic transformations (e.g. noise, drop, low/high pass filter) that are highly relevant in real life. Furthermore, in the absence of man-

ually transcribed speech, we use a differential testing methodology to expose fairness violations. In particular, AequeVox identifies the bias in ASR systems via a two step approach: Firstly, AequeVox registers the increase in error rates for speech from two groups when subjected to a metamorphic transformation. Subsequently, if the increase in the error rate of one group exceeds the other by a given threshold, AequeVox classifies this as a violation of fairness. To the best of our knowledge, we are unaware of any such differential testing methodology. As a by product of our AequeVox framework, we highlight words that contribute to errors by comparing the word counts from the original speech. This information can be further used to improve the ASR system.

Existing works [17,49] isolate certain sensitive attributes (e.g. gender) and use such attributes to test for fairness. Isolating these attributes is difficult in speech data, making it challenging to apply existing techniques to evaluate the fairness of ASR systems. AequeVox tackles this by formalizing a unique fairness criteria targeted at ASR systems. Despite some existing efforts in testing ASR systems [5,13], these are not directly applicable for fairness testing. Additionally, some of these works require manually labelled speech transcription data [13]. Finally, differential testing via TTS [5] is not appropriate to determine the bias towards certain speakers, as they might use different vocalization that might be impossible (and perhaps irrational) to generate via a TTS. In contrast, AequeVox works on speech signals directly and defines transformations directly on these signals. AequeVox also does not require any access to manually labelled speech data for discovering fairness violations. In summary, we make the following contributions in the paper:

1. We formalize a notion of fairness for ASR systems. This formalization draws parallels between the Equality of Outcomes [36] and the quality of service provided by ASR systems in varying environments.
2. We present AequeVox, which systematically combines metamorphic transformations and differential testing to highlight whether speech from a certain group (e.g. female) is subject to fairness violations by ASR systems. AequeVox neither requires access to ground truth transcription data nor does it require access to the ASR model structures.
3. We propose a fault localization method to identify the different words contributing to fairness errors.
4. We evaluate AequeVox with three different ASR systems namely Google Cloud, Microsoft Azure and IBM Watson. We use speech from the Speech Accent Archive [54], the Ryerson Audio-Visual Database of Emotional Speech and Song (RAVDESS) [30], Multi speaker Corpora of the English Accents in the British Isles (Midlands) [11], and a Nigerian English speech dataset [2]. Our evaluation reveals that speech from non-native English speakers and female speakers exhibit higher fairness violations as compared to native English speakers and male speakers, respectively.
5. We validate the fault localization of AequeVox by showing that the identified *faulty* words generally introduce more errors to ASR systems even when used within speech generated via TTS systems. The inputs to the TTS system are randomly generated sentences that conform to a valid grammar.

Table 1: Notations used

Notation	Description
GR_B	Base group
GR_k	$k \in (1, n)$. Various comparison groups
MT	Metamorphic transformations
ASR	Automatic Speech Recognition system under test
τ	A user specified threshold beyond which the difference in word error rate for the base and comparison groups is considered a violation of individual fairness

6. We evaluate (via the user study) the human comprehensibility score of the transformations employed by AequeVox on the speech signal. The lowest comprehensibility score was 6.78 and 82.9% of the transformations had a comprehensibility score of more than seven.

2 Background

In this section, we introduce the necessary background information.

Fairness in ASR Systems: A recent work, FairSpeech [26], uses conversational speech from black and white speakers to find that the word error rate for individuals who speak African American Vernacular English (AAVE) is nearly twice as large in all cases.

Testing ASR Systems: The major testing focus, till date has been on image recognition systems and large language models. Few papers have probed ASR systems. One such work, Deep-Cruiser [13] applies metamorphic transformations to audio samples to perform coverage-guided testing on ASR systems. Iwama et al. [23] also perform automated testing on the basic recognition capabilities of ASR systems to detect functional defects. CrossASR [5] is another recent paper that applies differential testing to ASR systems.

The Gap in Testing ASR Systems: There is little work on automated methods to formalise and test fairness in ASR systems. In this work, we present AequeVox to test the fairness of ASR systems with respect to different population groups. It accomplishes this with the aid of differential testing of speech samples that have gone through metamorphic transformations of varying intensity. Our experimentation suggests that speech from different groups of speakers receives significantly different quality of service across ASR systems. In the subsequent sections, we describe the design and evaluation of our AequeVox system.

3 Methodology

In this section, we discuss AequeVox in detail. In particular, we motivate and formalize the notion of *fairness* in ASR systems. Then, we discuss our methodology to systematically find the violation of fairness in ASR systems. The notations used are described in Table 1.

Motivation: Equality of outcomes [36] describes a state in which all people have approximately the same material wealth and income, or in which the general

economic conditions of everyone's lives are alike. For a software system, equality of outcomes can be thought of as everyone getting the same quality of service from the software they are using. For a lot of software services, providing the same quality of service is baked into the system by design. For example, the results of a search engine only depend on the query. The quality of the result generally does not depend on any sensitive attributes such as race, age, gender and nationality. In the context of an ASR, the quality of service does depend on these sensitive attributes. This inferior quality of service may be especially detrimental in safety-critical settings such as emergency medicine [18] or air traffic management [27,21].

In our work, we show that the quality of service provided by ASR systems is vastly different depending on one's gender/nationality/accent. Suppose there are two groups of people using an ASR system, males and females. They have approximately the same level of service when using this service at their homes. However, once they step into a different environment such as a noisy street, the quality of service drops notably for the female users, but does not drop noticeably for the male users. This is a violation of the principle of equality of outcomes (as seen for software systems) and more specifically, group fairness [14]. Such a scenario is unfair (violation of group fairness) because some groups enjoy a higher quality of service than others.

In our work, we aim to automate the discovery of this unfairness. We do this by simulating the environment where the behaviour of ASR systems are likely to vary. The simulated environment is then enforced in speech from different groups. Finally, we measure how different groups are served in different environments.

Formalising Fairness in ASRs: In this section, we formalise the notion of fairness in the context of automated speech recognition systems (ASRs). The fairness definition in ASRs is as follows:

$$|ASR_{Err}(GR_i) - ASR_{Err}(GR_j)| \leq \tau \tag{1}$$

Here, GR_i and GR_j capture speech from distinct groups of people. If the error rates induced by ASR for group GR_i ($ASR_{Err}(GR_i)$) and for group GR_j ($ASR_{Err}(GR_j)$) differ beyond a certain threshold, we consider this scenario to be unfair. Such a notion of unfairness was studied in a recent work [26].

In this work, we want to explore whether different groups are fairly treated under varying conditions. Intuitively, we subject speech from different groups to a variety of simulated environments. We then measure the word error rates of the speech in such simulated environments and check if certain groups fare better than others. Formally, we capture the notion of fairness targeted by AequeVox as follows:

$$\begin{aligned} D_i &\leftarrow ASR_{Err}(GR_i) - ASR_{Err}(GR_i + \delta) \\ D_j &\leftarrow ASR_{Err}(GR_j) - ASR_{Err}(GR_j + \delta) \\ |D_i - D_j| &\leq \tau \end{aligned} \tag{2}$$

Here we perturb the speech of the two groups (GR_i and GR_j) by adding some δ to the speech. We compare the degradation in the speech (D_i and D_j). If the

Algorithm 1 AequeVox Fairness Testing

procedure Fairness_Testing($GR_B, MT, GR_1, \cdots, GR_n, \tau, ASR_1, ASR_2$)
  $Error\_Set \leftarrow \varnothing$
  for $T \in MT$ **do**
    $GR_B^T \leftarrow T(GR_B)$
    ▷ $\mathcal{L}$ computes the average word level levenshtein distance
    ▷ between the outputs of ASR_1 and ASR_2
    $d_B \leftarrow \mathcal{L}(ASR_1(GR_B), ASR_2(GR_B))$
    $d_B^T \leftarrow \mathcal{L}(ASR_1(GR_B^T), ASR_2(GR_B^T))$
    $D_B \leftarrow d_B^T - d_B$
    for $k \in (1, n)$ **do**
      $GR_k^T \leftarrow T(GR_k)$
      $d_k \leftarrow \mathcal{L}(ASR_1(GR_k), ASR_2(GR_k))$
      $d_k^T \leftarrow \mathcal{L}(ASR_1(GR_k^T), ASR_2(GR_k^T))$
      $D_k \leftarrow d_k^T - d_k$
      if $D_B - D_k > \tau$ **then**
        $Error\_Set \leftarrow Error\_Set \cup (GR_B, GR_k, T)$
      end if
    end for
  end for
  return $Error\_Set$
end procedure

degradation faced by one group is far greater than the one faced by the other, we have a fairness violation. This is because speech from both groups ought to face similar degradation when subject to similar environments (simulated by δ perturbation) when equality of outcomes [36] holds. More specifically, this is a group fairness violation because the quality of service (outcome) depends on the group [14,51].

Example: To motivate our system, let us sketch out an example. Consider texts of approximately the same length spoken by two sets of speakers whose native languages are L_1 and L_2 respectively. Let us assume that both sets of speakers read out a text in English. AequeVox uses two ASR systems and obtains the transcript of this speech. AequeVox then employs differential testing to find the word-level levenshtein distance [29] between these two sets of transcripts. Let us also assume that the average word-level levenshtein distance is two and four for L_1 and L_2 native speakers, respectively.

AequeVox then simulates a noisy environment by adding noise to the speech and obtains the transcript of this transformed speech. Let us assume now that the average levenshtein distance for this transformed speech is 4 and 25 for L_1 and L_2 native speakers, respectively. It is clear that the degradation for the speech of native L_2 speakers is much more severe. In this case, the quality of service that L_2 native speakers receive in noisy environments is worse than L_1 native speakers. This is a violation of fairness which AequeVox aims to detect.

The working principle behind AequeVox holds even if the spoken text is different. This is because AequeVox just measures the relative degradation in ASR performance for a set of speakers. For large datasets, we are able to measure the average degradation in ASR performance with respect to different groups of speakers (e.g. *male*, *female*, *native, non-native English* speakers).

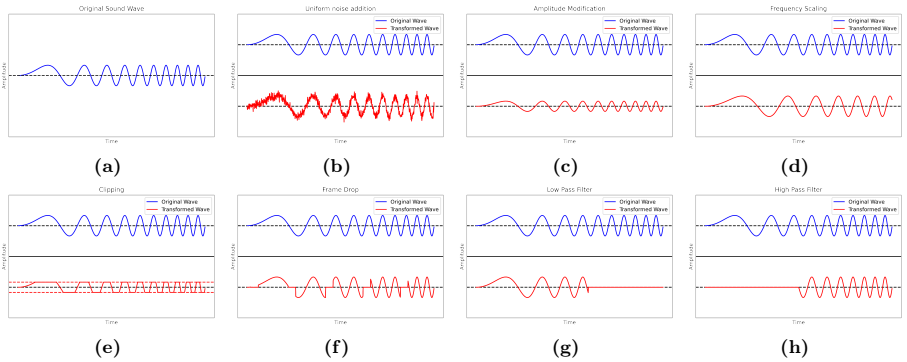


Fig. 2: Sound wave transformations

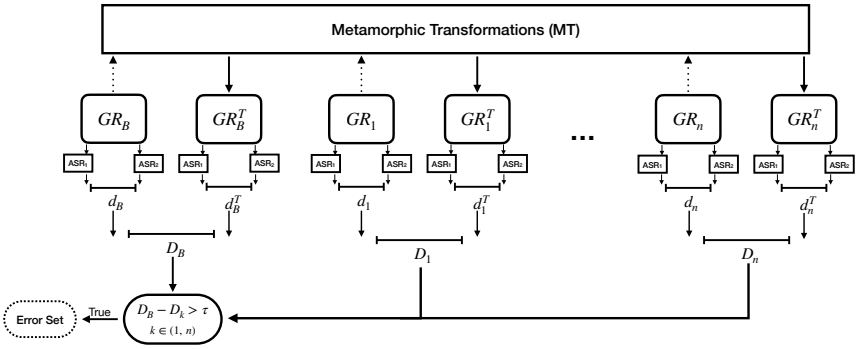


Fig. 3: AequeVox System Overview

Metamorphic Transformations of Sound: The ability to operate in a wide range of environments is crucial in ASR systems as they are deployed in safety-critical settings such as medical emergency services [18] and air traffic managment [21], [27], which are known to have interference and noise. Metamorphic speech transformations serve to simulate such scenarios. The key insight for our metamorphic transformations comes from how waves are represented and what can happen to these waves when they're transmitted in different mediums. We realise this insight in the fairness testing system for ASR systems. To the best of our knowledge AequeVox is the first work that combines this insight from acoustics, software testing and software fairness to evaluate the fairness of ASR systems. AequeVox uses the addition of noise (Figure 2 (b)), amplitude modification (Figure 2 (c)), frequency modification (Figure 2 (d)), amplitude clipping (Figure 2 (e)), frame drops (Figure 2 (f)), low-pass filters (Figure 2 (g)), and high-pass filters (Figure 2 (h)) as metamorphic speech transformations. We choose these transformations because they are the most common distortions for sound in various environments [1].

System Overview: Algorithm 1 provides an outline of our overall test generation process. We realise the notion of fairness described in Equation (2) using differential testing. The error rates (ASR_{Err}) for a particular speech clip are

found by finding the difference between the outputs of two ASR systems, ASR_1 and ASR_2. It is important to note that we make a design choice to use *differential testing* to find the error rate (ASR_{Err}). This helps us eliminate the need for ground truth transcription data which is both labor intensive and expensive to obtain. Furthermore, AequeVox realises the δ seen in Equation (2) by using metamorphic transformations for speech (*see Figure 3*). These speech metamorphic transformations represent the various simulated environments for which AequeVox wants to measure the quality of service for different groups. Additionally, the user can customise this δ per their requirements. In our implementation we use eight distinct metamorphic transformations as δ (*see Figure 2*). Specifically, we investigate how fairly do two ASR systems (ASR_1 and ASR_2) treat groups ($GR_k^T \mid k \in \{1, 2, \cdots n\}$) with respect to a base group (GR_B). AequeVox achieves this by taking a dataset of speech which contains data from two or more different groups (e.g. male and female speakers, Native English and Non-native English speakers) and modifies these speech snippets through a set of transformations (MT). These are then divided into base group transformed speech (GR_B^T) and the transformed speech for other groups ($GR_k^T \mid k \in \{1, 2, \cdots n\}$). As seen in Algorithm 1, the average word-level levenshtein distance (word-level levenshtein distance divided by the number of words in the longer transcript) between the outputs of the two ASR systems is captured by d_B and d_B^T for the original and transformed speech respectively. Similarly, for the comparison groups $GR_k^T (k \in \{1, 2, \cdots n\})$ the word-level levenshtein distance is captured by d_k and d_k^T. The higher the levenshtein distance the larger the error in terms of differential testing. In other words, larger error in differential testing would mean that the ASR systems disagree on a higher number of words.

To capture the degradation in the quality of service for the speech subjected to simulated environments (MT), we compute the difference between the word-level levenshtein distance for the original and transformed speech. Specifically, we compute D_B as $d_B^T - d_B$ and D_k as $d_k^T - d_k (k \in \{1, 2, \cdots n\})$ for the base and comparison groups, respectively. The higher this metric (D_B and D_k), the more severe the degradation in ASR quality of service because of the transformation T.

We compare these metrics and if D_B exceeds D_k by some threshold τ, we classify this as an error for the *base group* (GR_B) and more specifically a violation of fairness (*see Figure 3*). In our experiments we set each of the groups in our dataset as the base group (GR_B) and run the AequeVox technique to find errors with respect to that base group. The lower the errors (as computed via the violation of the assertion $D_B - D_k \leq \tau$), the fairer the ASR systems are with respect to groups GR_B. As an example, let us say Russian speakers are the base group (GR_B), English speakers are the comparison group (GR_k) and the value of τ is 0.1. If D_B is strictly greater than D_k by 0.1, then fairness violation is counted for the Russian speakers. Otherwise, no fairness errors are recorded.

Fault Localisation: AequeVox introduces a word-level fault localisation technique, which does not require any access to ground truth data. We first illustrate a use case of this fault localisation technique.

Algorithm 2 AequeVox Fault Localizer

```
1: procedure Fault_Localizer(WC, WC^{T_θ}, ω, param^T)
2:     Drop_Count ← ∅
3:     Non_Robust_Words ← ∅
4:     for word ∈ WC.keys() do
5:         init_count ← WC[word]
6:         ▷ Returns the minimum count of word across all the parameter
7:         ▷ of transformation T
8:         min_count ← get_min(WC^{T_θ}[word], param^T)
9:         count_diff ← max ((init_count − min_count), 0)
10:        if count_diff > ω then
11:            Non_Robust_Words ← Non_Robust_Words ∪ {word}
12:        end if
13:        Drop_Count ← Drop_Count ∪ {count_diff}
14:    end for
15:    return Non_Robust_Words, Drop_Count
16: end procedure
```

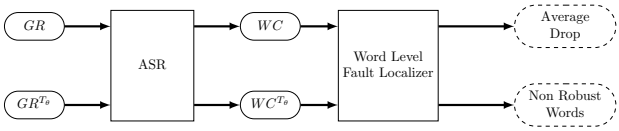


Fig. 4: AequeVox Fault Localization Overview

Example: Let us consider a corpus of English sentences by a group of speakers (say GR) who speak language L_1 natively. AequeVox builds a dictionary for all the words in the transcript obtained from ASR_1. An excerpt from such a dictionary appears as follows: $\{brother : 16, nice : 25, is : 33, \cdots\}$. This means the words *brother, nice* and *is* were seen 16, 25 and 33 times in the transcript respectively. Now, assume AequeVox simulates a noisy environment by adding noise with various signal to noise (SNR) ratios as follows: $\{10, 8, 6, 4, 2\}$. This is the parameter for the transformation ($param^T$).

Once AequeVox obtains the transcript of these transformed inputs, it creates dictionaries similar to the ones seen in the preceding paragraph. Let the relevant subset of the dictionary for SNR *two* (2) be $\{brother : 1, nice : 23, is : 32, \cdots\}$. We use this to determine that the utterance of the word *brother* is not robust for noise addition for the group GR. This is because, the word brother appears significantly less in the transcript for the modified speech, as compared to the transcript for the original speech.

AequeVox fault localisation overview: Algorithm 2 provides an overview of the fault localization technique implemented in AequeVox. The goal of the AequeVox fault localisation is to find words for a group (GR) that are not robust to the simulated environments. Specifically, AequeVox finds words which are not recognised by the ASR when subjected to the appropriate speech transformations.

The transformation is represented by T_θ. Here, $T \in MT$ is the transformation and $\theta \in param^T$ is the parameter of the transformation, which controls the severity of the transformation.

As seen in Algorithm 2, AequeVox builds a word count dictionary for each word in WC and WC^{T_θ} for the original speech and for each $\theta \in param^T$ re-

spectively. For each word, AequeVox finds the difference in the number of appearances for a word in WC and in WC^{T_θ} for $\theta \in param^T$. To compute the difference, we locate the minimum number of appearances across all the transformation parameters $\theta \in param^T$ (i.e. min_count in Algorithm 2). This is to locate the worst-case degradation across all transformation parameters. The difference is then calculated between min_count and the number of appearances of the word in the original speech (i.e. $init\_count$). If the difference exceeds some user-defined threshold ω, then AequeVox classifies the respective words as non robust w.r.t the group GR and transformation T.

We envision that practitioners can then review the data generated by fault localization (i.e. Algorithm 2) and target the non-robust words to further improve their ASR systems for speech from underrepresented groups [24] and accommodate for speech variability [22]. In **RQ3**, we validate our fault localization method empirically and in **RQ4**, we show how the proposed fault localization method can be used to highlight fairness violations.

4 Datasets and Experimental Setup

ASR Systems under Test: We evaluate AequeVox on three commercial ASR systems from Google Cloud Platform (GCP), IBM Cloud, and Microsoft Azure. We use the standard models for GCP and Azure, and the *BroadbandModel* for IBM. In all three cases, the audio samples were identically encoded as .wav files using Linear 16 encoding.

In each of the following transformations, we vary a parameter, θ. We call this the transformation parameter. Some of the transformations have abbreviations within parentheses. Such abbreviations are used in later sections to refer to the respective transformations.

Amplitude Scaling (Amp): For amplitude scaling, we scale the audio sequence by a constant by multiplying each individual audio sample by θ.

Clipping: The audio samples are scaled such that their amplitude values are bound by $[-1, 1]$. AequeVox then clips these samples such that the amplitude range is $[-\theta, \theta]$. These clipped samples are then rescaled and encoded.

Drop/Frame: For Drop, AequeVox divides the audio into 20ms chunks. θ% of these chunks are then randomly discarded (amplitude set to zero) from the audio. For Frame, AequeVox divides the audio into θms chunks and 10% of these chunks are then randomly discarded. No two adjacent chunks are discarded.

High Pass (HP)/ Low Pass (LP) Filter: Here we apply a butterworth [7] filter of order two to the entire audio file with θ determining the cut-off frequency.

Noise Addition (Noise): θ represents signal to noise (SNR) ratio [25] of the transformed audio signal. A lower θ means higher noise in the transformed audio.

Frequency Scaling (Scale): In this case, θ is the sampling frequency. The lower the value of θ, the slower the audio. In this transformation, the audio is slowed down θ times.

Table 2: Transformations Used

Transformation Type	θ Used				
	Least Destructive ⟶ Most Destructive				
Amplitude	0.5	0.4	0.3	0.2	0.1
Clipping	0.05	0.04	0.03	0.02	0.01
Drop	5	10	15	20	25
Frame	10	20	30	40	50
HP	500	600	700	800	900
LP	900	800	700	600	500
Noise	10	8	6	4	2
Scale	0.9	0.8	0.7	0.6	0.5

Table 3: Datasets Used

Dataset	Duration(s)	#Clips	#Distinct Speakers
Accents	25-35	28	28
RAVDESS	3	32	8
Midlands	3-5	4	4
Nigerian English	4-6	4	4

Table 2 lists all the different values used for θ. An additional parameter ($\theta = 2.0$) is used for *Amp*.

Datasets: We use the Speech Accent Archive (Accents) [54], the Ryerson Audio-Visual Database of Emotional Speech and Song (RAVDESS) [30], Multi speaker Corpora of the English Accents in the British Isles (Midlands) [11], and a Nigerian English speech dataset [2] to evaluate AequeVox taking care to ensure male and female speakers are equally represented. Table 3 provides additional details about the setup.

5 Results

In this section, we discuss our evaluation of AequeVox in detail. In particular, we structure our evaluation in the form of four research questions (**RQ1** to **RQ4**). The analysis of these research questions appears in the following sections.

RQ1: What is AequeVox's efficacy?

We structure the analysis of this research question into three sections, each corresponding to a dataset we have used in our analysis. All of the relevant data is presented in Table 4 with the lowest errors for each dataset bolded. We first analyse the number of errors (used interchangeably with fairness violations) for each case. Subsequently, we analyse the sensitivity of the errors with respect to the values of τ ($\tau \in \{0.01, 0.05, 0.1, 0.15\}$). Detecting violations of fairness is regulated by parameter τ. Lower values of τ imply that the degradation of word error rates between two groups should be similar, and conversely higher values of τ allow for the difference in degradation of word error rates to be more severe between two groups. Next, we analyse the sensitivity of the pairs of the ASR systems under test. Concretely, we analyse the errors found in the Microsoft Azure and IBM Watson *(MS_IBM)*, Google Cloud and IBM Watson *(IBM_GCP)*, and Microsoft Azure and Google Cloud *(MS_GCP)* pairs. Finally, we analyse the sensitivity of the AequeVox test generation with respect to the eight different types of transformations implemented (see Figure 2).

Table 4: Errors Discovered by AequeVox

	Accents							RAVDESS		Nigerian/Midlands English	
	English	Ganda	French	Gujarati	Indonesian	Korean	Russian	Male	Female	Midlands	Nigerian
τ Sensitivity											
0.01	**168**	381	267	232	178	499	354	**12**	92	**36**	75
0.05	**75**	245	99	101	85	340	227	**8**	53	**26**	65
0.10	43	145	39	49	**34**	172	161	**5**	21	**17**	55
0.15	26	73	**8**	24	14	75	111	**3**	10	**14**	44
ASR Sensitivity											
MS IBM	**36**	369	128	126	64	388	303	**10**	57	**30**	86
GCP IBM	131	325	123	147	**98**	342	361	**9**	64	**31**	96
MS GCP	145	150	162	**133**	149	356	189	**9**	55	**32**	57
Transition Sensitivty											
Clipping	**4**	81	38	159	72	182	237	**0**	24	50	**3**
Drop	**8**	113	33	29	40	184	45	**0**	21	**4**	33
Frame	**14**	106	61	25	36	170	26	**1**	13	**13**	19
Noise	**5**	128	54	86	22	217	213	**0**	24	**5**	43
LP	39	158	108	57	**14**	110	208	**0**	45	**4**	34
Amplitude	81	19	44	33	**14**	40	26	**0**	27	**8**	40
HP	114	168	29	**9**	61	87	57	**9**	20	**1**	51
Scale	47	71	46	**8**	52	96	41	18	**2**	**8**	16
Total Errors	312	844	413	406	**311**	1086	853	**28**	176	**93**	239

It is important to note that we excluded the two most destructive Scale transformations. This is because the word error rate for these transformations is 0.89 on average out of 1. This degradation may be attributed to the transformation itself rather than the ASR. To avoid such cases, we exclude these transformations from this research question.

Accents Dataset: Native English speakers and Indonesian speakers have the lowest number of errors. On average, speech from non-native English speakers generates *109% more errors* in comparison to speech from native English speakers. For the two smallest values of τ, speech from the native English speakers shows the *least number* of fairness violations. Speech from native English speakers has the lowest, second lowest and third lowest errors for the pairs of ASRs, *(MS_ IBM)*, *(MS_ GCP)* and *(IBM_ GCP)* respectively. Speech from native English speakers has the lowest errors for the clipping, two types of frame drops and noise transformations and the second lowest errors for the low-pass filter transformation. The high-pass filter and scaling induce a comparable number of errors from native and a majority of the non-native English speakers. However, speech from native English speakers has the highest number of errors when subject to the amplitude transformation.

Speech from non-native English speakers generally exhibits more fairness violations in comparison to speech from native English speakers.

RAVDESS Dataset: Speech from male speakers has significantly lower errors than speech from female speakers. On average, speech from female speakers generates *528.57% more errors* in comparison to speech from male speakers. Speech from male speakers shows significantly fewer fairness violations for all values of τ, and for all ASR pairs tested. Clipping, both types of frame drops, noise, low-pass, amplitude and high-pass transformations induce significantly

fewer errors on speech from male speakers. However, speech from male speakers has more errors when subject to scale transformations.

Speech from female speakers has significantly more fairness violations in comparison to speech from male speakers.

Midlands/Nigeria Dataset: Speech from UK Midlands English (ME) speakers has significantly fewer errors than speech from Nigerian English (NE) speakers. On average, speech from NE speakers generates *156.9% more errors* in comparison to speech from ME speakers. Speech from ME speakers has significantly fewer fairness errors for all values of τ, and for all ASR pairs tested. For the transformations scale, drop, noise, amplitude, low pass and high pass filters, the speech from ME speakers has significantly fewer errors than speech from NE speakers. Clipping induces more errors in speech from ME speakers, while the frame transformation induces comparable number of errors in speech from both groups.

Speech from Nigerian English speakers has significantly more fairness errors in comparison to speech from UK Midlands speakers.

RQ2: What are the effects of transformations on comprehensibility?

To better understand the effects of the transformations (*see Figure 2*) on the comprehensibility of the speech we conducted a user study. Speech of one randomly chosen *female native English* speaker from the Accents [54] dataset was used since the audio contains nearly all the sounds present in the English language [54]. Survey participants were presented with the original audio file along with a set of transformed speech files in order of increasing intensity. All the transformations (*see Figure 2*) and transformation parameters (*see Table 2*) were used. We asked 200 survey participants (sourced through Amazon mTurk) the following question:

`How comprehensible is (transformed) Speech with respect to the Original speech?`

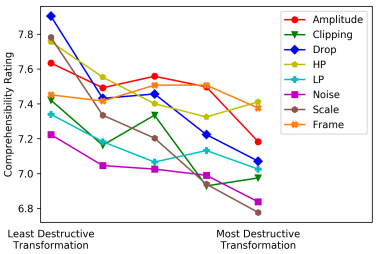


Fig. 5: Average Transformation Comprehensibility Ratings

The rating of one (1) is *Not Comprehensible at all* and the rating of ten (10) is *Just as Comprehensible as the Original.*

Unsurprisingly, as seen in Figure 5, increasing the intensities of the transformation had a generally detrimental effect on the comprehensibility of the speech. But none of the transformations majorly affect the comprehensibility of the speech. All of the transformations had an average comprehensibility rating above 6.75 and 82.9% of the transformations had a comprehensibility rating above 7.

Table 5: Fairness errors where the transformations have a comprehensibility rating of at least 7.2

	Accents							RAVDESS		Nigerian/Midlands English	
	English	Ganda	French	Gujarati	Indonesian	Korean	Russian	Male	Female	Midlands	Nigerian
Total Errors	246	509	240	**166**	225	687	329	**28**	88	**55**	161

Table 6: Grammar-generated sentence examples

ASR	Microsoft	Google Cloud	IBM Watson
Robust	Ashley likes **fresh** smoothies Paul adores **spoons** of cinnamon	Karen loves **plastic** straws Donald hates **big** decisions	William detests **plastic** cups Steven detests **big** flags
Non-robust	Ashley detests **thick** smoothies Ryan likes **slabs** of cake	John loves **spoons** of cinnamon Robert loves **bags** of concrete	Betty likes **scoops** of ice cream Amanda is fond of **things** like groceries

The average degradation in comprehensibility for the least destructive parameter across all transformations was 24.36%. Noise was the most destructive at 27.75% and drop was the least destructive (20.96%).

The average degradation in comprehensibility for the most destructive parameter across all transformations was 29.18%. In this case, scaling was the most destructive at 32.23% whereas drop was the least destructive with 25.88%.

Additionally, for each transformation, we analyse the percentage drop of comprehensibility between the least and the most destructive transformation parameters. The average drop is 4.82% across all transformations. The scaling and drop transformations show high relative percentage drops of 10.05% and 8.32% respectively. Amplitude, clipping, noise, high-pass and low-pass filters show closer to average drops between 3.1% and 4.5%. Frame, on the other hand, shows very low relative drops at 0.76%.

All the transformations, though destructive, are comprehensible by humans.

For safety critical applications, we recommend that future work test the whole gamut of transformations. For other use cases, practitioners may choose the transformations that satisfy their needs. To aid this, AequeVox allows the users to choose the comprehensibility threshold of the transformations. As seen in Table 5, our conclusion holds even if we choose the transformations with higher comprehensibility threshold (7.2). We highlight the group with the least errors in each dataset to aid in readability. In particular, we observe that speech from male and UK Midlands speakers generally exhibit fewer errors. Setting aside speech from native Gujarati speakers, speech from native English speakers exhibits comparable or better performance than speech from other groups.

RQ3: Are the outputs produced by AequeVox fault localiser valid?

To study the validity of the outputs of the fault localiser, we study the number of errors for the predicted robust and non-robust words. We do this by generating speech containing the predicted robust and non-robust words for each ASR tested. We choose an ω of three, three and two for GCP, MS Azure and IBM respectively to choose the non-robust words (*see Algorithm 2*). We choose the robust words from the set of words that do not show any errors

in the presence of noise ($count\_diff = 0$ in *Algorithm 2*) for these specific ASR systems. Specifically, we test whether the robust and non-robust words identified by the fault localiser in the Accents dataset are robust in the presence of noise. Our goal is to show that if noise is added to speech containing these non-robust words, the ASR will be less likely to recognise them. Vice-versa, if noise is added to the predicted robust-words they are less likely to be affected.

To generate the speech from the output we generate sentences containing the robust and non-robust words predicted by the fault localiser for each ASR using a grammar and then use a text-to-speech (TTS) service to generate speech. The actual randomly selected robust and non-robust words (in bold) and the examples of the sentences generated by the grammar can be seen in Table 6. We use the Google TTS for MS Azure and we use the Microsoft Azure TTS for GCP and IBM to generate the speech.

To evaluate the generality of outputs of the fault localisation technique, we use the speech produced by the TTS and then add noise to that speech. This speech is used to generate a transcript from the ASR and the transcript is used to evaluate how many of the predicted robust and non-robust words are incorrect in the transcript. We add the most noise possible to the TTS speech in our AequeVox framework. Specifically, the signal to noise (SNR) ratio is 2. We use the TTS generated speech for 50 sentences for each of the robust and non-robust cases. Each sentence has either a robust or a non-robust word.

The results of the experiments are seen in Table 7. In the transcript of the speech with noise added at SNR 2, robust words show *zero* errors for the predicted robust words for Microsoft and Google Cloud and 21 errors for IBM. The non-robust words, on the other hand, had 23, 15 and 30 errors.

The predicted non-robust words have a higher propensity for errors than the robust words.

Table 7: Transcript Errors

ASR	No. of Erroneous Sentences
Microsoft (MS)	
Robust	0
Non-Robust	23
Google Cloud (GCP)	
Robust	0
Non-Robust	15
IBM Watson (IBM)	
Robust	21
Non-Robust	30

Table 8: Grammarly Scores

ASR	Overall Score	Correctness	Clarity
Microsoft (MS)			
Robust	99		
Non-Robust	99		
Google Cloud (GCP)			
Robust	100	Looking Good	Very Clear
Non-Robust	99		
IBM Watson (IBM)			
Robust	100		
Non-Robust	96		

Note on grammar validity: Since the grammars used by us to validate the explanations of AequeVox are handcrafted, they may be prone to errors. To verify these hand crafted grammars, we use 100 sentences produced by each grammar and use the online tool Grammarly [3] to investigate the semantic and syntactic correctness of the sentences and the clarity. The sentences generated by the grammars have a high overall average score of 98.33 out of 100, with the

Table 9: Average word mispredictions in the Accents dataset using the AequeVox localisation techniques

	Accents						
	English	Ganda	French	Gujarati	Indonesian	Korean	Russian
ASR Sensitivity							
GCP	1.21	1.51	1.21	1.17	**1.07**	1.55	1.64
IBM	**1.03**	1.94	1.38	1.35	1.48	1.92	1.70
MS Azure	0.47	0.66	0.40	0.48	**0.36**	0.87	0.63
Transition Sensitivity							
Clipping	**2.00**	2.53	2.12	2.60	2.29	2.81	3.13
Drop	**0.30**	1.02	0.52	0.54	0.57	1.15	0.74
Frame	**0.38**	0.89	0.68	0.56	0.51	1.19	0.65
Noise	**0.57**	1.60	0.85	1.27	0.71	1.74	1.54
LP	1.72	2.22	1.90	1.79	**1.58**	1.98	2.13
Amplitude	0.17	0.15	0.11	0.12	**0.06**	0.20	0.16
HP	0.74	0.75	0.38	**0.22**	0.49	0.64	0.76
Scale	1.38	1.79	1.42	**0.90**	1.54	1.89	1.45

lowest being 96 (*see Table 8*). On the correctness and clarity measure, all the sentences generated by the grammars score *Looking Good* and *Very Clear*.

RQ4: Can the fault localiser be used to highlight unfairness?

The goal of this RQ is to investigate if the output of Algorithm 2 can call attention to bias between different groups. Specifically, we evaluate if some groups show fewer faults, on average than others. To this end, we use the fault localisation algorithm (*Algorithm 2*) on the accents dataset and record the number of words incorrect in the transcript, on average for each group of the accents dataset. This is done for each ASR under test. It is also important to note that this technique uses no ground truth data and requires no manual input. This technique is designed to work with just the speech data and metadata (groups).

Table 9 shows the average word drops across all transformations for the accents dataset for each ASR under test. We highlight the best performing groups by bolding the values. Speech from native-English speakers shows the lowest average word drops for the IBM Watson ASR and the third lowest for GCP and MS Azure ASRs. We also investigate the average word drops for each transformation in AequeVox averaged across all ASRs. Speech from native English speakers has the lowest average word drops for the Clipping, two types of frame drops and noise transformations and the second lowest errors for the low-pass filter transformation. (*see Table 9*). For the rest of the transformations, namely amplitude, high-pass filter and scaling, we find that both speech from non-native English speakers and speech from native English speakers have comparable average word drops in the majority of cases (*see Table 9*). This result is consistent with results seen in RQ1.

The technique seen in Algorithm 2 can be used to highlight bias in speech and the results are consistent with RQ1.

6 Threats to Validity

User Study: In conducting the study, two assumptions were made. Firstly, we assume that the degree to which comprehensibility changes when subject to transformations is independent of the characteristics of the speaker's voice. Secondly, we assume that the speech is reflective of the broader English language. In future work, a larger scale user study could be performed to verify the results.

ASR Baseline Accuracy: AequeVox measures the degradation of the speech to characterise the unfairness amongst groups and ASR systems. If the baseline error rate is very high, then the room for further degradation is very low. As a result, AequeVox expects ASR services to have a high baseline accuracy. To mitigate this threat, we use state-of-the-art commercial ASR systems which have high baseline accuracies.

Completeness and Speech Data: AequeVox is incomplete, by design, in the discovery of fairness violations. AequeVox is limited by the speech data and the groups of this speech data used to test these ASR systems. With new data and new groups, it is possible to discover more fairness violations. The practitioners need to provide data to discover these. In our view, this is a valid assumption because the developers of these systems have a large (and growing) corpus of such speech data. It is also important to note that AequeVox does not need the ground truth transcripts for this speech data and such speech data is easier to obtain.

Fault Localisation: To test AequeVox's fault localisation, we identify the robust and non-robust words in the speech and subsequently construct sentences (with the aid of a grammar). These sentences are then converted to speech using a text-to-speech (TTS) software and the performance of the robust and non-robust words is measured. In the future, we would like to repeat the same experiment with a fixed set of speakers, which allows us capture the peculiarities of speech in contrast to the usage of TTS software.

7 Related Work

In the past few years, there has been significant attention in testing ML systems [35,48,32,47,55,34,50,40,56,16,52,8,41,19]. Some of these works target coverage-based testing [48,55,34,32] or leverage property driven testing [41], while others focus on effective testing in targeted domains e.g. text [50,40]. None of these works, however, are directly applicable for testing ASR systems. In contrast, the goal of AequeVox is to automatically discover violations of fairness in ASR systems without access to ground truth data.

DeepCruiser [13] uses metamorphic transformations and performs coverage-guided fuzzing to discover transcription errors in ASR systems. Concurrently, CrossASR [5] uses text to generate speech from a TTS engine and subsequently employs differential testing to find bugs in the ASR system. In contrast to these systems, the goal of AequeVox is to automatically find violations of fairness

by measuring the degradation of transcription quality from the ASR when the speech is transformed. AequeVox compares this degradation across various groups of speakers and if the difference is substantial, AequeVox characterises this as a fairness violation. Moreover, AequeVox neither requires access to manually labelled speech data nor does it require any white/grey box access to the ASR model. Works on audio adversarial testing [23], [10], [9], [37], [28] aims to find an imperceptible perturbation that are specially crafted for an audio file. In contrast, AequeVox aims to find fairness violations. Additionally, AequeVox also proposes automatic fault localisation for ASR systems without using a ground truth transcript.

Unlike AequeVox, recent works on fairness testing have focused on credit rating [17,49,4,57,42,44,43,41], computer vision [12,6] or NLP systems [33,45]. In the systems that deal with such data, it is possible to isolate certain sensitive attributes (gender, age, nationality) and test for fairness based on these attributes. It is challenging to isolate such sensitive attributes in speech data, necessitating the need for a separate fairness testing framework specifically for speech data.

Frameworks such as LIME [38], SHAP [31], Anchor [39] and DeepCover [46] attempt to reason why a model generates a specific output for a specific input. In contrast to this, AequeVox's fault localisation algorithm identifies utterances spoken by a group which are likely to be not recognised by ASR systems in the presence of a destructive interference (such as noise). Recent fault localization approaches either aim to highlight the neurons [15] or training code [53] that are responsible for a fault during inference. In contrast, AequeVox highlights words that are likely to be transcribed wrongly without having any access to the ground truth transcription and with only blackbox access to the ASR system.

8 Conclusion

In this work, we introduce AequeVox, an automated fairness testing technique for ASR systems. To the best of our knowledge, we are the first work that explores considerations beyond error rates for discovering fairness violations. We also show that the speech transformations used by AequeVox are largely comprehensible through a user study. Additionally, AequeVox highlights words where a given ASR system exhibits faults, and we show the validity of these explanations. These faults can also be used to identify unfairness in ASR systems.

AequeVox is evaluated on three ASR systems and we use four distinct datasets. Our experiments reveal that speech from non-native English, female and Nigerian English speakers exhibit more errors, on average than speech from native English, male and UK Midlands speakers, respectively. We also validate the fault localization embodied in AequeVox by showing that the predicted non-robust words exhibit *223.8%* more errors than the predicted robust words across all ASRs.

We hope that AequeVox drives further work on systematic fairness testing of ASR systems. To aid future work, we make all our code and data publicly available at **github.com/sparkssss/AequeVox** and **10.5281/zenodo.5897347**

References

1. Audio data augmentation (2021), https://www.kaggle.com/CVxTz/audio-data-augmentation
2. Crowdsourced high-quality nigerian english speech data set (2021), http://openslr.org/70/
3. Grammarly (2021), https://app.grammarly.com/
4. Aggarwal, A., Lohia, P., Nagar, S., Dey, K., Saha, D.: Black box fairness testing of machine learning models. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 625–635 (2019)
5. Asyrofi, M.H., Thung, F., Lo, D., Jiang, L.: Crossasr: Efficient differential testing of automatic speech recognition via text-to-speech. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 640–650 (2020). https://doi.org/10.1109/ICSME46990.2020.00066
6. Buolamwini, J., Gebru, T.: Gender shades: Intersectional accuracy disparities in commercial gender classification. In: Conference on fairness, accountability and transparency. pp. 77–91. PMLR (2018)
7. Butterworth, S., et al.: On the theory of filter amplifiers. Wireless Engineer **7**(6), 536–541 (1930)
8. Calò, A., Arcaini, P., Ali, S., Hauer, F., Ishikawa, F.: Simultaneously searching and solving multiple avoidable collisions for testing autonomous driving systems. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference. pp. 1055–1063 (2020)
9. Carlini, N., Wagner, D.: Audio adversarial examples: Targeted attacks on speech-to-text. In: 2018 IEEE Security and Privacy Workshops (SPW). pp. 1–7. IEEE (2018)
10. Chen, G., Chen, S., Fan, L., Du, X., Zhao, Z., Song, F., Liu, Y.: Who is real bob? adversarial attacks on speaker recognition systems. In: IEEE Symposium on Security and Privacy (2021)
11. Demirsahin, I., Kjartansson, O., Gutkin, A., Rivera, C.: Open-source Multi-speaker Corpora of the English Accents in the British Isles. In: Proceedings of The 12th Language Resources and Evaluation Conference (LREC). pp. 6532–6541. European Language Resources Association (ELRA), Marseille, France (May 2020), https://www.aclweb.org/anthology/2020.lrec-1.804
12. Denton, E., Hutchinson, B., Mitchell, M., Gebru, T., Zaldivar, A.: Image counterfactual sensitivity analysis for detecting unintended bias. In: CVPR 2019 Workshop on Fairness Accountability Transparency and Ethics in Computer Vision (2019)
13. Du, X., Xie, X., Li, Y., Ma, L., Zhao, J., Liu, Y.: Deepcruiser: Automated guided testing for stateful deep learning systems. CoRR **abs/1812.05339** (2018), http://arxiv.org/abs/1812.05339
14. Dwork, C., Hardt, M., Pitassi, T., Reingold, O., Zemel, R.: Fairness through awareness. In: Proceedings of the 3rd innovations in theoretical computer science conference. pp. 214–226 (2012)
15. Eniser, H.F., Gerasimou, S., Sen, A.: Deepfault: Fault localization for deep neural networks. In: Hähnle, R., van der Aalst, W.M.P. (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. Lecture Notes in Computer Science, vol. 11424, pp. 171–191. Springer (2019)

16. Feng, Y., Shi, Q., Gao, X., Wan, J., Fang, C., Chen, Z.: Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 177–188 (2020)
17. Galhotra, S., Brun, Y., Meliou, A.: Fairness testing: testing software for discrimination. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017. pp. 498–510 (2017). https://doi.org/10.1145/3106237.3106277, http://doi.acm.org/10.1145/3106237.3106277
18. Goss, F.R., Zhou, L., Weiner, S.G.: Incidence of speech recognition errors in the emergency department. International journal of medical informatics **93**, 70–73 (2016)
19. Guo, Q., Xie, X., Li, Y., Zhang, X., Liu, Y., Li, X., Shen, C.: Audee: Automated testing for deep learning frameworks. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 486–498. ACM (Dec 2020)
20. Hawley, M.S.: Speech recognition as an input to electronic assistive technology. British Journal of Occupational Therapy **65**(1), 15–20 (2002)
21. Helmke, H., Ohneiser, O., Mühlhausen, T., Wies, M.: Reducing controller workload with automatic speech recognition. In: 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC). pp. 1–10. IEEE (2016)
22. Huang, C., Chen, T., Li, S.Z., Chang, E., Zhou, J.L.: Analysis of speaker variability. In: INTERSPEECH. pp. 1377–1380 (2001)
23. Iwama, F., Fukuda, T.: Automated testing of basic recognition capability for speech recognition systems. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 13–24. IEEE (2019)
24. Jain, A., Upreti, M., Jyothi, P.: Improved accented speech recognition using accent embeddings and multi-task learning. In: Interspeech. pp. 2454–2458 (2018)
25. Johnson, D.H.: Signal-to-noise ratio. Scholarpedia **1**(12), 2088 (2006)
26. Koenecke, A., Nam, A., Lake, E., Nudell, J., Quartey, M., Mengesha, Z., Toups, C., Rickford, J.R., Jurafsky, D., Goel, S.: Racial disparities in automated speech recognition. Proceedings of the National Academy of Sciences **117**(14), 7684–7689 (2020)
27. Kopald, H.D., Chanen, A., Chen, S., Smith, E.C., Tarakan, R.M.: Applying automatic speech recognition technology to air traffic management. In: 2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC). pp. 6C3–1. IEEE (2013)
28. Kreuk, F., Adi, Y., Cisse, M., Keshet, J.: Fooling end-to-end speaker verification with adversarial examples. In: 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). pp. 1962–1966. IEEE (2018)
29. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. vol. 10, pp. 707–710. Soviet Union (1966)
30. Livingstone, S.R., Russo, F.A.: The ryerson audio-visual database of emotional speech and song (ravdess): A dynamic, multimodal set of facial and vocal expressions in north american english. PloS one **13**(5), e0196391 (2018)
31. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30, pp. 4765–4774. Curran Associates, Inc. (2017), http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

32. Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y., Zhao, J., Wang, Y.: Deepgauge: multi-granularity testing criteria for deep learning systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 120–131 (2018)
33. Ma, P., Wang, S., Liu, J.: Metamorphic testing and certified mitigation of fairness violations in NLP models. In: Bessiere, C. (ed.) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020. pp. 458–465
34. Odena, A., Olsson, C., Andersen, D., Goodfellow, I.: Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In: International Conference on Machine Learning. pp. 4901–4911. PMLR (2019)
35. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: Automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017. pp. 1–18 (2017)
36. Phillips, A.: Defending equality of outcome. Journal of political philosophy **12**(1), 1–19 (2004)
37. Qin, Y., Carlini, N., Cottrell, G., Goodfellow, I., Raffel, C.: Imperceptible, robust, and targeted adversarial examples for automatic speech recognition. In: International conference on machine learning. pp. 5231–5240. PMLR (2019)
38. Ribeiro, M.T., Singh, S., Guestrin, C.: "why should I trust you?": Explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016. pp. 1135–1144 (2016)
39. Ribeiro, M.T., Singh, S., Guestrin, C.: Anchors: High-precision model-agnostic explanations. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 32 (2018)
40. Ribeiro, M.T., Wu, T., Guestrin, C., Singh, S.: Beyond accuracy: Behavioral testing of NLP models with checklist. In: Jurafsky, D., Chai, J., Schluter, N., Tetreault, J.R. (eds.) Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020. pp. 4902–4912. Association for Computational Linguistics (2020)
41. Sharma, A., Demir, C., Ngomo, A.C.N., Wehrheim, H.: Mlcheck-property-driven testing of machine learning models. arXiv preprint arXiv:2105.00741 (2021)
42. Sharma, A., Wehrheim, H.: Testing machine learning algorithms for balanced data usage. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 125–135. IEEE (2019)
43. Sharma, A., Wehrheim, H.: Automatic fairness testing of machine learning models. In: IFIP International Conference on Testing Software and Systems. pp. 255–271. Springer (2020)
44. Sharma, A., Wehrheim, H.: Higher income, larger loan? monotonicity testing of machine learning models. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 200–210 (2020)
45. Soremekun, E., Udeshi, S., Chattopadhyay, S.: Astraea: Grammar-based fairness testing. arXiv preprint arXiv:2010.02542 (2020)
46. Sun, Y., Chockler, H., Huang, X., Kroening, D.: Explaining image classifiers using statistical fault localization. In: Vedaldi, A., Bischof, H., Brox, T., Frahm, J. (eds.) Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XXVIII. Lecture Notes in Computer Science, vol. 12373, pp. 391–406. Springer (2020)

47. Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., Kroening, D.: Concolic testing for deep neural networks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 109–119 (2018)
48. Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 303–314 (2018)
49. Udeshi, S., Arora, P., Chattopadhyay, S.: Automated directed fairness testing. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 98–108 (2018)
50. Udeshi, S.S., Chattopadhyay, S.: Grammar based directed testing of machine learning systems. IEEE Transactions on Software Engineering (2019)
51. Verma, S., Rubin, J.: Fairness definitions explained. In: 2018 ieee/acm international workshop on software fairness (fairware). pp. 1–7. IEEE (2018)
52. Wang, J., Chen, J., Sun, Y., Ma, X., Wang, D., Sun, J., Cheng, P.: Robot: Robustness-oriented testing for deep learning systems. In: ICSE '21: 43rd International Conference on Software Engineering (2021)
53. Wardat, M., Le, W., Rajan, H.: Deeplocalize: Fault localization for deep neural networks. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. pp. 251–262. IEEE (2021)
54. Weinberger, S.H., Kunath, S.A.: The speech accent archive: towards a typology of english accents. In: Corpus-based Studies in Language Use, Language Learning, and Language Documentation, pp. 265–281. Brill Rodopi (2011)
55. Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J., See, S.: Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 146–157 (2019)
56. Xie, X., Zhang, Z., Chen, T.Y., Liu, Y., Poon, P.L., Xu, B.: Mettle: a metamorphic testing approach to assessing and validating unsupervised machine learning systems. IEEE Transactions on Reliability **69**(4), 1293–1322 (2020)
57. Zhang, J., Harman, M.: "ignorance and prejudice" in software fairness. In: International Conference on Software Engineering. vol. 43. IEEE (2021)

SMT-Based Planning Synthesis for Distributed System Reconfigurations

Simon Robillard[1](✉) and Hélène Coullon[2]

[1] LIRMM, CNRS, Université de Montpellier, France
[2] IMT Atlantique, Inria, LS2N, Nantes, France

Abstract. Large distributed systems with an emphasis on adaptability are now considered a necessity in many domains, yet reconfiguration of these systems is still largely carried out in an ad hoc fashion, a process that is both inefficient and error-prone. In this paper, we tackle the planification problem for the reconfiguration of distributed systems in the component-based reconfiguration model Concerto. Specifically, given some tasks to execute and a desired final state of the system, we show how to compute a reconfiguration plan that guarantees satisfaction of inter-component dependencies and is also optimized for parallel execution. Our technique relies on an SMT solver to compute the required dependencies between components and ultimately schedule the reconfiguration. We illustrate the use of this technique on a variety of synthetic examples as well as a real use case in the context of an OpenStack system.

Keywords: reconfiguration, planning, synthesis, component models, distributed systems

1 Introduction

Large distributed software systems are now ubiquitous, with component-based systems (e.g., service-oriented architectures or microservices) offering a convenient way to structure large applications. Indeed, isolating functionalities in components and building systems through composition greatly enhances adaptability and scalability of applications, two important requirements for many organizations. This approach is also promoted by the massive adoption of highly-distributed computing infrastructures such as cloud and edge computing.

However, the advantages of distributed architectures come at the price of increased complexity and technical challenges related to observability, coordination, maintenance, etc. Notably, the system reconfigurations that are required to achieve adaptability commonly lead to faults. For example, a study of 597 unplanned outages that affected popular cloud services between 2009 and 2015 found that 16% of them were caused by a software or hardware upgrade [16]. The study concludes that "the complexity of cloud hardware and software ecosystem has outpaced existing testing, debugging, and verification tools". Indeed, testing and debugging methods are largely inadequate in the context of distributed systems, while the adoption of more suitable formal methods remains marginal in

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 268–287, 2022.
https://doi.org/10.1007/978-3-030-99429-7_15

industry. The latter can be attributed to the difficulty of using formal methods and tools. Yet formal methods can lighten the burden of program developers and system administrators instead of adding to it, with synthesis techniques used to generate correct-by-construction programs. In that spirit, we propose to employ a Satisfiability Modulo Theories (SMT) solver to automate the planning of reconfigurations (deployment, migrations, software updates, etc.) of component-based systems, i.e., to generate programs that coordinate the non-functional operations required to perform such reconfigurations. There have been some attempts to synthesize reconfiguration programs for component-based systems (some of them relying on an SMT solver), but they either target ad hoc, non-executable models [20], or are limited to specific cases such as deployment [22], where the problem of executing parallel tasks is reduced to finding a precedence order. In contrast, our work targets the full scope of the component-based reconfiguration model Concerto [9], which provides a formally-defined execution model with expressive constraints on parallelism, as well as a concrete execution engine, making it suitable for formal analysis and experimental work.

In Concerto, reconfigurations are driven by asynchronous *behavior* requests to components. The execution of a behavior may depend on the state of other components: such dependencies are denoted by *ports* that form the interface of components, indicating their provisions and requirements towards each other. Section 2 gives an overview of Concerto, for a more complete presentation, the reader can refer to [9]. Our goal with this work is to automatically generate reconfiguration scripts for systems of Concerto components, i.e., determine required behaviors and coordinate their execution. We take as starting point a reconfiguration goal composed of behaviors to execute over some components and a specification of the final state of the system, particularly the statuses of ports. That goal may be provided by a system administrator, or could have been generated in the context of an autonomic control loop [19]. Importantly, it is a partial specification that typically only mentions parts of the system. For example, an administrator may specify only that a certain utility component should execute a behavior to update its software, whereas the completion of this task actually requires other components to suspend and later resume their activity.

Since a reconfiguration goal can require changes in any component of a system, the search space for reconfiguration scripts grows rapidly with the number of components. To synthesize reconfigurations for large systems, we propose a novel technique that takes advantage of the nature of component-based models. It first solves the problem for each component individually, by considering the internals of the component to find relevant behaviors, under the simplifying assumption that external requirements are all satisfied. Later the method coordinates behaviors over the whole system, relying on a first-order encoding of the scheduling problem and making use of the model-finding capabilities of an SMT solver. If this step fails due to unsatisfied dependencies, individual component reconfiguration goals are refined and the process iterated. Section 3 describes this method, and Section 4 measures its performance and scalability on a variety of synthetic examples, and illustrates its applicability on a real use case.

2 Reconfiguration With Concerto

Components and Assemblies. A distributed system in Concerto is represented as an *assembly*, i.e., a collection of *components* that correspond to control entities for the elements of the system. Components are not intended to represent the functional aspects of those elements, but instead to pilot the actions (installation, maintenance, suspension of service, etc.) required to operate them during their lifespan. In other words, a Concerto component is a wrapper around a new or legacy piece of software (e.g., service, module), typically written by its developer, that acts as replacement for scripts to install and maintain it.

The structural interface of a component is provided by its *provide ports* and *use ports*. Provide ports denote services or data provided by that component when those ports are *active*, while use ports denote requirements that the component has when those ports are active. Ports can be connected in an assembly to allow the satisfaction of component requirements. Connected ports impose synchronization rules between their components: a use port cannot be activated unless connected to an active provide port (the user component may have to wait for that requirement to be fulfilled in order to continue its internal activity) and a provide port cannot be deactivated while connected to an active use port.

Internally, components are characterized by *places* representing milestones in the life cycle, and *transitions* between places, mapped to concrete reconfiguration actions (e.g., starting a virtual machine, downloading an image, etc.). The internal state of a component is given by its places: at any point during execution, one or more places are active. While a place π is active, transitions originating from it can be (simultaneously) fired, after which π ceases to be active. Conversely, a place π' becomes active after the completion of all the transitions that reach it. The completion of a transition takes a non-deterministic duration after firing, modeling the execution of the associated action. Active places also determine the statuses of ports: each port is bound to a set of places, and is active whenever one of them is active. Thus the status of ports changes according to the life cycle of the component. In graphic representations, ports are linked to the place (or set of places, denoted by rounded boxes) to which they are bound.

The last characteristic attribute of a component is its set of *behaviors*. A behavior is a subset of the transitions in a component, such that the associated subgraph is acyclic. At any point in an execution, a component may execute one behavior. Only then can the transitions in that behavior be fired. The behaviors of a component serve as its operational interface: a component may have one behavior including the actions to start it, another including the actions to update it, etc. A component can be requested to execute a behavior, which will determine its evolution and the actions that it performs. Graphically, different behaviors are represented by depicting transitions in different colors.

Figure 1 gives a graphic representation of an assembly. Component `dep1` includes three places (`uninstalled`, `installed` and `running`) and three transitions (arrows between places) that belong to three behaviors (`deploy`, `update`, and `uninstall`). Place `running` is active (denoted by a token) and bound to pro-

vide port `service`, whereas places `installed` and `running` are bound to provide port `config`. Both ports are connected to use ports belonging to `server`.

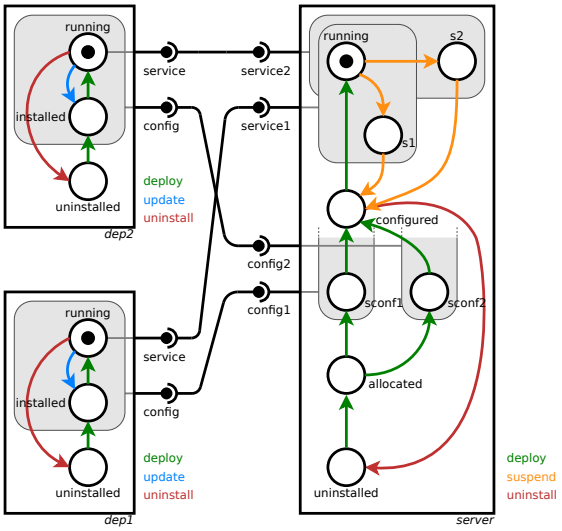


Fig. 1: A Concerto assembly with three components. For readability, the bindings of ports `config1` and `config2` are only partially depicted: they also contain places `configured`, `running`, `s1` and `s2`.

Reconfiguration Scripts. Concerto is equipped with a simple language to execute reconfigurations. Whereas a Concerto component is written by a developer, the reconfiguration language is intended to be used by system administrators or DevOps engineers. Components are piloted through asynchronous requests via the command `pushB(id, b)` that asks the component identified by `id` to execute behavior `b`. The command takes its name from the fact that requests received by a component are queued and asynchronously executed by that component in the order in which they were received. While a component executes a behavior request, transitions in that behavior are fired until the component reaches a state where none of them can be fired. The behavior request is then considered complete, and the component executes the next one, until no more requests remain. The Concerto language also provides synchronization commands: `wait(id)` blocks the execution of the reconfiguration program until the component identified by `id` has executed all behaviors requests submitted to it, and `waitAll()` blocks the execution until all components have executed all pending behavior requests. These three commands allow parallel asynchronous execution in Concerto, leading to more efficient reconfigurations. Based on the description of the components provided by their developers, Concerto can execute reconfiguration scripts , allowing for empirical performance comparisons [10].

The goal of this work is to generate a reconfiguration script using the three aforementioned commands to execute behaviors over components and bring them to a desired state. In addition to those three commands, the Concerto language also provides four usual commands to modify the topology of an assembly: create and delete components, connect and disconnect them. These operations are out of the scope of reconfiguration planning as we define it. Indeed, the decision to modify the topology of the assembly is usually taken by the same entity that determines reconfiguration goals (system administrator or autonomic analysis tool) [15,17] rather than left to the planning phase. Furthermore, if topological changes in the assembly are deemed necessary, they can almost always be implemented through a reconfiguration script with the following steps: (i) creations of components, (ii) creations of connections, (iii) changes in component states, (iv) deletions of connections and (v) deletions of components [5,7]. The main difficulty is to determine the operations of the third step that take the components to a safe state, in particular ensuring that none of the connections that will be deleted include an active use port. Computing a reconfiguration program to lead components to a desired state (or to have them perform some required operations) is the focus of this paper.

As an example, consider the assembly in Figure 1, where all the components are running. We wish to run software updates on `dep1` and `dep2`, but this will deactivate their provide port `service`. To carry out the updates, component `server` must first deactivate its corresponding use ports, which is accomplished by executing its behavior `suspend`. Figure 2a depicts a reconfiguration script that performs this, then returns the components to a running state. No explicit synchronization is needed between the suspension of `server` and the updates: the execution model of Concerto ensures that the updates cannot be executed as long as the provide ports are in use. An explicit synchronization is however needed before re-deploying the server, to prevent it from reactivating its use ports before the updates start. As a side note, the ports `config` (that represent configuration information that is not affected by the update, such as connection information) remain active throughout the reconfiguration: the fine-grained management of dependencies in Concerto avoids a full restart of the system. This assembly also illustrates the capacity of Concerto components to execute actions in parallel: for example, after `server` has reached place `allocated`, it can fire multiple transitions, corresponding to independent reconfiguration actions.

Concerto provides structured semantic tools to design efficient reconfiguration plans with highly parallel, asynchronous execution. However, taking full advantage of these features adds complexity to the internal structure of components and to associated reconfiguration scripts. Automated synthesis of reconfiguration scripts is therefore particularly useful in this context.

3 Reconfiguration Script Synthesis

This section describes the synthesis process used to generate reconfiguration scripts. This process takes as input a description of the current state of the

system, namely the topology of the assembly (components and their connections) and the active places. We assume that the system is in a state where no component has pending behavior requests or ongoing transitions. Besides that information, the synthesis process also depends on a reconfiguration goal that is composed of (i) constraints Γ_{ports} on the final state of ports and (ii) a set of behaviors Γ_{bhv} to execute on designated components.

The constraints Γ_{ports} are given by a partial function that maps specific instances of component ports to a boolean indicating whether that port is required to be active or inactive. A reconfiguration satisfies that goal if it ends in a state such that for any component c and port p, if $\Gamma_{ports}(c, p)$ is defined, the port p of component c is active if and only if $\Gamma_{ports}(c, p) = \top$. Where the value of Γ_{ports} is undefined, any status of the port satisfies the constraint. This means that a reconfiguration goal does not have to specify a unique final state for components, but instead allows for multiple target states. It may appear tedious to specify constraints for all components of an assembly when a reconfiguration is specifically aimed at a subset of it, but in practice the current state of the assembly can be used to guide the choice of Γ_{ports} for those other components. A reasonable strategy might specify that provide ports active before the reconfiguration should remain active, and leave other ports unspecified.

The other element of the reconfiguration goal is the set Γ_{bhv}, where each element is a pair composed of a component and a behavior. The reconfiguration satisfies it if it executes at least all these behaviors on the corresponding components. The set Γ_{bhv} alone may not correspond to a feasible reconfiguration. For example, a system administrator wishing to update the components of Figure 1 might give a behavior goal $\Gamma_{bhv} = \{(\texttt{dep1}, \texttt{update}), (\texttt{dep2}, \texttt{update})\}$ and a port goal Γ_{ports} that maps every port instance to $\top$. The behaviors listed in that reconfiguration goal are not enough to carry it out, as it lacks a behavior to deactivate the use ports of the server prior to the update, and behaviors to reactivate all ports after the update. The synthesis process must therefore deduce necessary behaviors to carry out the reconfiguration goal, then schedule their executions in a suitable order. It proceeds as follows:

1. for each component independently, we find a sequence of behaviors that satisfies the goal, assuming that ports requirements are fulfilled (Subsection 3.1);
2. we find a global schedule for these sequences of behaviors (Subsection 3.2);
3. if the scheduling problem is found unsatisfiable, we analyze the incomplete schedule to deduce unsatisfied port requirements, compute additional reconfiguration sub-goals and iterate the process (Subsection 3.3);
4. once a feasible solution has been found, we attempt to optimize it by relaxing synchronization conditions (Subsection 3.4).

3.1 Determining Sequences of Component Behaviors

A procedure $\mathit{localSeq}(c, \mathit{act}^c, \Gamma_{bhv}, \Gamma_{ports})$ finds a sequence of behaviors that satisfies a reconfiguration goal Γ for a single component c starting in a state with active places act^c. This is achieved by enumerating all sequences of behaviors

with at most one occurrence of any behavior, and selecting one that satisfies the goal constraints. In practice, this enumeration is short because the number of behaviors of a component is usually small. More importantly, for a given component state (denoted by its active places), many behaviors do not have transitions originating from the active places. Since executing these behaviors would not have any effect, they can be ignored during the enumeration. Consequently, the number of useful sequences of behaviors to analyze is often much lower than the number of permutations. If no satisfying sequence is found by *localSeq*, then the problem has no solution, and the whole synthesis process fails. However, if multiple solutions are returned, the best possible sequence is picked, according to some (possibly user-defined) selection criterion. Some interesting optimization criteria are: the length of a sequence, its execution time (if time estimations are available for individual transitions, this may be computed with great accuracy [10]), the number of transitions it executes sequentially, or the number of ports it (de)activates. In our experiments, we used this last criterion, as it picks the component reconfiguration that is least likely to induce changes in other components, leading to simpler and potentially faster reconfiguration plans.

In order to coordinate sequences or behaviors across the assembly, we keep track of ports requirements and activity during each behavior of a sequence. In particular, for each behavior in a sequence, we record use ports of the component that are activated at least once by the behavior (they must be connected to an active provide port during the execution of the behavior), and provide ports that are deactivated at least once (they must *not* be connected to an active use port). In addition, we also record the status of each port at the end of the behavior. This information is computed with a simple traversal of the behavior graph, starting from the places that are active at the beginning of the behavior.

In the example of the update for the assembly in Figure 1, *localSeq* determines that components `dep1` and `dep2` should each execute the sequence [`update`, `deploy`]: the first behavior is included in Γ_{bhv} and the second is required to take the components to a state that satisfies Γ_{ports}.

3.2 Assembly-Level Reconfiguration Scheduling

Once sequences of behaviors to execute over each component have been determined, we turn our attention to the whole assembly and attempt to compute a sequence of reconfiguration commands (specifically, behavior requests and synchronization requests) that execute these behaviors. The challenge is to coordinate these behaviors in a way that satisfies all port requirements. To facilitate coordination and to restrict the search space, we specifically try to generate a reconfiguration composed of *steps*, such that each component executes at most one behavior per step, and each step is followed by a global synchronization request. This assumption on parallelism is reminiscent of the BSP model [4]. Figure 2b gives an example of such a reconfiguration, to be compared with Figure 2a, which achieves the same result with fewer synchronization points.

```
pushB(server, suspend)
pushB(dep1, update)
pushB(dep2, update)
pushB(dep1, deploy)
pushB(dep2, deploy)
wait(dep1)
wait(dep2)
pushB(server, deploy)
wait(server)
```

(a) Target reconfiguration program.

```
pushB(server, suspend)
waitAll()
pushB(dep1, update)
pushB(dep2, update)
waitAll()
pushB(dep1, deploy)
pushB(dep2, deploy)
waitAll()
pushB(server, deploy)
waitAll()
```

(b) A reconfiguration with four synchronized steps.

Fig. 2: A reconfiguration plan to perform updates on components `dep1` and `dep2` of the assembly in Figure 1, then restore the system to a working state.

SMT Constraints To find a reconfiguration plan, ordering constraints and port requirements are encoded as a problem in a many-sorted first-order logic (i.e., the logic is equipped with *sorts* that partition the domain, similarly to a simple type system), and an SMT solver is used to obtain a solution. That encoding of the scheduling problem centers around a sort $\mathsf{Behavior}$, with a finite number of elements that represent the behaviors to schedule. The main task of the SMT solver is to find an interpretation for a function $\mathsf{schedule}$ that maps behaviors to a reconfiguration step during which to execute them. Conceptually, $\mathsf{schedule}$ could range over natural numbers, with behavior b executed at the ith step if $i = \mathsf{schedule}(\mathsf{b})$. However, such a model would require constraints with universal quantifiers over natural numbers, which pose a challenge for SMT solvers. It is also unnecessary, since there are only a finite number of behaviors to schedule: the number of steps required is at most the number of behaviors, when only one component executes a behavior at each step. If behaviors are executed in parallel over different components, fewer steps are required. Consequently, to improve the performance of the solver, the different steps of the reconfiguration are represented by another finite-domain sort Step, with elements $\mathsf{step}_1, \ldots \mathsf{step}_\mathsf{n}, \mathsf{step}_\mathsf{final}$. The element $\mathsf{step}_\mathsf{final}$ represents the ultimate state of the system rather than a reconfiguration step. Accordingly, the scheduling function has the signature $\mathsf{schedule} : \mathsf{Behavior} \to \mathsf{Step}$, and the problem contains the constraint $\mathsf{schedule}(\mathsf{b}) \neq \mathsf{step}_\mathsf{final}$ for each behavior b.

A successor function $\mathsf{succ} : \mathsf{Step} \to \mathsf{Step}$ is needed to describe the effect of a reconfiguration step on the subsequent state of the system. Constraints $\mathsf{succ}(\mathsf{step}_\mathsf{i}) = \mathsf{step}_{\mathsf{i}+1}$ (for $0 \leqslant i < n$), $\mathsf{succ}(\mathsf{step}_\mathsf{n}) = \mathsf{step}_\mathsf{final}$ and $\mathsf{succ}(\mathsf{step}_\mathsf{final}) = \mathsf{step}_\mathsf{final}$ define the interpretation of succ. Likewise, to easily express sequentiality constraints, a function $\mathsf{int} : \mathsf{Step} \to \mathsf{Int}$ maps each step to its step number, as defined by constraints $\mathsf{int}(\mathsf{step}_\mathsf{i}) = i$. With this function, sequentiality is easily expressed: for any two consecutive behaviors b_1 and b_2 in the sequence of behaviors to schedule for a given component, the constraint $\mathsf{int}(\mathsf{schedule}(\mathsf{b}_1)) < \mathsf{int}(\mathsf{schedule}(\mathsf{b}_2))$ is added. This function reintroduces an infinite domain, which we sought to eliminate with the sort Step. However, since the problem contains

no quantifiers over integers, the solver only has to check that the aforementioned formula is satisfied by a speculated interpretation of schedule. This limited form of integer reasoning has a negligible impact on the search.

The main difficulty in scheduling a reconfiguration lies in ensuring that ports requirements are satisfied for each behavior of a component. A predicate $\mathsf{act_p} : \mathsf{Step} \rightarrow \mathsf{Bool}$ is introduced for each (use or provide) port p to indicate the activity status of the port at the beginning of reconfiguration steps. The status of each port p after each behavior b is uniquely defined, as determined during the computation of the sequences of behaviors of the component to which the port belongs. Correspondingly, a constraint $[\neg]\mathsf{act_p}(\mathsf{succ}(\mathsf{schedule}(\mathsf{b})))$ is added to reflect that status. The square brackets denote the absence or presence of the negation, depending on whether the port is inactive or active at the end of the behavior. Conversely, the status of a port cannot change if its component is not executing a behavior. For a component with behaviors $\mathsf{b_1}, \ldots, \mathsf{b_n}$, the constraint $\mathsf{schedule}(\mathsf{b_1}) \neq \mathsf{step_i} \wedge \cdots \wedge \mathsf{schedule}(\mathsf{b_n}) \neq \mathsf{step_i} \implies (\mathsf{act_p}(\mathsf{step_i}) \iff \mathsf{act_p}(\mathsf{succ}(\mathsf{step_i})))$ is added for every step i such that $0 \leqslant i < n$. Ports requirements can then be modeled. Let u be a use port that needs to be provided (i.e., connected to an active provide port) during behavior b, and p the provide port to which it is connected, the constraint $\mathsf{act_p}(\mathsf{schedule}(\mathsf{b}))$ ensures that p is active (and u provided) when b begins. Conversely, for a provide port p deactivated by a behavior b and connected to a use port u, $\neg\mathsf{act_u}(\mathsf{schedule}(\mathsf{b}))$ ensures that u is inactive when b begins. Furthermore, for any behavior b that activates a use port u and any behavior b' that deactivates the connected provide port p, the constraint $\mathsf{schedule}(\mathsf{b}) \neq \mathsf{schedule}(\mathsf{b}')$ ensures that the behaviors are executed at different steps, hence separated by a synchronization barrier.

The problem[3] is passed to an SMT solver. If satisfiable, the interpretation found for schedule is used to build a reconfiguration script such as in Figure 2b.

Note that the scheduling problem could be encoded as a SAT problem. However, SMT solvers can reason about the theory EUF (equality and uninterpreted functions) using a dedicated congruence algorithm. We also use (non-recursive) data types, for which some SMT solvers have a dedicated reasoning algorithm [3], to represent the domains of Behavior and Step. These capabilities allow us to encode the problem straightforwardly and obtain solutions efficiently. Also note that the size of the scheduling problem is only a function of the number of behaviors to schedule and the number of component ports, but does not depend on the internal complexity of components, so that optimized components with several parallel transitions will not adversely affect the synthesis method.

3.3 Determining Missing Behaviors

Until now, we have considered the scheduling problem under the assumption of a fixed sequence of behaviors to schedule for each component. In general, a set of behaviors may have no feasible schedule. For example, it is not possible to

[3]Illustrating instances for the running example, in the SMT-LIB file format, can be found at https://doi.org/10.5281/zenodo.5820571.

fully execute the behavior `update` on components `dep1` and `dep2` of the assembly in Figure 1 without first deactivating the use ports `service1` and `service2` of component `server`, i.e., executing its behavior `suspend`. To plan reconfigurations for an incomplete set of behaviors, we use our SMT encoding of the scheduling problem to detect the point in the reconfiguration at which additional changes must be performed, then we create new component reconfiguration sub-problems and use the solutions to augment the sequences of behaviors to schedule.

Let S be a mapping that associates to each component a sequence of behaviors (i.e., the sequence to be executed by that component, as determined in Subsection 3.1), a *maximal executable schedule* S' of S is a mapping that associates to each component c a prefix of $S(c)$, such that (i) the scheduling problem corresponding to the sequences in S' has a solution (ii) no reconfiguration problem built by extending a prefix in S' with one behavior has a solution. Intuitively, a maximal executable schedule is a point up to which the reconfiguration S can be carried out, before unsatisfied port requirements prevent further execution.

Procedure 1 iteratively computes a maximal executable schedule S' and uses the resulting information to refine the sequences of behaviors to execute for each component, until a solution is found that executes them all. By analyzing the statuses of ports in the assembly at the end of the execution of S' (which depend only on the last behavior in each sequence), and comparing them to the requirements of the first unscheduled behaviors in S, we deduce a set of provide ports to activate and use ports to deactivate to allow further scheduling of S, and compute intermediary ports constraints Γ'_{ports}. For each component c that does not have unscheduled behaviors in S, we determine a sequence s_1 of behaviors that satisfies this intermediate goal (assuming that the component starts with active places $act^c_{S'}$ corresponding to its state after executing the last behavior in $S'(c)$) and a sequence s_2 that takes the component from its state after executing s_1 (active places $act^c_{s_1}$) to one that satisfies the port constraints Γ_{ports} of the original goal. Sequences of behaviors to execute are thus extended ($[]$ denotes the empty sequence, and $s_1 \cdot s_2$ the concatenation of two sequences). To ensure a monotonic search, sequences are extended only for components c without unscheduled behaviors in S, i.e., not the components that brought about the intermediary goal Γ'_{ports}. If no such extension can be found ($\neg progress$), the scheduling of S is blocked by a circular dependency between components and the synthesis process fails. If the procedure terminates, it returns a reconfiguration script corresponding to a solution of the scheduling problem of S.

Consider the example of running updates in the assembly of Figure 1. Initially (see Subsection 3.1), the mapping S of sequences of behaviors computed with *localSeq* is defined by $S(\texttt{dep1}) = S(\texttt{dep2}) = [\texttt{update}, \texttt{deploy}]$, and $S(\texttt{server}) = [\,]$, because Γ_{bhv} does not include any behavior for that component, and the component is already in a state that satisfies Γ_{ports}. This combination of sequences of behaviors has no feasible schedule. In particular, the mapping S' that associates to every component the empty sequence is found to be a maximal executable schedule of S. The first unscheduled behaviors in S are two instances of `update`, they require use ports `service1` and `service2` of component `server` to be deac-

```
Procedure globalSolution(A, Γ_bhv, Γ_ports) is
    for c ∈ A do S(c) ← localSeq(c, act^c_A, Γ_bhv, Γ_ports);
    while findMaxExecSchedule(S) ≠ S do
        S' ← findMaxExecSchedule(S) ;
        Γ'_ports ← port conditions required to execute, for every component c,
          the first behavior in S(c) that is not in S'(c);
        progress ← false ;
        for c ∈ A such that S'(c) = S(c) do
            s1 ← localSeq(c, act^c_S', Γ_bhv \S'(c), Γ'_ports) ;
            s2 ← localSeq(c, act^c_s1, ∅, Γ_ports) ;
            if s1 ≠ [] or s2 ≠ [] then
                S(c) ← S'(c) · s1 · s2 ;
                progress ← true ;
            end
        end
        if ¬ progress then fail ;
    end
    return reconfigurationScriptOfSolution(S) ;
end
```

Procedure 1: Synthesizes a reconfiguration script.

tivated. Consequently, two new reconfiguration sub-goals are created for `server`. The first requires it to reach a state where the two ports are deactivated, a call to *localSeq* returns the solution $s_1 = [\texttt{suspend}]$. From the resulting component state, the second reconfiguration sub-goal requires `server` to go to a state that satisfies Γ_{ports}, in this case *localSeq* returns the sequence $s_2 = [\texttt{deploy}]$. S is updated so that $S(\texttt{server}) = [\texttt{suspend}, \texttt{deploy}]$. At this point, S is found to be a maximal executable schedule of itself, and the corresponding solution is returned, i.e., the reconfiguration plan in Figure 2b. Note that Procedure 1 is not guaranteed to terminate, nor is it a complete search algorithm. In particular, it relies on two heuristics: the selection function used when *localSeq* finds multiple candidate sequences, and the choice of maximal executable schedule for a given mapping S.

Computing a Maximal Executable Schedule Procedure 1 relies on a function *findMaxExecSchedule* to compute a maximal executable schedule of a mapping S, illustrated in Procedure 2, that maintains a mapping containing prefixes of elements in S (initially mapping every component to the empty sequence) and incrementally extends those prefixes, checking every time the satisfiability of the corresponding scheduling problem. This procedure calls the SMT solver to check the satisfiability of the scheduling problems. In the actual implementation, some simple checks are also used to quickly detect some trivially unsatisfiable or satisfiable instances of the scheduling problem, although these are left out of Procedure 2 for clarity. The procedure continues until all behaviors have been included or no additional behavior can be scheduled. A maximal executable

schedule always exists (the mapping that associates every component to the empty sequence always has a satisfiable scheduling problem, and may be maximal), and *findMaxExecSchedule* always finds one. However maximal executable schedules are not unique, and a bad choice may result in an ineffective reconfiguration plan. In the example above, during the second iteration, the mapping S of sequences to schedule is defined by $S(\texttt{server}) = [\texttt{suspend}, \texttt{deploy}]$ and $S(\texttt{dep1}) = S(\texttt{dep2}) = [\texttt{update}, \texttt{deploy}]$. S itself is a maximal executable schedule of S, but so is the mapping S' defined by $S'(\texttt{server}) = [\texttt{suspend}, \texttt{deploy}]$ and $S'(\texttt{dep1}) = S'(\texttt{dep2}) = [\,]$. S' corresponds to the case where the server is restarted too early. Picking this maximal executable schedule will ultimately lead to a reconfiguration that stops the server at least twice. To avoid this, a good heuristic for *findMaxExecSchedule* is to extend in priority the prefixes for which the added behavior is least likely to affect other components, i.e., those that deactivate the fewest provide ports and activate the fewest use ports.

```
Procedure findMaxExecSchedule(S) is
    suffixes ← S ;
    for c such that suffixes(c) is defined do prefixes(c) ← [];
    progress ← true ;
    while progress do
        progress ← false ;
        for c such that suffixes(c) ≠ [] do
            b ← head(suffixes(c)) ;
            if the scheduling problem for prefixes extended with b is satisfiable
             then
                progress ← true ;
                prefixes(c) ← prefixes(c) · [b] ;
                suffixes(c) ← tail(suffixes(c)) ;
            end
        end
    end
    return prefixes ;
end
```

Procedure 2: Computes a maximum executable schedule for sequences of behaviors S.

3.4 Relaxation of Synchronization Barriers

The assumption that reconfigurations should proceed in globally synchronized steps, although useful to find a solution, severely limits the potential for inter-component parallelism, a key feature of Concerto. A final optimization stage takes the reconfiguration plan with synchronized steps and relaxes synchronization where possible. First, every command `waitAll()` is replaced with a sequence

of commands `wait(c)` for every component c that executes a behavior in the preceding step. This preserves the semantics of the reconfiguration and makes the targets of synchronization explicit. Then, for a given step i and a given command `wait(c)` after this step, we apply the following rule: if for all behaviors executed by c since the last command `wait(c)` up to step i, no provide (resp., use) port is deactivated (resp., activated) and connected to a use (resp., provide) port that is activated (resp., deactivated) at step $i+1$, then `wait(c)` can be delayed until after step $i+1$. This rule is applied for every step in order, delaying barriers as late as possible and removing duplicates. This transformation reduces the number of barriers yet ensures that behaviors with conflicting effects on ports remain separated by an explicit synchronization. Port requirements for behaviors do not have to be taken into account, as the Concerto execution model ensures implicit synchronization for those. As an example, this optimization applied to the reconfiguration plan in Figure 2b yields the one in Figure 2a.

4 Experiments

The implementation described here, the examples, and the experimental results are available at https://doi.org/10.5281/zenodo.5820571.

4.1 Implementation

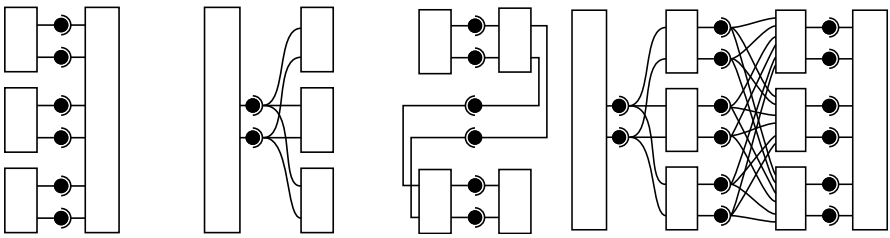
We implemented the synthesis process in a Python tool that attempts to produce a reconfiguration script for a given assembly and reconfiguration goal. The process is entirely automated. Given a description of an assembly and a reconfiguration goal, it generates relevant scheduling problems and interacts with an SMT solver to generate reconfiguration programs. Intermediate scheduling problems can be output in the SMT-LIB file format, the standard used by most SMT solvers [2], and can be solved using any solver that complies with version 2.6 of the SMT-LIB standard. The preferred mode of operation for our tool does not output files, but interacts with the SMT solver Z3 [23] through the Z3 Python API. This interface makes it easy to analyze interpretations returned by the solver for satisfiable problems, and thus to reconstruct schedules. This is the mode of operation used to conduct the experiments described below.

4.2 Results Over Synthetic Examples

To test our technique on a variety of cases, we devised assemblies with four types of topology. In *central-user* assemblies, a set of provider components, each with a pair of provide ports, is connected to different use ports of one central user component. In *central-provider* assemblies, one central provider component has a pair of provide ports that is connected to (a pair of use ports of) multiple other components. In *linear* assemblies, components form a chain such that each component has a pair of provide ports connected to the pair of use ports of the next component. In *stratified* architectures, components are organized in

levels containing up to three components, such that each component in a level has a pair of provide ports connected to use ports on every component in the level above (i.e., a provide port can be connected to up to three use ports). Every component in these assemblies is equipped with behaviors to deploy it, update or suspend it, and uninstall it. Figure 3 depicts those four topologies, with internal nets of components omitted for clarity. As an example of the internal structure of components, Figure 1 shows the central-user assembly with three components. For other types and sizes of assembly, components follow similar internal structures, adapted to offer adequately many ports.

For each architecture, we generated assemblies with 10, 30 and 100 components (scaling the number of providers for *central-user*, the number of users for *central-provider*, the length of the chain for *linear* or the number of levels for *stratified*), and ran three scenarios. The *deployment* scenario starts with all components uninstalled, Γ_{ports} requires the activation of a provide port on the last component(s) in the dependency order, while Γ_{bhv} is empty. The *update* scenario starts with all components running, Γ_{ports} requires a similar final state, and Γ_{bhv} includes update behaviors for components that are first in the dependency order and no behavior for the others. The *uninstall* scenario starts with all components running, Γ_{ports} requires the deactivation of all ports, while Γ_{bhv} is empty. Each scenario affects every component of the assembly.



(a) Central-user (b) Central-provider (c) Linear (d) Stratified

Fig. 3: The four assembly topologies in synthetic examples.

Table 1 describes the solving process and resulting solution for these 36 examples. Experiments were executed on a computer with an 8-core 1.6GHz processor and 16 GiB of RAM. Solutions were successfully generated for all but 4 examples (the process was aborted after one hour). For 21 of them, the process took less than a minute. Results indicate that the solving time, and ultimately the success of the method, depend on the topology of the assembly: the assemblies for which some reconfigurations could not be computed within one hour are those with long chains of dependencies (linear and stratified assemblies with 100 components). This can be explained in two ways: firstly, the propagation of port requirements and the deduction of missing behaviors requires a number of iterations of the main loop of Procedure 1 proportional to the length of the longest chain of dependency. Secondly, architectures with long chains of dependencies

are less conducive to parallel execution of behaviors, and therefore the instances of scheduling problems solved have a high number of steps, leading to a large search space and long solving times. For example, the deployment of 100 component in the linear architecture ultimately requires 100 steps. For each of the 17 instances of the scheduling problem solved to compute that reconfiguration, the SMT solver took on average 147 seconds to return a solution. In contrast, to deploy 100 components in the central-user architecture, the reconfiguration script requires only 2 steps, as a result the SMT solver was able to return a solution after only 0.21 seconds on average each time it was called. For difficult problems, the solving time is dominated by calls to the SMT solver as shown in the solving time column of Table 1 (in parentheses, the cumulated time taken by the SMT solver). Overall, these examples show that our method is able to plan reconfigurations affecting large number of components. Furthermore, architectures with a very large number of components, such as microservice architectures, typically have a shallow depth rather than long chains of dependencies, and scale horizontally [21,24,25], similarly to our central-user and central-provider architectures. Our method scales well in those conditions.

Writing a correctly coordinated reconfiguration plan with tens of asynchronous behaviors is a non-trivial task. It is particularly difficult when explicit synchronizations commands are needed in the reconfiguration script. The execution model of Concerto ensures that this is seldom necessary, but some synchronization barriers are required, e.g., in the update scenarios to prevent early restarts that would block the updates. Our synthesis technique determines synchronization points required for completion of the reconfigurations, but it also avoids synchronization points that would slow the execution unnecessarily. It performs these tasks quickly, with a time gain that is especially significant when compared to the service interruption that an incorrect reconfiguration would cause.

4.3 OpenStack Use Case

We also tested our method on a real OpenStack system. OpenStack is the de facto standard open-source solution to address the IaaS level of the cloud paradigm, it can be seen as the open-source operating system of the cloud.

In previous work [8], Madeus, a subset of Concerto restricted to deployment, was used to deploy an OpenStack system. Following the deployment strategy of the reference production deployment tool Kolla, 11 components were specified, resulting in a real OpenStack deployment up to 70% faster than Kolla. Here we use the same components, extended with behaviors for reconfiguration. We analyzed the official installing, updating and uninstalling procedures of OpenStack to design the associated internal nets. Figure 4 depicts those components and their connections, with details of the internal structure depicted for the four main components, and omitted for clarity on seven others. The reconfiguration starts with all components running. The reconfiguration scenario requires an update of the database component ($\Gamma_{bhv} = \{(\texttt{mariadb}, \texttt{update}), (\texttt{mariadb}, \texttt{deploy})\}$) and Γ_{ports} specifies that all ports must eventually return to their initial (active) state. Our method generates a reconfiguration plan in 1.95 seconds, correctly deducing

	assembly		solving		plan	
	arch.	size	smt	time (s)	steps	bhvs
deployment	c-user	10	2 *(2)*	0.25 *(0.02)*	2	10
		30	5 *(5)*	1.99 *(0.18)*	2	30
		100	17 *(17)*	23.94 *(3.59)*	2	100
	c-provider	10	2 *(2)*	0.33 *(0.08)*	2	10
		30	5 *(5)*	3.95 *(1.80)*	2	30
		100	17 *(17)*	93.07 *(68.67)*	2	100
	linear	10	2 *(2)*	0.40 *(0.08)*	10	10
		30	5 *(5)*	9.19 *(4.27)*	30	30
		100	17 *(17)*	2689.86 *(2512.22)*	100	100
	stratified	10	3 *(3)*	0.64 *(0.09)*	5	10
		30	6 *(6)*	12.04 *(5.28)*	12	30
		100	18 *(18)*	1274.52 *(1121.40)*	35	100
update	c-user	10	6 *(5)*	1.69 *(0.78)*	4	20
		30	12 *(11)*	25.07 *(18.59)*	4	60
		100	36 *(35)*	1737.29 *(1654.67)*	4	200
	c-provider	10	13 *(4)*	1.79 *(0.41)*	4	20
		30	40 *(11)*	22.98 *(9.97)*	4	60
		100	133 *(34)*	685.54 *(541.69)*	4	200
	linear	10	50 *(5)*	12.72 *(3.26)*	20	20
		30	446 *(11)*	1388.50 *(825.06)*	60	60
		100	–	–	–	–
	stratified	10	20 *(6)*	14.55 *(5.82)*	13	26
		30	147 *(16)*	2306.85 *(1885.17)*	34	86
		100	–	–	–	–
interruption	c-user	10	3 *(3)*	0.54 *(0.14)*	3	11
		30	6 *(6)*	6.01 *(2.76)*	3	31
		100	18 *(18)*	162.51 *(125.51)*	3	101
	c-provider	10	4 *(4)*	0.91 *(0.30)*	3	19
		30	10 *(10)*	12.36 *(7.25)*	3	59
		100	34 *(34)*	571.73 *(514.48)*	3	199
	linear	10	10 *(10)*	1.30 *(0.19)*	10	10
		30	30 *(30)*	39.31 *(13.69)*	30	30
		100	–	–	–	–
	stratified	10	4 *(4)*	2.50 *(1.10)*	9	19
		30	11 *(11)*	132.59 *(108.53)*	23	59
		100	–	–	–	–

Table 1: Results of the synthesis process on synthetic examples. For each problem, the table indicates the architecture of the assembly and (arch.) its number of components (size), the number of problems solved by the SMT solver (smt) followed in parentheses by the number of those that were found satisfiable, the total solving time in seconds followed in parentheses by the cumulated time taken by the SMT solver (time), the number of steps in the solution before relaxation of the synchronization barriers (steps), and the number of behaviors executed in that solution (bhvs).

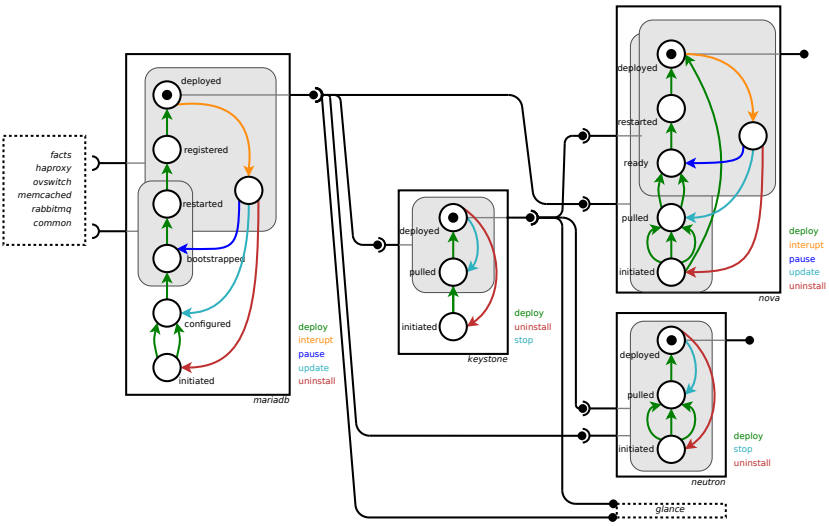


Fig. 4: A Concerto assembly for an OpenStack system.

missing behaviors for `mariadb` as well as components affected by the interruption of its service, i.e., `keystone`, `nova`, `neutron`, and `glance`. The generated plan coordinates 12 behaviors on these 5 components. After optimization, it includes only 2 synchronization points needed to ensure the complete re-deployment of `mariadb` and `keystone`, whose services are required by other components.

While the scale of this use case may seem limited, its architecture is not trivial. This real-world scenario leads to a complicated synchronization problem. The 12 behaviors in the reconfiguration program require 8 global synchronization steps before optimization. The optimization phase reduces this to 2 individual synchronization points, thus enhancing the level of parallelism and asynchrony of the reconfiguration program, while preserving its correctness. A DevOps engineer or system administrator would be challenged to write such a program without errors or unnecessary synchronization points, whereas our solution only requires them to specify a reconfiguration goal.

5 Related work

For models with fixed component life cycles, planning and scheduling techniques have been used to plan reconfigurations [1, 13]. Pre-established protocols can also be used: while such solutions are in general less flexible, they have desirable features such as decentralized coordination [14] or recovery policies [6]. Comparatively fewer works study the problem of reconfiguration planning in models with programmable component life cycles, such as Concerto. Kikuchi et al. [20] synthesize reconfiguration plans with a model finder. Unlike us, they assume that all available reconfiguration operations are given in the input of the scheduling problem, which may limit scalability. Operations and reconfiguration goal are

encoded in the Alloy specification language, and synthesis is performed by the Alloy Analyzer. This work relies on a simple ad hoc component-based model, with reconfiguration operations that must be sequentially ordered. The model does not have specific execution semantics, instead the list of operations has to be given by the user, with their effects described as constraints on the states before and after the operations. Therefore the correctness of the correspondence between the synthesized procedure and its executable counterpart depends on the user. Metis [22] closes that gap between planning and execution, as it schedules deployment plans for distributed systems in the Aeolus model [12], which has formal execution semantics. The authors first describe the problem as a generic planning problem and use standard planners to solve it, then present a specialized solving algorithm. Metis is limited to deployment rather than general reconfiguration, making the computation of dependencies more straightforward. Aelous shares many similarities with Concerto, but lacks intra-component parallelism and asynchronous commands in its reconfiguration language. These features improve the efficiency of reconfigurations but also make them more difficult to plan. Note that these features can also be represented through planning and scheduling problems [18], typically solved by approximation.

The problem of determining reconfiguration goals (i.e., the analysis phase) is complementary to the planning problem. Engage [15] uses a SAT solver to build a complete target configuration (a set of components to deploy) from a partial specification, based on a hierarchical specification of a distributed software stack. It also performs limited planning, namely sequentially ordering deployments. Engage does not account for the state of the system, and is thus limited to initial deployments or reconfigurations from the ground up. Zephyrus [11] and ConfSolve [17] are two tools to infer, from the state of the system/environment, a target configuration that could be used as an entry of our planning tool.

6 Conclusion

We have described a synthesis method for reconfiguration plans of component-based systems, that relies on (i) finding local solutions at the component level, (ii) finding a schedule that coordinates those solutions at the assembly level, with the help of an SMT solver, (iii) determining unsatisfied dependencies to refine the reconfiguration goal until it becomes satisfiable, and (iv) optimizing the synthesized reconfiguration plan to improve its level of parallel and asynchronous execution. Dividing the problem in this manner, as opposed to attempting to solve it at once with an SMT solver, is a key to solving large instances, although it leads to incompleteness (the third step relies on an incomplete search guided by some heuristic choices). This design decision does not appear to affect the success of the method or the quality of synthesized plans, and allows the technique to scale to applications with large number of components, as demonstrated in our experiments on synthetic examples and a real use case. To improve scalability on complex architectures, this technique could be adapted to a hierarchical composition model, which would lend itself to a recursive resolution algorithm.

References

1. Arshad, N., Heimbigner, D., Wolf, A.L.: Deployment and dynamic reconfiguration planning for distributed software systems. In: Proceedings. 15th IEEE International Conference on Tools with Artificial Intelligence. pp. 39–46. IEEE (2003)
2. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org` (2016)
3. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. Journal on Satisfiability, Boolean Modeling and Computation **3**, 21–46 (2007)
4. Bisseling, R.H.: Parallel Scientific Computation: A Structured Approach Using BSP. Oxford Unviersity Press (2020)
5. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: 35th International Conference on Software Engineering (ICSE). pp. 13–22 (2013)
6. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 13–22. IEEE (2013)
7. Cansado, A., Canal, C., Salaün, G., Cubo, J.: A formal framework for structural reconfiguration of components under behavioural adaptation. Electronic Notes in Theoretical Computer Science **263**, 95–110 (2010)
8. Chardet, M., Coullon, H., Pérez, C., Pertin, D., Servantie, C., Robillard, S.: Enhancing separation of concerns, parallelism, and formalism in distributed software deployment with Madeus (2020), https://hal.inria.fr/hal-02737859, preprint
9. Chardet, M., Coullon, H., Robillard, S.: Toward safe and efficient reconfiguration with Concerto. Science of Computer Programming **203** (2021)
10. Chardet, M., Hélène, C., Perez, C.: Predictable efficiency for reconfiguration of service-oriented systems with Concerto. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). pp. 340–349 (2020)
11. Di Cosmo, R., Lienhardt, M., Treinen, R., Zacchiroli, S., Zwolakowski, J., Eiche, A., Agahi, A.: Automated synthesis and deployment of cloud applications. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 211–222 (2014)
12. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: A component model for the cloud. Information and Computation pp. 100–121 (2014)
13. El Maghraoui, K., Meghranjani, A., Eilam, T., Kalantar, M., Konstantinou, A.V.: Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. pp. 404–423. Springer (2006)
14. Etchevers, X., Coupaye, T., Boyer, F., De Palma, N.: Self-configuration of distributed applications in the cloud. In: 2011 IEEE 4th International Conference on Cloud Computing. pp. 668–675. IEEE (2011)
15. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: A deployment management system. In: ACM SIGPLAN PLDI. pp. 263–274 (2012)
16. Gunawi, H.S., Hao, M., Suminto, R.O., Laksono, A., Satria, A.D., Adityatama, J., Eliazar, K.J.: Why does the cloud stop computing? lessons from hundreds of service outages. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. pp. 1–16 (2016)

17. Hewson, J.A., Anderson, P., Gordon, A.: A declarative approach to automated configuration. In: lisa'12: Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques. pp. 51–66 (2012)
18. Keller, A., Hellerstein, J.L., Wolf, J.L., Wu, K.L., Krishnan, V.: The CHAMPS system: Change management with planning and scheduling. In: 2004 IEEE/IFIP Network Operations and Management Symposium (IEEE Cat. No. 04CH37507). vol. 1, pp. 395–408. IEEE (2004)
19. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer (2003)
20. Kikuchi, S., Tsuchiya, S., Hiraishi, K.: Synthesis of configuration change procedure using model finder. IEICE TRANSACTIONS on Information and Systems **96**(8), 1696–1706 (2013)
21. Kratzke, N., Quint, P.C.: Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. Journal of Systems and Software **126**, 1–16 (2017). https://doi.org/https://doi.org/10.1016/j.jss.2017.01.001
22. Lascu, T.A., Mauro, J., Zavattaro, G.: A planning tool supporting the deployment of cloud applications. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence. pp. 213–220 (2013)
23. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 337–340. Springer (2008)
24. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice Architecture. O'Reilly Media, Inc. (2016)
25. Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: A systematic mapping study. In: CLOSER (2018)

Semantic Clone Detection via Probabilistic Software Modeling

Hannes Thaller[1](✉), Lukas Linsbauer[2], and Alexander Egyed[1]*

[1] Johannes Kepler University Linz, Austria
{hannes.thaller, alexander.egyed}@jku.at
[2] Technical University of Braunschweig, Germany
l.linsbauer@tu-braunschweig.de

Abstract. Semantic clone detection is the process of finding program elements with similar or equal runtime behavior. For example, detecting the semantic equality between the recursive and iterative implementation of the factorial computation. Semantic clone detection is the de facto technical boundary of clone detectors. In recent years, this boundary has been tested using interesting new approaches. This article contributes a semantic clone detection approach that detects clones which have 0 % syntactic similarity. We present Semantic Clone Detection via Probabilistic Software Modeling (SCD-PSM) as a stable and precise solution to semantic clone detection. PSM builds a probabilistic model of a program that is capable of evaluating and generating runtime data. SCD-PSM leverages this model and its model elements for finding behaviorally equal model elements. This behavioral equality is then generalized to semantic equality of the original program elements. It uses the likelihood between model elements as a distance metric. Then, it employs the likelihood ratio significance test to decide whether this distance is significant, given a pre-specified and controllable false-positive rate. The output of SCD-PSM are pairs of program elements (i.e., methods), their distance, and a decision on whether they are clones or not. SCD-PSM yields excellent results with a Matthews Correlation Coefficient greater than 0.9. These results are obtained on classical semantic clone detection problems such as detecting recursive and iterative versions of an algorithm, but also on complex problems used in coding competitions.

Keywords: semantic clone detection · probabilistic software modeling · clone detection

1 Introduction

Copying and pasting source code fragments leads to code clones, which are considered an anti-pattern. Code clones increase maintenance costs [31,32], promote

* The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH. This research was funded in part, by the Austrian Science Fund (FWF) [P25513].

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 288–309, 2022.
https://doi.org/10.1007/978-3-030-99429-7_16

bad software design [29,13,17], and introduce or propagate bugs [4,28,14]. However, duplicating code fragments also allows faster adaptation to requirements, the re-use of stable and well-tested solutions [25,26], and helps to overcome language limitations [21,35], thereby lowering development costs. The impact of code clones and the contradicting evidence various studies provide are the topics of an ongoing discussion in the community. Meanwhile, it is certain that developers will continue duplicating source code to leverage its benefits, despite its drawbacks. The key is the awareness and management of clones to maximize efficiency while balancing quality.

Traditionally, the clone taxonomy distinguishes between four types of clones [35,2,34]. Type 1-3 describe code clones caused by copying and pasting the source code with or without changes. Type 4 clones describe code clones that do not have any syntactic similarity but implement the same functionality (semantic equivalence). For example, the recursive and iterative implementation of an algorithm (e.g., Fibonacci computation) have no syntactic similarity while implementing the same functionality. Existing tools have limited or no capabilities to detect Type 4 clones [19]. Most current studies exclude them because of the lack of tool support [23,35,2,39,11]. Nevertheless, Type 4 clones exist, and recent research efforts have tried to deepen the understanding of them [19,49,20]. This article provides a significant contribution to semantic clone detection in the form of novel concepts and a prototype implementing them.

We present *Semantic Clone Detection via Probabilistic Software Modeling (SCD-PSM)*. SCD-PSM extends our work on Probabilistic Software Modeling (PSM) [43] via a semantic clone detection pipeline. PSM builds Probabilistic Models (PMs) from programs. It analyzes the static structure and dynamic runtime behavior and replicates the program in the form of a generative probabilistic model. These models allow developers to reason about the semantics of a program. SCD-PSM extends this work by leveraging the PMs and causal reasoning to find semantically (i.e., behaviorally) equivalent code elements. SCD-PSM allows full quantification of the behavioral distance of code elements via likelihoods. Furthermore, the likelihood evaluation via PMs allows for statistical significance tests to decide whether a pair of code elements are clones. SCD-PSM detects semantic clones with no textual similarity, such as the iterative and recursive version of an algorithm. The average performance of the approach reaches a Matthews Correlation Coefficient of 0.965 on a complex problem set indicating a robust method for semantic clone detection. This work extends our previous work [41] with a full evaluation and the theoretical foundation.

Section 2 provides the background needed to understand SCD-PSM including the basics of PSM. Section 3 clarifies what semantic clones are in the context of this work. Section 4 presents the approach in which representation, search space, and the various similarity stages are described. Section 5 evaluates the approach while Section 6 discusses the results. Limitations of the approach and possible threats are given in Section 7 and Section 8. Section 9 compares the work to the state-of-art and Section 10 concludes this article.

```
int fa(int n){
  product = 1
  for(i = 1; i <= n i++)
    product *= i
  return product
}
```

Listing 1.1: *for-loop* implementation of factorial

```
int fb(int n){
  product = 1
  i = 1
  while(i <= n)
    product *= i
      i++
  return product
}
```

Listing 1.2: *while-loop* implementation of factorial

```
int fc(int n){
  if(n <= 1) return 1
  return fc(n - 1) * n
}
```

Listing 1.3: *Recursive* implementation of factorial

```
int fd(int n, String guard){
  if(n < 1 && guard == "val")
    return -1
  if(n < 1 && guard == "throw")
    throw Exception()
  return fc(n)
}
```

Listing 1.4: *Delegate* implementation of factorial

2 Background

The clone detection research community has a long history and defines many concepts, algorithms, and tools. In contrast, Probabilistic Software Modeling (PSM) is relatively new and combines software engineering and probabilistic modeling. Some terms need clarification; others require an introduction if they diverge from their traditional names.

2.1 Clone Detection

Clone detection is the process of finding two similar program fragments. Listings 1.1 to 1.4 are four different implementations of the factorial function ($n!$). Listing 1.1 is a *for*-loop implementation, Listing 1.2 uses a *while*-loop, and Listing 1.3 is recursively defined. Finally, Listing 1.4 delegates its implementation to `fc()` from Listing 1.3 but may also return -1 in case of invalid inputs (including $n = 0$).

Representation, pairing, similarity evaluation, and clone decision are the core concepts of clone detection. *Representations* describe on which artifact the detector operates, such as text, graphs (e.g., AST), or probabilistic models. *Pairing* describes the selection of two code fragments that are potentially clones (e.g., `fa()` and `fb()`). Each pair is called a *candidate clone pair* (or candidate pair). The *similarity evaluation* measures the similarity of a candidate pair, e.g., by counting the number of different characters. Finally, the *clone decision* labels the candidate pair as a clone given a criterion on the similarity, e.g., less than ten different characters.

The properties of the similarity metric split clones into two groups [35]. Type 1-3 clones capture textual similarity while Type 4 clones capture semantic similarity [2,23,24,35,34,44]. Type 1 (Exact Clones) clones are program fragments

that are identical except for variations in white-space and comments. Type 2 (Parameterized Clones) clones are program fragments that are structurally or syntactically similar except for changes in identifiers, literals, types, and comments. Type 3 (Near-Miss Clones) clones are program fragments that include insertions or deletions in addition to changes in identifiers, literals, types, and layouts. Type 4 (Semantic Clones) clones are program fragments that are functionally or semantically similar (i.e., perform the same computation) without textual similarities. These types are increasingly challenging to detect, with Type 4 being the most complex one. Note that the definition of *Semantic Clones* is often relaxed, where up-to 50% syntactic similarity of the code fragments is allowed (e.g., BigCloneBench [39]). However, we consider these clones as complex Type 3 clones (additions, deletions, reordering) and *not* as semantic clones. This means that semantic clones in the context of this work are clones with no syntactic similarity except for per-chance similarities.

We will use $a \simeq b$ to denote that a is a clone of b. Furthermore, $a \not\simeq b$ denotes that a is not a clone of b.

2.2 Programs & Code Elements

PSM generalizes object-oriented terms to describe *code elements* in a program. Code elements are *types* $\boldsymbol{T}$, *properties* Pr, and *executables* $\boldsymbol{Ex}$ that refer to, e.g., classes, fields, and methods in Java [1], or classes, properties, and functions in Python [45]. Additional code elements are *parameters* Pr and *results* Re of executables that refer to parameters and return values of a method. Properties, parameters, and results are *atomic* code elements that have identifiable states at runtime. Types and executables are *compositional* elements that act as a collection of atomic elements. Types *declare* properties and executables, capturing structural relationships. Executables have behavioral relationships that are categorized into *Inputs* (I) and *Outputs* (O). *Inputs* are *received parameters* $Pa^{\mathcal{I}}$, *read properties* $Pr^{\mathcal{I}}$, and *requested invocation results* $Re^{\mathcal{I}}$. *Outputs* are *returned executable results* $Re^{\mathcal{O}}$, *written properties* $Pr^{\mathcal{O}}$, and *provided parameters* $Pa^{\mathcal{O}}$. We will denote atomic elements in lowercase, and compositional elements in bold-face lowercase, e.g., n and $\boldsymbol{fa}$ in Listing 1.1. Executable results are named after their executables, e.g., fa in Listing 1.1. $\boldsymbol{fc} = \{n^{Pa,\mathcal{I}}, fc^{Re,\mathcal{I}}, fc^{Re,\mathcal{O}}\}$ denotes the code elements of Listing 1.3. For the sake of readability, we will omit the superscript classifiers if it is unambiguously possible, e.g., $\boldsymbol{fa} = \{n, fa\}$. The subset of *inputs* is denoted by $\boldsymbol{fc}^{\mathcal{I}} = \{n^{Pa,\mathcal{I}}, fc^{Re,\mathcal{I}}\}$ and *outputs* by $\boldsymbol{fc}^{\mathcal{O}} = \{fc^{Re,\mathcal{O}}\}$. Finally, the set of all input and output combinations is given by $bmex^{\mathcal{IO}} = \{(i, o) \in \boldsymbol{ex}^{\mathcal{I}} \times \boldsymbol{ex}^{\mathcal{O}}\}$. For example, $\boldsymbol{fd}^{\mathcal{IO}} = \{(n, fd), (guard, fd)\}$ describes the IO pairs of `fd()`.

2.3 Probabilistic Software Modeling

Probabilistic Software Modeling (PSM) [40] is a data-driven modeling paradigm that transforms a program into a Probabilistic Model (PM). PSM extracts the structure and behavior of a program. Code elements and their dependency graph represent the *structure* as described in Section 2.2. All observable events at

runtime represent the *behavior*. The resulting PM and its model elements are a probabilistic copy of the program.

Model elements in the PM are the equivalent to code elements in the program. $P(x)$ denotes the probability distribution of variable x, e.g., $P_{\boldsymbol{fa}}(n)$ denotes the probability distribution of input parameter n of the `fa`-method. $p(x)$ denotes the probability of a specific event of a variable, e.g., $p_{\boldsymbol{fa}}(n = 2)$. This extends the notation of code elements with probabilistic quantities. However, the notation reasons about the probabilistic behavior of code elements instead of their structural properties.

Each model element is a flow-based latent variable model [7] that learns an invertible mapping between the original observations and an isotropic unit norm Gaussian $\mathcal{N}(0, \mathbf{1})$ with $f : X \mapsto Z$. An example for $x \in X$ may be $n \in \boldsymbol{fa}$ with $n^z \in \boldsymbol{fa}^z$ being its latent Gaussian representation. The Gaussian latent space enables the model elements to generate new samples and evaluate the likelihood of samples.

Generation (or Sampling) draws, either marginally or conditionally, observations from a model element simulating the execution of the corresponding code element. For example, drawing 100 observations from $\boldsymbol{fa} \sim P_{\boldsymbol{fa}}(n, fa)$, i.e., values for $n^{\mathcal{I}}$ and $fa^{\mathcal{O}}$, simulates 100 program executions of this method. An example for *conditional generation* would be $\boldsymbol{fa}_{|n<10} \sim P_{\boldsymbol{fa}}(fa \mid n < 10)$ that only draws observations where $n < 10$. The process involves sampling from the latent Gaussian variables, and inverting the Gaussian samples to the original domain via the flow $f^{-1}(\boldsymbol{z}) = \boldsymbol{x}$. *Evaluation* takes observations and evaluates their likelihood under a model element. For example, $P_{\boldsymbol{fa}}(n = 4, fa = 24)$ evaluates the likelihood of input 4 and output 24 under the fa model element. The process of evaluation involves mapping a given sample into the latent space and evaluating it under the Gaussians $p_{\mathcal{N}(0,\mathbf{1})}(f(\boldsymbol{x}))$. Generation and evaluation are the core of any PSM applications and of SCD-PSM. A detailed description is given in our previous work [43].

3 Semantic Clones

A clear understanding of what SCD-PSM defines a *semantic clone* is essential in understanding the approach and its design choices.

Definition 1. *A semantic clone is a pair of executables whose (partial) input, and output relationships exhibit significant (conditional) similarities.*

Definition 1 defines semantic clones over the similarity between IO relationships of executables. This holds if the IO relationships are only partially similar, i.e., not all combinations of IO pairs between executables have to be similar. For example, fd in Listing 1.4 has two IO pairs ($\boldsymbol{fd}^{\mathcal{IO}} = \{(n, fd), (guard, fd)\}$) while fa in Listing 1.1 has one IO pair ($\boldsymbol{fa}^{\mathcal{IO}} = \{(n, fa))\}$). According to the definition, at least one IO pair comparison needs to be similar such that both executables are declared as a semantic clone (e.g., $(n, fd) \simeq (n, fa)$).

Furthermore, the similarities between IO pairs may only be conditional, i.e., the similarity of matching IO pairs might be depending on the state of any other

code element in the comparison context. For example, the IO pair $(n, fd) \simeq (n, fa)$ is only a perfect clone in case that `fd.guard != "val"`. If `fd.guard == "val"`, the IO behavior would differ in case of $n = 1$ (`fd(1)` $\mapsto -1$ while `fa(1)`$\mapsto 1$). According to the definition, at least parts of the behavior need to be similar, capturing complex multidimensional behavioral patterns in IO relationships.

The rationale behind the comparison of IO relationships is one of cause and effect. If a pair of executables exhibit similar effects given similar causes, then their computational behavior is identical. Extending this rationale by multiple inputs and outputs leads to *partial conditional similarity*.

4 Approach

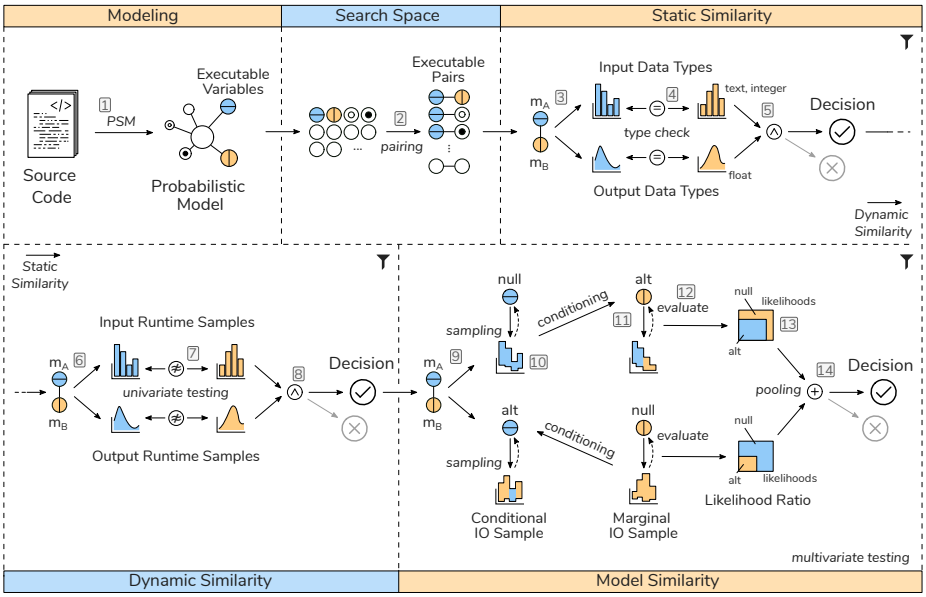


Fig. 1: The modeling phase transforms the program into a PM. The search space phase then pairs the PM model elements into candidate pairs. Finally, Static-, Dynamic- and Model Similarity evaluate the behavioral equality of the candidates.

Figure 1 illustrates SCD-PSM. It is a five-fold approach consisting of the following steps:

A. [**Modeling**] PSM builds a probabilistic model that reflects the original program;
B. [**Search Space**] A search space of candidate pairs is constructed by pairing executable model elements;

C. [**Static Similarity**] The static similarity stage accepts candidate pairs with matching data types;
D. [**Dynamic Similarity**] The dynamic similarity stage accepts candidate pairs with similar runtime data;
E. [**Model Similarity**] The model similarity stage accepts candidate pairs with similar model behavior.

The approach represents a rejecting filter pipeline that candidate pairs must traverse in order to be declared a clone. Static-, Dynamic-, and Model Similarity represent filter stages of increasing complexity.

The main contribution of this work is the implementation of a semantic clone detection pipeline on top of PSM. Further, we provide an effective process of traversing the potentially large search space of candidate pairs. Finally, we show that the behavioral equivalence of model elements generalizes to the semantic equivalence of code elements.

4.1 Modeling

Starting from the *Source Code* in Figure 1, PSM builds a *Probabilistic Model* (PM) [40] of the program (1). The PM is also called the Inference Graph (IG), which is a cluster graph [22] with Non-Volume Preserving Flows (NVPs) [7] as clusters. SCD-PSM uses this PM as a representation for the clone detection, similar to text-based clone detectors that use text fragments. The PM is the output of PSM and is considered as given in the context of SCD-PSM.

Executable model elements in the PM act as a surrogate to the executables in the program. SCD-PSM pairs these model elements and computes their similarity. If a behaviorally equivalent model element pair is found, then it can be seen as a semantically equivalent code element pair. In conclusion, the SCD-PSM allows for method-level semantic clone detection based on PMs representing the original executables in the program.

4.2 Search Space

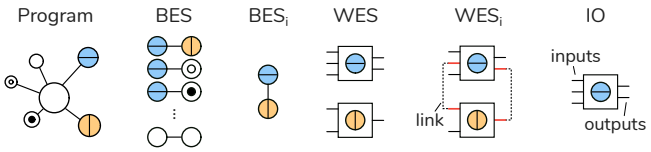


Fig. 2: SCD-PSM operates on four levels of abstraction: program, between executable, within executable, and the IO level.

SCD-PSM conducts method-level semantic clone detection, which operates on multiple abstraction levels. Figure 2 illustrates these levels, starting with the program and ending with the inputs and outputs of an executable.

The second step in Figure 1 builds a *within- and between-executable space* that SCD-PSM searches for clones. The *Between-Executable Space (BES)* is the set of executable combinations

$$\boldsymbol{BES} = \{\{\boldsymbol{a}, \boldsymbol{b}\} \in \boldsymbol{Ex} \times \boldsymbol{Ex} \mid \boldsymbol{a} \neq \boldsymbol{b}\}, \tag{1}$$

where $\boldsymbol{exa}, \boldsymbol{exb}$ is a *candidate pair* (or executable pair), and $\boldsymbol{Ex}$ is the set of all executables in the current analysis (illustrated in Figure 2). The theoretical size of the between-executable space are all 2-length combinations without replacement, given by

$$|\boldsymbol{BES}| = \frac{|\boldsymbol{Ex}|!}{2 \cdot (|\boldsymbol{Ex}| - 2)!}, \tag{2}$$

where $|\cdot|$ describes the size of the underlying set. Note that the size of the BES is smaller than the Cartesian product since $\{a, b\} = \{b, a\}$. Figure 1 shows this pairing process in the Search Space aspect (2) from Figure 1. The *Within-Executable Space (WES)* is the product of IO pairs

$$\boldsymbol{WES}^{ab} = \{(i, j) \in \boldsymbol{a}^{\mathcal{IO}} \times \boldsymbol{b}^{\mathcal{IO}}\}. \tag{3}$$

Figure 2 illustrates the WES and one IO pair from the WES that we also call *link*. The theoretical size of the within-executable space is

$$\left|\boldsymbol{WES}^{ab}\right| = \left|\boldsymbol{a}^{\mathcal{IO}}\right| \cdot \left|\boldsymbol{b}^{\mathcal{IO}}\right| \tag{4}$$

For the sake of visualization, IO pairs are not shown in Figure 1 but are abstracted in their executable elements. The maximum theoretical search space is

$$S = \sum_i \left|wes(\boldsymbol{BES}_i)\right|, \tag{5}$$

given that *wes* describes a construction function according to Equation (3), and $\boldsymbol{BES}_i$ is the i'th candidate pair.

In practice, SCD-PSM evaluates only a fraction of possible combinations because of the skip evaluation. The *skip evaluation* consists of two search space limiting factors: greedy evaluation and transitive similarity. *Greedy evaluation* stops the search through the WES once a similar pair is found. The initial detection process only confirms the similarity of a candidate pair. A post-analysis can then extract all possible IO similarities for potential actions. *Transitive similarity* skips evaluations in the BES, because of $a \simeq b \simeq c$ then also $a \simeq c$ holds. In conclusion, SCD-PSM compares IO pairs of executable model elements and uses skip evaluation to traverse the search space efficiently.

4.3 Static Similarity

The static similarity stage is a filter that accepts candidate pairs based on their data type, as shown in Figure 1. Data types in a PSM model are integers, floats, and text.

Input (3) of the stage are the IO pairs $\boldsymbol{WES}^{ab} = wes(\{\boldsymbol{a}, \boldsymbol{b}\})$ of a candidate. The filter *criteria* (4) accepts a candidate pair if at least *one* link (i.e., IO pair) has a matching data type, i.e., the input but also the output have a matching data type. *Output* (5) is a boolean decision whether the candidate pair is a clone or not from a static viewpoint. If positive, then the candidate pair is moved to the next pipeline stage, i.e., the *Dynamic Similarity* evaluation (see Figure 1). If negative, then the candidate pair is marked as being *not* a clone $\boldsymbol{a} \not\simeq \boldsymbol{b}$ and no further processing is conducted. For example, the IO pairs $(n, fa) \simeq (n, fb)$ would be statically accepted as clones as both inputs and outputs have the same data type (integer). A counterexample is given by $(n, fa) \simeq (guard, fd)$ where the input data types are integers and text.

The static similarity indicates that the analyzed program is given in a programming language that allows for static analysis. Programs written in programming languages without static typing can not make use of this filter stage. In conclusion, the static similarity stage filters candidates based on their data type.

4.4 Dynamic Similarity

The dynamic similarity stage is a filter that accepts candidate pairs based on the runtime data, as shown in Figure 1. Candidates pairs are accepted if at least *one* IO pair (6) has an *insignificant* diverging runtime distribution (7). This boolean decision is evaluated via a Kolmogorov-Smirnov test [30], and determines whether a pair is a clone from a dynamic viewpoint (8). For example, the IO pair $(n, fa) \simeq (n, fd)$ with `guard == true` would be excluded form the filter given that runtime events with $n = 0$ reach a majority. In comparison, $(n, fa) \simeq (n, fb)$ would be accepted by the stage.

A requirement is that the candidates use a synthetic trigger. Otherwise, the comparison of the data distributions may fail because of the different modus operandi of the program. For example, running `fa` and `fb` where $n_{fa} = \mathcal{U}(0, 4)$ and $n_{fb} = \mathcal{U}(5, 10)$ would cause the dynamic stage to fail even if the implementations are equivalent. Property-based [12] or random testing can be used to generate diverse synthetic inputs.

In conclusion, the dynamic similarity stage pre-filters candidates based on univariate tests on the input and output events.

4.5 Model Similarity

The model similarity stage is a filter that accepts candidate pairs based on the models, as shown in Figure 1. This stage conducts a multivariate test by sampling from the executable models and cross evaluating them. This test includes the evaluation of conditional influences caused by elements that are not actively participating in an IO pair. For example, $(n, fd) \simeq (n, fa)$ holds but is conditionally dependent on *guard*. The model similarity can factor *guard* into its decision while the dynamic stage can only evaluate the average behavior of an IO pair.

Input (9) are the IO pairs of a candidate $\boldsymbol{WES}^{ab} = wes(\{\boldsymbol{a}, \boldsymbol{b}\})$. The crosswise log-likelihood ratio of the models is computed by *(conditional) generation*

and *evaluation*. *Output* is a boolean decision on whether the candidate pair is a clone or not, from a model viewpoint. Figure 1 illustrates the entire process of the model similarity.

(A) A reference $M^{null} = \boldsymbol{a}$ and an alternative model $M^{alt} = \boldsymbol{b}$ is selected.
(B) An IO-pair $p = \boldsymbol{WES}_i^{ab}$ is selected as the target of the comparison (link).
(C) A reference sample D^{null} is drawn from M^{null} (10).
(D) An alternative sample $D^{alt|null}$ is drawn from M^{alt} by optimizing towards the p dimensions in the D^{null}, effectively conditioning the drawn samples (11).
(E) D^{null} is evaluated under M^{null} resulting the reference log-likelihood LL^{null}
(F) $D^{alt|null}$ is evaluated under M^{alt} (12) yielding the alternative log-likelihood LL^{null}.
(G) Finally, the likelihood ratio of the link is computed by $\lambda = LL^{alt} - LL^{null}$

The roles between the *null* and *alt* models are then swapped, and the process is repeated. Both log-likelihood ratios are then combined by a pooling operator to produce the clone decision (14).

The role-swap is needed to avoid sub-model relationships. For example, if $M^{null} = \mathcal{N}(0, 3)$ and $M^{alt} = \mathcal{N}(0, 1)$ then LL^{alt} will be very high because M^{alt} is a sub-model from M^{null}. Reversing the roles highlights the differences in the models.

The final decision is based on the Generalized Likelihood Ratio Test (GLRT) [10]. It measures whether the log-likelihoods are significantly different from 0, where λ is the test statistic. The null hypothesis is rejected for small ratios $\lambda \leq c$ where c is set to an appropriate false-positive rate. For example, $\lambda < \log(0.01)$ allows 1 out of 100 candidates to be a false-positive, i.e., wrongly rejecting semantic equivalence. The pooling operator combines the link results either via hard or soft pooling. *Hard pooling* conducts for both links a GLRT yielding a positive decision if *both* links are positive. *Soft pooling* averages the link log-likelihoods ratios and then computes the GLRT yielding a positive decision if the joint GLRT is positive. Hard pooling does not allow any sub-model relationships, while soft pooling relaxes this constraint.

In conclusion, the model similarity conducts a multivariate significance test between two models, including possible conditional dependencies.

5 Study

This study answers the following research questions.

Q1 Does behavioral equality between model elements generalize to semantic equality of code elements?
Q2 Does the skip evaluation significantly reduce the computational demand of SCD-PSM?
Q3 Does the skip evaluation negatively impact the detection performance (i.e., precision, recall, and MCC)?

Q1 answers whether semantic clones can be detected via SCD-PSM. Q2 answers whether the search space can be efficiently processed using skip evaluation. Q3 answers how the skip evaluation influences the performance of the detection process. This is important because candidate pairs might be skipped based on false-positives or false-negatives.

5.1 Setup

We implemented a prototype for SCD-PSM on top of Gradient [40], a prototype for PSM. The elements and data flow of the detection process are shown in Figures 1 and 2.

1. The input *Source Code* were 13 different clone classes with a total of 108 implementation variants. This includes classical algorithms implemented recursively and iteratively such as bubble sort, as well as hard problems from the programming competition Google Code Jam[1].
2. The *Probabilistic Model* was computed via Gradient, a PSM prototype. We used the same hyper-parameters as reported in our previous work [43].
3. The *Search Space*, i.e., the *BES* and *WES*, was created according to Section 4.2 based on *all* available examples.
4. Each valid candidate pair was then submitted to the *Static-, Dynamic*, - and *Model-Similarity* stages and filtered according to Sections 4.3 to 4.5. Candidates that passed the entire filter pipeline were marked as clones.

5.2 Dataset

The study uses three well-known algorithms and 10 Google Code Jam 2017 (GCJ)[1] problems. The total dataset contains 108 implementation variants across 13 clone classes described by *Instance*.

Each clone class was differentially tested to verify the behavior across instances. Factorial, Fibonacci, and Sort do not need any further explanation. The GCJ problems are well specified complex optimization problems packaged in an everyday theme.

The dataset contains in total 5778 (see Equation (2)) candidate pairs of which 458 are semantic clones and 5320 are not. This yields a positive to negative ratio of $1 : 11.6$, indicating a highly imbalanced distribution. An even more pronounced imbalance is to be expected in real-world applications.

Each instance was triggered with input data to allow PSM to model the different implementations. Factorial, Fibonacci, and Sort were triggered by sampling from a uniform distribution $\mathcal{U}(0, 20)$. GCJ problems were triggered by the input data provided by the competition. Each instance received the same trigger.

GCJ problems read from and write to the standard stream, which is impractical in terms of reproducibility. Our dataset is constructed such that each implementation has a `run`-method representing the cloned executable. The study results are limited to the `run`-method even if the solutions use helper methods.

[1] https://codingcompetitions.withgoogle.com/codejam/archive

Helper methods may, for example, be methods that compute parts of the final solution, or reorganize the data. This guarantees a proper problem scope, a well-defined recall and precision, and a clearly defined benchmark for future reproducibility.

5.3 Controlled Variables

The study controls for the search space *Evaluation* strategy, *Dynamic False-Positive Rate (D-FPR)*, *Model False-Positive Rate (M-FPR)*, and *Pooling*.

Evaluation describes how the search space is processed: *exhaustive*, or *skip*. The exhaustive evaluation compares each executable candidate with each other. The skip evaluation uses the transitive similarity (see Section 4.2) and may skip evaluations if possible.

Dynamic False-Positive Rate (D-FPR) defines the critical value α of the Kolmogorov-Smirnov test with 0.001 and 0.01, at which similarity is rejected.

Model False-Positive Rate (M-FPR) defines the critical value c of the Generalized Likelihood Ratio test with 0.001 and 0.01, at which similarity is rejected.

Pooling defines how the likelihood ratios from the two link directions are combined (see Figure 1, (8)) with values: *hard*, or *soft*. *Hard* pooling evaluates whether each link reaches the critical value c and accepts the clone if both links evaluate as positive with $\lambda_{Link_A} \leq \frac{\log c}{2}$ and $\lambda_{Link_B} \leq \frac{\log c}{2}$. *Soft* pooling evaluates the average log-likelihood ratios (geometric mean of likelihoods) $\frac{\lambda_{Link_A} + \lambda_{Link_B}}{2} \leq \log c$, and compares it against the critical value c.

An additional fixed parameter is the *number of particles*. It defines the sample size that is generated during the model similarity $|D| = 50$.

5.4 Response Variables

The response measures of the study are the number of *Skip Evaluations*, processing *Duration*, *TP, FP, TN, FN*, *Precision*, *Recall*, *F1*, and *Matthews Correlation Coefficient*.

Skip Evaluations measures the number of evaluations that were skipped due to the skip evaluation strategy.

Duration measures the elapsed time to compute one candidate pair.

TP, FP, TN, FN measures the True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) detection results compared to the ground truth.

Precision measures the fraction of detected clones that are truly clones.

Recall measures the fraction of semantic clones that have been found.

F1 measures the accuracy of a binary classification as the harmonic mean of recall and precision.

Table 1: Results of the top-5 and bottom-1 experiment along with the average performance of the top-5.

Controlled Variables					Response Variables									
Nr	Evaluation	D-FPR	M-FPR	Pooling	Duration	TP	FP	TN	FN	Skip	Precision	Recall	F1	**MCC**
1	skip	0.100	0.001	soft	1560	437	0	5320	21	345	1.000	0.954	0.977	**0.975**
2	skip	0.010	0.001	soft	1620	437	0	5320	21	345	1.000	0.954	0.977	**0.975**
3	exhaustive	0.010	0.001	soft	1680	425	0	5320	33	0	1.000	0.928	0.963	**0.960**
4	skip	0.010	0.010	soft	1920	423	0	5320	35	332	1.000	0.924	0.960	**0.958**
5	exhaustive	0.100	0.001	soft	2040	421	0	5320	37	0	1.000	0.919	0.958	**0.955**
16	exhaustive	0.100	0.010	hard	2820	293	0	5320	165	0	1.000	0.639	0.780	**0.787**
1-5	skip	0.010	0.001	soft	1740	428	0	5320	29	340	1.000	0.936	0.967	**0.965**

Duration in seconds

Matthews Correlation Coefficient (MCC) measures the quality of the clone detection in the form of a correlation ranging from -1 to 1, with 0 being a random selection. The MCC will be the reference performance metric as it is the most robust metric in an imbalanced binary classification setting [3]. It is a correlation coefficient which may be interpreted by the guidelines proposed by Evans [9].

5.5 Comparison of Clone Detectors

In total, eight alternative approaches are used to contextualize the performance of SCD-PSM. The alternatives have a wide variety in terms of internal representation and clone detection capabilities as listed in Table 3. ASTNN (8) and ASTNN Leaky (9) are the same approach but have different evaluation methods. ASTNN Leaky (9) uses a random split of the dataset as reported by the authors [50]. It overestimates the performance of the approach via a lack of isolation between training and test dataset. For example, $fa \simeq fb$ and $fa \simeq fc$ might be in the train split while $fb \simeq fc$ might be in the test split. ASTNN (8) uses a group-wise Cross Validation (CV), where clone classes are entirely isolated either into the training or test proportion of the dataset. This represents a real-world situation where first the detector is fitted and then applied to a new system with unknown code fragments.

Detectors that report lines instead of methods may produce more results (TP, FP, TN, FN) than present in the dataset. A similar situation is given by ASTNN Leaky that runs multiple evaluations via the cross validation.

5.6 Experiment Results

Creating the PSM model with Gradient took 2134.38 s, resulting in an average modeling time of 19.75 s for the 195 executables. This includes 87 helper methods.

Table 1 contains the aggregate results of the top-5 experiments along with the results of the worst experiment. The bottom line in Table 1 is the average

Table 2: Performance breakdown of the best performing experiment listed as Nr. 1 in Table 1.

Stage	Duration	TP	FP	TN	FN	Precision	Recall	F1	**MCC**
initial	–	458	5320	0	0	0.079	1.000	0.147	
static	0.0001	458	1504	3816	0	0.233	1.000	0.379	**0.409**
dynamic	0.208	451	50	5270	7	0.900	0.985	0.941	**0.936**
model	1.749	437	0	5320	21	1.000	0.954	0.977	**0.975**
	0.344	437	0	5320	21	0.996	0.954	0.977	**0.975**

Duration in seconds

performance of the top-5 experiments. The generally expected performance of the approach is *very strong* with an MCC of 0.965. High confidence for negative examples is given with no false-positives reflecting the pipeline's FPR rates (D-FPR × M-FPR). The best experiment featured a *skip evaluation*, *0.100* D-FPR and *0.001* M-FPR rates, and *soft pooling* (Nr. 1) with an MCC of 0.975. The worst experiment featured a *exhaustive evaluation*, D-FPR of 0.100, M-FPR of 0.010, and *hard pooling* (Nr.16) with a *strong* MCC of 0.787. A total of 345 candidates were skipped while reaching a recall of 0.933.

Table 2 lists the cumulative performance of the best model, starting with an initial prediction that all candidates are semantic clones (rejecting pipeline). The *static* stage finds 71.729 % (3816) of the FPs, improving the MCC by 0.409. The *dynamic* stage additionally removes another 27.330 % (1454) of FPs but introduces 1.528 % (7) of the possible FNs. An improvement of the MCC by 0.527 is achieved via the dynamic stage. Finally, the *model* stage removes the remaining 0.939 % (50) FPs but introduces additional 3.056 % (14) additional FNs. The model stage improves the MCC by 0.039.

On average, 5.884 % (340) of the total 5778 evaluations could be skipped. This equals 74.235 % of the total 458 TPs. On average 37.359 % (50 354) of the total 134 782 IO pair evaluations could be saved via greedy evaluation. The average duration of the exhaustive experiments was 2394 s, leading to 414 ms per candidate. Skip experiments lasted on average 1988 s with 344 ms per candidate. The static stage lasted on average for <0.001 % of the time per candidate (see Table 2), the dynamic stage for 0.106 %, and the model stage for 0.893 %.

Table 3 lists the detection results of eight alternative clone detectors. Simian, NiCad, and CCAligner found no clones in the dataset. PMD, SourcererCC, Oreo, and iClones found some clones (< 20) with a low recall (4 %). Each of these detectors has a *very weak* performance below an MCC of 0.20 ASTNN with the leaky evaluation has a *very strong* performance with an MCC of 0.976. ASTNN 3-Group CV has a *strong* performance with an MCC of 0.711. The longest computational duration is given by ASTNN with 1034 min.

Table 3: Detection results of other clone detectors on the dataset.

Nr	Tool	Note	Repr.	Type	Duration	TP	FP	TN	FN	Precision	Recall	F1	**MCC**
1	Simian [16]		Text	1	0.138	0	0	5320	458		0.000		
2	NiCad [5]		Text	3	1.291	0	0	5320	458		0.000		
3	CCAligner [47]		Token	3	1.109	0	4	5316	458	0.000	0.000		**-0.007**
4	PMD [33]		Token	2	1.389	8	12	5308	450	0.400	0.017	0.033	**0.069**
5	SourcererCC [37]		Token	3/4	36.86	10	0	5320	448	1.000	0.021	0.042	**0.142**
6	Oreo [36]		Model	3/4	79.00	17	5	5315	441	0.772	0.037	0.070	**0.158**
7	iClones [15]		Token	3/4	0.980	13	0	5320	445	1.000	0.028	0.055	**0.161**
8	ASTNN [50]	3-Group CV	Model	4	1034	296	29	1415	162	0.911	0.646	0.756	**0.711**
9	ASTNN (Leaky)	Random Split	Model	4	2028	442	4	5316	16	0.991	0.965	0.978	**0.976**
10	SCD-PSM	Top 1-5	Model	4	1740	428	0	5320	29	1.000	0.936	0.967	**0.965**

Duration in seconds

6 Discussion

The goal of the study was to provide evidence of whether behavioral equality of model elements generalizes to semantic equality of code elements (Q1). Furthermore, we were interested in the skip evaluation and its performance implications (Q2 and Q3).

6.1 Research Question 1 — Detection Performance

Table 1 and Table 2 present strong results in favor of Q1. The MCC for the top-5 experiments was *very strong* with all MCCs being above 0.9. Even the worst experiment still yielded a *moderate* performance of 0.749.

Table 3 provides additional context to the results by presenting the detection results of alternative clone detectors. As expected, tools relying heavily on the textual representation of clones have very low recall (Simian, NiCad, CCAligner, PMD) on the dataset. Most clones found by the alternative tools span only a few lines of code. In contrast, iClones finds large clones that include array accesses and manipulations. ASTNN is the best comparison tool and finds many clones with good precision. The approach is sensitive to hyper-parameters and to the training and test split, leading in some cases to a test performance close to MCC of 0. The low recall for Type 1-3 detectors indicate the high quality of the dataset. The moderate recall for Type 3/4 detectors indicate the high quality of SCD-PSM. Given this evidence, we conclude that Q1 holds.

> Q1 — Behavioral equality between model elements generalizes to semantic equality of code elements, allowing for semantic clone detection via probabilistic software modeling.

6.2 Research Question 2 — Skip Evaluation Scalability

The goal of the static and dynamic stage is to reduce the number of evaluations that the model stage must conduct. Each stage incurs an increasing cost of evaluation per candidate, with the model stage taking the largest share of the evaluation time, 89 %. Every TP has to pass the model stage to be declared a clone (rejecting pipeline). The skip evaluation avoided, on average, the re-computation of 74 % (340) of the TP candidate pairs. The greedy evaluation avoided, on average, the evaluation of 37 % of IO pairs. This offloads most of the evaluation time to the earlier stages, which are computationally inexpensive, while shortcutting the model stage. In comparison to the alternative detectors, SCD-PSM needs substantially more time to compute (1.32 min vs. 29 min). An exception is ASTNN which has a similar runtime as SCD-PSM. Most of the runtime of SCD-PSM is caused by the operational overhead, e.g., loading the model from the database. Optimizing this overhead, as a theoretical maximum, could reduce the overall runtime on the dataset to 6.49 min given the average durations for each stage in Table 2. In conclusion, the skip evaluation reduces the number of model evaluations, which are responsible for most of the evaluation time, down to a quarter.

> Q2 — Skip evaluation reduces the number of evaluations for the most expensive stage (model) in the SCD-PSM pipeline significantly.

6.3 Research Question 3 — Skip Evaluation Effects

Skip evaluation can cause cascading errors given an FP. Once an FP is introduced, every semantic clone related to the FP has a chance to become an FP in the same (wrong) clone class itself. These cascading FPs are potential sources of serious performance degradation. Skip evaluation experiments are ranked higher and are significantly better than experiments that conducted an exhaustive search. However, the absolute performance gain is only a MCC of 0.056, hinting at a per-chance significance introduced by the small sample size (16 experiments). Nevertheless, given the evidence in Table 1 and Section 5.6, we can conclude that skip evaluation does not affect the performance of the detector.

> Q3 — The skip evaluation has no negative impact on the performance of the detector given low false-positive rates.

7 Limitations

SCD-PSM inherits the limitations of PSM, such as its need for a runnable program to build the model. PSM only models the application structure and its data, not references. References are changing addresses with no relation to the running

program. Hence, they have no meaningful underlying distribution that can be modeled. However, once references are dereferenced, e.g., by accessing a field, their accessed data will be part of the model and therefore usable in SCD-PSM. Nevertheless, algorithms with the sole purpose of manipulating references do not work with SCD-PSM.

PSM explodes lists into singular values, since distributions do not contain any order information. This means executables that change the order of sequences are matched based on the values, not their order. As a consequence, an ascending and descending sorting algorithm are semantically equivalent, leading to a false-positive. Extending PSM to distributions of sequences alleviates the issue but is not a trivial task.

SCD-PSM cannot detect Type 2-3 clones since textual similarities represent a different problem set. A proof can easily be constructed by adding an arbitrary number of statements that do not influence the behavior of the program but mislead text based detectors. Inversely, changing one character, e.g., a multiplication to a division, may alter the entire behavior while preserving the general textual similarity.

We employed a controlled laboratory evaluation strategy that allowed us to exactly evaluate the performance metrics and fairly compare them between different clone detectors. This follows a recent trend [38,46,48] in the light of some criticism of opportunistic evaluations on arbitrary open source projects. The controlled laboratory evaluation provides purely functional performance results given a fixed and controlled sample of programs. The generalizability of results obtained from laboratory evaluations is limited; Using an opportunistic evaluation strategy avoids this problem. However, the strategy is prone to biases caused by the human oracles (often the authors themselves) or proxy oracles that evaluate the clones. Moreover, a fair comparison between detectors is hardly possible because the true recall of clones is in general unknown. A combination of both evaluation strategies may yield precise and generalizable results. The extension to this study is part of our future work.

8 Threats to Validity

A threat to validity in any semantic clone detection study is given by the programs and code fragments used in the evaluation. Semantic clones may not exhibit the same functional behavior or share too many lexicographical similarities. This study tested every clone class on its behavioral equality. Furthermore, we evaluated text-, token-, graph- and model-based detectors capable of detecting Type 1-3 clones. The low performance of Type 1-3 detectors confirmed the high quality of semantic clones in the benchmark.

9 Related Work

We started this article by defining what *semantic clones* means in the context of our approach (Section 3). While our definition is motivated by the capabilities of our approach, we can see strong similarities to the definition of Juergens

[19]. Both definitions define behavioral similarity via IO relationships. Also, Juergens already discussed a notion of partial and conditional similarity. This understanding of Type 4 clones can be seen in multiple more recent studies [8,6,27]. In that, we see the progress of the community in terms of Type 4 clones as the definition becomes more specific.

Many studies evaluated textual clones. However, only a few studies have reported results on semantic clones without relaxing the definition of Type 4. Rattan [34] et al. provided a review of clone detection studies including approaches focused on Type 4 clones. They concluded that some approaches solve approximations (i.e., complex Type 3 clones) of Type 4 clones.

Test-based methods randomly trigger the execution of candidates and measure whether equal inputs cause similar outputs. Jiang and Su [18] were able to find semantically equivalent methods without any syntactic similarities. A similar approach was presented by Deissenboeck et al. [6]. One issue with test-based clone detection is that candidates need a similar signature. Differences in data types or the number of parameters can not be effectively handled. SCD-PSM works similarly to test-based methods in that it observes the runtime and compares the resulting behavior. However, SCD-PSM builds generative models from the observed behavior, capable of generating, conditioning, and evaluating data. This allows SCD-PSM to bridge signature mismatches by imputing missing code elements and the using a generalized type system.

Zhao and Huang [51] developed DeepSim, which phrases the problem as a binary classification task. DeepSim uses neural networks to learn encodings of the control and data flow without observing the program's runtime. PSM also uses neural networks but learns an underlying representation of the data flow and runtime. DeepSim was also evaluated on a Google Code Jam dataset. It reached an F1 score of 0.76 on the GCJ 2016 competition, while SCD-PSM reached 0.967 on the GCJ 2017. While not entirely comparable, the results are a good approximation given the similarity in the datasets.

10 Conclusions and Future Work

In this article, we presented Semantic Clone Detection via Probabilistic Software Modeling (SCD-PSM). PSM builds a Probabilistic Model (PM) from a program that can be used to simulate or evaluate a program. We used these PMs to detect semantic clones in programs that have 0 % syntactic similarity.

We discussed the representation, search space, static-, dynamic-, and model-similarity stages forming the main aspects of SCD-PSM. The study evaluated SCD-PSM in great detail resulting in an average MCC greater than 0.9. Also, the study showed the capability to control the false-positive rate, which is important for an industry adoption. Finally, we concluded that behavioral equality of model elements generalizes to semantic equality of code elements.

Our future work focuses on constructing a comprehensive benchmark covering controlled and real-world systems for improved generalizability of clone detection studies. Furthermore, semantic clone detection has the potential to enable new methods for fault localization applications [42].

References

1. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edn. (2000)
2. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and Evaluation of Clone Detection Tools. IEEE Transactions on Software Engineering **33**(9), 577–591 (2007). https://doi.org/10.1109/TSE.2007.70725
3. Boughorbel, S., Jarray, F., El-Anbari, M.: Optimal classifier for imbalanced data using matthews correlation coefficient metric. PloS one **12**(6), e0177678 (2017)
4. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. ACM SIGOPS Operating Systems Review **35**(5), 73 (Dec 2001). https://doi.org/10.1145/502059.502042
5. Cordy, J.R., Roy, C.K.: The NiCad Clone Detector. In: 2011 IEEE 19th International Conference on Program Comprehension. p. 219–220 (Jun 2011). https://doi.org/10.1109/ICPC.2011.26
6. Deissenboeck, F., Heinemann, L., Hummel, B., Wagner, S.: Challenges of the Dynamic Detection of Functionally Similar Code Fragments. In: 2012 16th European Conference on Software Maintenance and Reengineering. p. 299–308 (Mar 2012). https://doi.org/10.1109/CSMR.2012.38
7. Dinh, L., Sohl-Dickstein, J., Bengio, S.: Density estimation using Real NVP. arXiv:1605.08803 [cs, stat] (May 2016)
8. Elva, R., Leavens, G.T.: JSCTracker : A Semantic Clone Detection Tool for Java Code (2012)
9. Evans, J.D.: Straightforward Statistics for the Behavioral Sciences. Brooks/Cole Pub. Co, Pacific Grove (1996)
10. Fan, J., Zhang, C., Zhang, J.: Generalized Likelihood Ratio Statistics and Wilks Phenomenon. The Annals of Statistics **29**(1), 153–193 (2001)
11. Farmahinifarahani, F., Saini, V., Yang, D., Sajnani, H., Lopes, C.V.: On Precision of Code Clone Detection Tools. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). p. 84–94 (Feb 2019). https://doi.org/10.1109/SANER.2019.8668015
12. Fink, G., Bishop, M.: Property-based testing: A new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes **22**(4), 74–80 (Jul 1997). https://doi.org/10.1145/263244.263267
13. Fowler, M., Beck, K.: Refactoring: Improving the Design of Existing Code. The Addison-Wesley Object Technology Series, Addison-Wesley, Reading, MA (1999)
14. Geiger, R., Fluri, B., Gall, H.C., Pinzger, M.: Relation of Code Clones and Change Couplings. In: Baresi, L., Heckel, R. (eds.) Fundamental Approaches to Software Engineering, vol. 3922, p. 411–425. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11693017_31
15. Göde, N., Koschke, R.: Incremental Clone Detection. In: 2009 13th European Conference on Software Maintenance and Reengineering. p. 219–228 (Mar 2009). https://doi.org/10.1109/CSMR.2009.20
16. Harris, S.: Simian - Similarity Analyser (2003)
17. Hunt, A., Thomas, D.: The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, Reading, Mass (2000)
18. Jiang, L., Su, Z.: Automatic Mining of Functionally Equivalent Code Fragments via Random Testing. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. p. 81–92. ISSTA '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1572272.1572283

19. Juergens, E., Deissenboeck, F., Hummel, B.: Code Similarities Beyond Copy & Paste. In: 2010 14th European Conference on Software Maintenance and Reengineering. p. 78–87. IEEE, Madrid (Mar 2010). https://doi.org/10.1109/CSMR.2010.33
20. Kafer, V., Wagner, S., Koschke, R.: Are there functionally similar code clones in practice? In: 2018 IEEE 12th International Workshop on Software Clones (IWSC). p. 2–8. IEEE, Campobasso (Mar 2018). https://doi.org/10.1109/IWSC.2018.8327312
21. Kapser, C.J., Godfrey, M.W.: "Cloning considered harmful" considered harmful: Patterns of cloning in software. Empirical Software Engineering **13**(6), 645–692 (Dec 2008). https://doi.org/10.1007/s10664-008-9076-6
22. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. Adaptive Computation and Machine Learning, MIT Press, Cambridge, MA (2009)
23. Koschke, R.: Survey of research on software clones. In: Koschke, R., Merlo, E., Walenstein, A. (eds.) Duplication, Redundancy, and Similarity in Software. No. 06301 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany (2007)
24. Krinke, J.: Identifying Similar Code with Program Dependence Graphs. Proceedings Eighth Working Conference on Reverse Engineering p. 301–309 (2001). https://doi.org/10.1109/WCRE.2001.957835
25. Krinke, J.: Is Cloned Code More Stable than Non-Cloned Code? Proceedings - 8th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2008 p. 57–66 (2008). https://doi.org/10.1109/SCAM.2008.14
26. Krinke, J.: Is Cloned Code Older than Non-Cloned Code? (2011)
27. Li, G., Liu, H., Jiang, Y., Jin, J.: Test-Based Clone Detection: An Initial Try on Semantically Equivalent Methods. IEEE Access **6**, 77643–77655 (2018). https://doi.org/10.1109/ACCESS.2018.2883699
28. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Transactions on Software Engineering **32**(3), 176–192 (2006). https://doi.org/10.1109/TSE.2006.28
29. Martin, R.C. (ed.): Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, Upper Saddle River, NJ (2009)
30. Massey, F.J.: The Kolmogorov-Smirnov Test for Goodness of Fit. Journal of the American Statistical Association **46**(253), 68–78 (Mar 1951). https://doi.org/10.1080/01621459.1951.10500769
31. Mayrand, Leblanc, Merlo: Experiment on the automatic detection of function clones in a software system using metrics. In: Proceedings of International Conference on Software Maintenance ICSM-96. p. 244–253. IEEE, Monterey, CA, USA (1996). https://doi.org/10.1109/ICSM.1996.565012
32. Monden, A., Nakae, D., Kamiya, T., Sato, S., Matsumoto, K.: Software quality analysis by code clones in industrial legacy software. In: Proceedings Eighth IEEE Symposium on Software Metrics. p. 87–94. IEEE Comput. Soc, Ottawa, Ont., Canada (2002). https://doi.org/10.1109/METRIC.2002.1011328
33. PMD: Pmd. PMD (2019)
34. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: A systematic review. Information and Software Technology **55**(7), 1165–1199 (Jul 2013). https://doi.org/10.1016/j.infsof.2013.01.008
35. Roy, C.K., Cordy, J.R.: A Survey on Software Clone Detection Research. Queen's School of Computing TR **115**, 115 (2007)
36. Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., Lopes, C.V.: Oreo: Detection of clones in the twilight zone. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018. p. 354–365. ACM Press, Lake Buena Vista, FL, USA (2018). https://doi.org/10.1145/3236024.3236026

37. Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V.: Sourcerercc: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering. p. 1157–1168 (2016)
38. Su, F.H., Bell, J., Harvey, K., Sethumadhavan, S., Kaiser, G., Jebara, T.: Code relatives: detecting similarly behaving software. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016. ACM Press (2016). https://doi.org/10.1145/2950290.2950321
39. Svajlenko, J., Roy, C.K.: Evaluating clone detection tools with BigCloneBench. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). p. 131–140. IEEE, Bremen, Germany (Sep 2015). https://doi.org/10.1109/ICSM.2015.7332459
40. Thaller, H., Linsbauer, L., Egyed, A.: Feature Maps: A Comprehensible Software Representation for Design Pattern Detection. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). p. 207–217. IEEE, Hangzhou, China (Feb 2019). https://doi.org/10.1109/SANER.2019.8667978
41. Thaller, H., Linsbauer, L., Egyed, A.: Towards Semantic Clone Detection via Probabilistic Software Modeling. In: 2020 IEEE 14th International Workshop on Software Clones (IWSC). p. 64–69. IEEE (2020)
42. Thaller, H., Linsbauer, L., Egyed, A., Fischer, S.: Towards Fault Localization via Probabilistic Software Modeling. In: 2020 IEEE 3rd International Workshop on Validation, Analysis, and Evolution of Software Tests (VST). p. 24–27. IEEE (2020)
43. Thaller, H., Linsbauer, L., Ramler, R., Egyed, A.: Probabilistic Software Modeling: A Data-driven Paradigm for Software Analysis. arXiv:1912.07936 [cs] (Dec 2019)
44. Thaller, H., Ramler, R., Pichler, J., Egyed, A.: Exploring code clones in programmable logic controller software. In: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). p. 1–8. IEEE, Limassol (Sep 2017). https://doi.org/10.1109/ETFA.2017.8247574
45. Van Rossum, G., Drake, F.L.: Python 3 Reference Manual. CreateSpace, Scotts Valley, CA (2009)
46. Wagner, S., Abdulkhaleq, A., Bogicevic, I., Ostberg, J.P., Ramadani, J.: How are functionally similar code clones syntactically different? An empirical study and a benchmark. PeerJ Computer Science **2**, e49 (Mar 2016). https://doi.org/10.7717/peerj-cs.49
47. Wang, P., Svajlenko, J., Wu, Y., Xu, Y., Roy, C.K.: Ccaligner: a token based large-gap clone detector. In: Proceedings of the 40th International Conference on Software Engineering. p. 1066–1077 (2018)
48. Wang, W., Li, G., Ma, B., Xia, X., Jin, Z.: Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE (feb 2020). https://doi.org/10.1109/saner48275.2020.9054857
49. Wei, H.H., Li, M.: Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence. p. 3034–3040. IJCAI'17, AAAI Press, Melbourne, Australia (Aug 2017)
50. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree (may 2019). https://doi.org/10.1109/ICSE.2019.00086
51. Zhao, G., Huang, J.: DeepSim: Deep learning code functional similarity. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 141–151. ESEC/FSE 2018, Association for Computing Machinery, Lake Buena Vista, FL, USA (Oct 2018). https://doi.org/10.1145/3236024.3236068

QMaxUSE: A Query-based Verification Tool for UML Class Diagrams with OCL Invariants

Hao Wu (✉)

Computer Science Department, Maynooth University, Maynooth, Ireland
haowu@cs.nuim.ie

Abstract. Verifying whether a UML class diagram annotated with Object Constraint Language (OCL) constraints is consistent involves finding valid instances that provably meet its structural and OCL constraints. Recently, many tools and techniques have been proposed to find valid instances. However, they often do not scale well when the number of OCL constraints significantly increases. In this paper, we present a new tool called QMaxUSE that is capable of automatically verifying a large number of OCL invariants. QMaxUSE works by decomposing them into a set of different queries. It then uses an SMT solver to concurrently verify each query and pinpoints conflicting OCL invariants. Our evaluation results suggest that QMaxUSE can offer up to 30x efficiency improvement in verifying UML class diagrams with a large number of OCL invariants.

1 Introduction

Verifying the consistency of a UML class diagram annotated with OCL constraints is a challenging task [1,2,3]. This is because it requires finding an instance satisfying both structural and OCL constraints at the same time. To tackle this challenge, many tools and techniques have been proposed [4,5,6,7,8]. However, most of these tools and techniques do not scale well when the number of OCL invariants significantly increases [9,10,11,12,13,5,14,15,16]. These tools often time out or cannot pinpoint the conflicting OCL invariants that cause a UML class diagram to become inconsistent.

In this paper, we present a new tool QMaxUSE that is capable of verifying a large number of complex OCL invariants in an efficient manner. This is achieved by two distinct features provided within QMaxUSE. (1) a query language that allows users to select parts of a UML class diagram to be verified. (2) a new specialised algorithm that is able to decompose a UML class diagram that has a large number of complex OCL invariants into different queries. These queries can then be verified concurrently via efficient SMT solving. The detailed explanation of our approach can be found in [17].

Related Work. Verifying the consistencies of a UML class diagram has gained much attention in recent years and many approaches and tools are proposed. A UML class diagram can be considered as a graph, so graph-based approaches are naturally employed for reasoning about consistencies [18,19,20,7,21]. Semeráth

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 310–317, 2022.
https://doi.org/10.1007/978-3-030-99429-7_17

et al. proposed a new graph solver that is able to generate much larger number of objects [22]. Their approach utilises a combination of multiple advanced graph-based and SAT-solving techniques to achieve large-scale graphs generation. On the other hand, many tools incorporate logic solvers to support OCL constraints solving [14,16,23,24,25]. However, many of them do not scale well and cannot pinpoint conflicting OCL constraints when a UML class diagram is inconsistent. Our goal here is to provide an open-source tool that is capable of not only locating conflicting OCL constraints but also preserves high-performance when the number of OCL constraints significantly increases.

2 Architecture

QMaxUSE is fully automatic and integrated with the USE modelling tool [26]. Currently, it is command-line based and can be run under operating system Windows 10 (x64), Ubuntu 20.04 (x64) and Mac OS Big Sur(x64). QMaxUSE is implemented in Java. It consists of nearly 33k lines of code, and approximately $3.5k$ lines of code are dedicated to its algorithms. The latest version of QMaxUSE is available at [27]:

https://github.com/classicwuhao/qmaxuse

The architecture of QMaxUSE is shown in Figure 1. Overall, it has four layers: front-end, query engine, translation and solver.

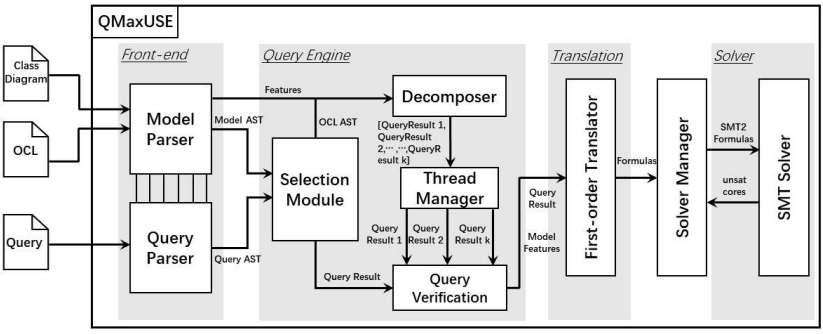


Fig. 1. The overall architecture of QMaxUSE.

Front-end. At the front-end layer, QMaxUSE uses parsers from USE to generate ASTs (abstract syntax trees) for a class diagram and OCL invariants. QMaxUSE provides a simple query language that allows users to choose a part of a class diagram and its OCL invariants to be verified. To parse a query issued by a user, we have designed and implemented a query parser. This parser is able to read multiple queries simultaneously in a specification file and produces corresponding ASTs.

Query Engine. QMaxUSE's query engine uses a set of selection algorithms to traverse the ASTs generated from the front-end layer to produce a query result. A query result essentially contains a set of classes, attributes, associations and

OCL invariants to be verified. At this layer, QMaxUSE also provides a *specialised algorithm* (Decomposer) that is able to decompose a class diagram along with OCL invariants into a set of different queries. These queries can then be verified concurrently using a query verification procedure.

Translation. At the translation layer, QMaxUSE uses a *first-order translator* to translate a query into a set of first-order formulas that can be verified by the SMT solver. The translation here is similar to the one described in [8]. We use uninterpreted functions to encode classes or attributes and linear integer inequalities to capture the multiplicities at an association-end. For an OCL invariant, we traverse its AST and generate an SMT formula by using a combination of first-order theories.

Solver. We have designed a new interface (*SolverManager*) to optimise the interaction between QMaxUSE and the SMT solver. This interface can reduce extra overhead between our first-order translator and an SMT solver by minimising the number of APIs calls. Currently, QMaxUSE uses Z3 as its default SMT solver and this new interface easily allows us to plug in other SMT solvers [28].

3 Design

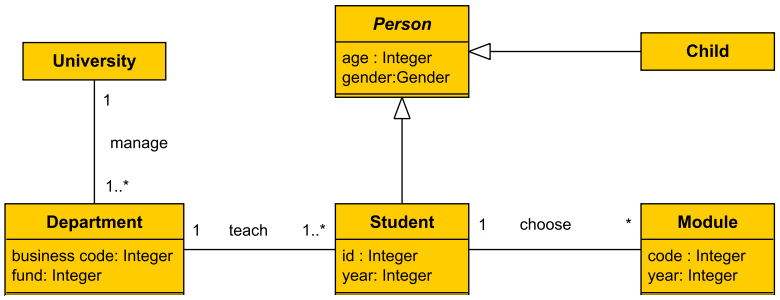
3.1 Query

QMaxUSE allows a user to verify a particular set of features of a UML class diagram through a query language. A *query* expression accepted by QMaxUSE must use a *select* statement. It allows users to choose multiple features along with OCL invariants from a UML class diagram. A *feature* here may include a *class*, an *attribute*, an *association* or an OCL *invariant*. For example, the following query (*query* 1) first selects the *University*, *Department*, *Student* and *Module* class, an association *teach* along with the invariant defined under the *Module* class from the UML class diagram in Figure 2.

query 1 : select University, Student.*, Department:teach:Student with Student::inv2, Module::*

Notably, we allow users to use a wild character $*$ to represent a set of features under a specified classifier. Further, it is quite common that an OCL invariant may use features from other classes in its expression. Hence, our selection algorithm implicitly selects these features during the execution of a query. Thus, *query* 1 also selects the *Person* class from Figure 2 since *inv*2 defined under the *Student* class imposes a constraint on the *age* attribute that is inherited from the *Person* class.

For each query issued by a user, QMaxUSE launches a verification procedure that is able to verify the consistencies of the collected features. This verification procedure casts the set of collected features to a set of SMT formulas that can be checked by an SMT solver. If the formulas are not satisfied, QMaxUSE reports inconsistencies by pinpointing the OCL invariants that cause conflicts. For example, QMaxUSE reports that there is a conflict between OCL invariant *inv*1 and *inv*2 after verifying the following query (*query* 2). It shows that both




```
context Person
inv1: Person.allInstances()->forAll(p|p.age>0 and p.age<18)

context Student
inv2: self.age>18
inv3: self.year>=1 and self.year<=6
inv4: Student.allInstances()->forAll(s1,s2:Student|s1<>s2 implies s1.id <> s2.id)
inv5: Student.allInstances()->forAll(s|s.modules->forAll(m|s.year=m.year))
inv6: Student.allInstances()->exists(s|s.year=6) and Student.allInstances()->exists(s|s.year<6)
inv7: Student.allInstances()->forAll(s|s.modules->notEmpty())

context Module
inv8: self.year>=1 and self.year<=5
```

Fig. 2. A UML class diagram with the 8 OCL class invariants shows how the students in each department can choose multiple modules to study.

*inv*1 and *inv*2 can make the *Student* class impossible to instantiate. Figure 3 shows a screenshot of QMaxUSE after executing *query* 2.

query 2 : `select Person.*, Student.* with Person::inv1, Student::inv2`

3.2 Concurrent Verification

QMaxUSE has a crafted algorithm that is designed for performing concurrent verification on UML class diagrams with a large number of OCL invariants. The main idea of this algorithm is that it is able to decompose a large number of complex OCL invariants into different queries. For each query, it launches a thread of verification procedure to verify that query. In this way, QMaxUSE is able to shift solving a large number of complex formulas from a single run into multiple simultaneous runs on a collection of much smaller and less complex formulas. Therefore, it is particularly powerful when the number of OCL invariants grows significantly.

A high-level structure of this dedicated algorithm is shown in Algorithm 1 [17]. This algorithm takes a UML class diagram annotated with OCL invariants (denoted as *model*) as its input and outputs a set S that contains all possible conflicting features. It first employs a novel decomposition algorithm to decompose a *model* into different parts and produces a query for each part of this model. It then executes each query and produces a new query result by explicitly choosing those features that are used by an OCL invariant expression in

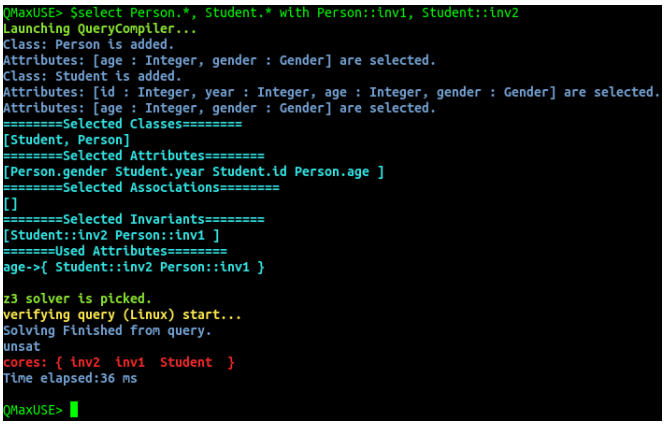


Fig. 3. A screenshot of running *query* 2 in QMaxUSE.

a query. Once the set of query results are generated, Algorithm 1 launches a number of threads to verify the formulas (Φ) that encode query results. If the Φ are not satisfied, then this means that there must be conflicts. Finally, our algorithm extracts those conflicting features and saves them into the set S.

Algorithm 1: ConcurrentVerification

Input : A UML class diagram annotated with OCL invariants (*model*)
Output: A set of conflicting features cause inconsistencies (S).

$R \leftarrow \emptyset \wedge S \leftarrow \emptyset$;
$Q \leftarrow Decompose(model)$; /*produce a set of queries Q*/
foreach $q \in Q$ **do**
  $q_r = q.execute()$; /* create a new query result q_r*/
  /* add features used in an OCL invariant into a query result q_r*/
  foreach $inv \in q$ **do**
    $q_r.add(inv.classes(), inv.attributes(), inv.associations(), inv)$;
  end
  $R.add(q_r)$;
end
/* verify *model* with $|R|$ number of threads. */
foreach $q_r \in R$ **do**
  $\Phi \leftarrow Translate(q_r)$; /*cast q_r to SMT formulas*/
  $ThreadManager.start(QueryVerification(\Phi, S))$;
  /*check satisfiability of Φ and saves each conflict occurred in Φ in the set S*/
end
return S;

	Name	OCL Structure Size				MaxUSE [12] (sec)	QMaxUSE (sec)	
		Invs	Nodes	Quant	Op	Time	Threads	Time
Part A	A1	6	30	3	9	0.38	2	0.364
	A2	7	52	8	1	0.148	1	0.087
	A3	1	7	2	1	0.174	3	0.426
	A4	8	73	7	18	0.204	3	0.241
Part B	B3	68	430	9	111	131.23	42	4.604
	B4	90	599	23	152	159.378	68	7.151
	C4	98	698	69	137	TO	68	8.111
	C5	156	1008	100	184	TO	90	114.41
	B5	136	925	44	228	TO	90	118.64
	D4	101	753	102	163	TO	56	8.211
	D5	166	1143	131	225	TO	95	14.026
	E3	37	403	31	102	59.535	27	2.587
	E4	105	985	56	246	TO	42	4.464
	E5	167	1134	68	325	TO	45	3.653

Table 1. Evaluation results. Invs=number of OCL invariants, Nodes=size of invariant ASTs, Quant=number of quantifiers, Op=number of operators. TO= Timeout (20min), MaxUSE=QMaxUSE without query and concurrent verification support.

4 Results

We use a benchmark from [8] to show the size and the complexities of OCL invariants QMaxUSE can handle. This benchmark has two parts. Part A only covers a small number of toy examples from [29] and Part B covers a wide range of OCL language features including: nested quantifiers, collections, logical/arithmetic operations and navigations. In particular, Part B contains a large number of complex and conflicting OCL invariants. Table 1 summarises part of our evaluation results for QMaxUSE [1]. The evaluation is carried out on an Intel(R) Core (TM) machine that has six 2.8GHz cores with 16G memory. The underlying SMT solver is the Z3 SMT solver (version 4.8.10). As it can seen that QMaxUSE is able to handle much larger size of OCL invariants. It is able to gain upto 30x efficiency in improvement in verifying large number of complex OCL invariants. For example, it takes 131.23 seconds to verify $B3$ in Group B without using our query and concurrent techniques while QMaxUSE is able to finish its verification in just 4.6 seconds.

5 Conclusion

In this paper, we have presented our latest verification tool QMaxUSE. We believe that QMaxUSE can add significant value in modelling community for two reasons. (1) Users now are able to use QMaxUSE to incrementally verify different parts of their class diagrams by issuing different queries. (2) Our preliminary evaluation results indicate that QMaxUSE can scale well on a large number of complex OCL invariants because of our concurrent verification algorithm.

[1] The complete benchmark is packed within QMaxUSE release files.

References

1. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams is EXPTIME-hard. In: International Workshop on Description Logics. (2003)
2. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artificial Intelligence **168**(1–2) (2005) 70–118
3. Queralt, A., Teniente, E.: Reasoning on uml class diagrams with ocl constraints. In: Conceptual Modeling, Springer (2006) 497–512
4. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. Data & Knowledge Engineering **73** (2012) 1–22
5. Dania, C., Clavel, M.: Ocl2msfol: A mapping to many-sorted first-order logic for efficiently checking the satisfiability of ocl constraints. In: International Conference on Model Driven Engineering Languages and Systems, ACM (2016) 65–75
6. Maraee, A., Balaban, M.: Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In: 3rd European Conference Model Driven Architecture, Springer (2007) 17–31
7. Balaban, M., Maraee, A.: Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy. ACM Transactions on Software Engineering and Methodology **22**(3) (2013) 24:1–24:42
8. Wu, H., Farrell, M.: A formal approach to finding inconsistencies in a metamodel. Software and Systems Modeling (2021)
9. González Pérez, C.A., Buettner, F., Clarisó, R., Cabot, J.: EMFtoCSP: A tool for the lightweight verification of EMF models. In: International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches, IEEE (2012) 44–50
10. Kuhlmann, M., Gogolla, M.: From uml and ocl to relational logic and back. In: International Conference on Model Driven Engineering Languages and Systems, Springer (2012) 415–431
11. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: Ocl-lite: Finite reasoning on UML/OCL conceptual schemas. Data & Knowledge Engineering **73** (2012) 1 – 22
12. Wu, H.: Maxuse: A tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In: Integrated Formal Methods, Springer (2017) 348–356
13. Wille, R., Soeken, M., Drechsler, R.: Debugging of inconsistent UML/OCL models. In: Design, Automation Test in Europe, IEEE (2012) 1078–1083
14. Wu, H., Monahan, R., Power, J.F.: Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In: International Symposium on Theoretical Aspects of Software Engineering, IEEE (2013) 175–182
15. Wu, H.: Generating metamodel instances satisfying coverage criteria via SMT solving. In: International Conference on Model-Driven Engineering and Software Development, IEEE (2016) 40–51
16. Soeken, M., Wille, R., Drechsler, R.: Verifying dynamic aspects of uml models. In: Design, Automation Test in Europe, IEEE (March 2011) 1–6
17. Wu, H.: A query-based approach for verifying UML class diagrams with OCL invariants (to appear). In: 18th European Conference on Modelling Foundations and Applications
18. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software and Systems Modeling **8**(4) (2009) 479–500
19. Hoffmann, B., Minas, M.: Defining models - meta models versus graph grammars. Electronic Communications of the EASST **29** (2010) 1–14

20. Hoffmann, B., Minas, M.: Generating instance graphs from class diagrams with adaptive star grammars. In: International Workshop on Graph Computation Models, Electronic Communications of the EASST (2011)
21. Maraee, A., Balaban, M.: Removing redundancies and deducing equivalences in UML class diagrams. In: International Conference Model-Driven Engineering Languages and Systems, Springer (2014) 235–251
22. Semeráth, O., Nagy, A.S., Varró, D.: A graph solver for the automated generation of consistent domain-specific models. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, Association for Computing Machinery (2018) 969–980
23. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: International Conference on Model Driven Engineering Languages and Systems, Springer (2007) 436–450
24. Semeráth, O., Vörös, A., Varró, D.: Iterative and incremental model generation by logic solvers. In: 19th International Conference on Fundamental Approaches to Software Engineering, Springer (2016) 87–103
25. Kuhlmann, M., Gogolla, M.: Strengthening SAT-based validation of UML/OCL models by representing collections as relations. In: Modelling Foundations and Applications. Volume 7349. Springer (2012) 32–48
26. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming **69**(1-3) (2007) 27–34
27. QMaxUSE: https://doi.org/10.5281/zenodo.5804509
28. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer (2008) 337–340
29. Gogolla, M., Büttner, F., Cabot, J.: Initiating a benchmark for UML and OCL analysis tools. In: International Conference on Tests and Proofs, Springer (2013) 115–132

Test-Comp Contributions

Advances in Automatic Software Testing: Test-Comp 2022

Dirk Beyer

LMU Munich, Munich, Germany

Abstract. Test-Comp 2022 is the 4th edition of the Competition on Software Testing. Research competitions are a means to provide annual comparative evaluations. Test-Comp focusses on fully automatic software test generators for C programs. The results of the competition shall be reproducible and provide an overview of the current state of the art in the area of automatic test-generation. The competition was based on 4 236 test-generation tasks for C programs. Each test-generation task consisted of a program and a test specification (error coverage, branch coverage). Test-Comp 2022 had 12 participating test generators from 5 countries.

Keywords: Software Testing · Test-Case Generation · Competition · Program Analysis · Software Validation · Software Bugs · Test Validation · Test-Comp · Benchmarking · Test Coverage · Bug Finding · Test-Suites · SV-Benchmarks · BenchExec · TestCov · CoVeriTeam

1 Introduction

The Competition on Software Testing (Test-Comp, `https://test-comp.sosy-lab.org`, [5, 6, 7, 9]) showcases the state of the art in the area of automatic software testing. For the 4th time, the competition provides an overview of the results achieved by implementations of the most recent ideas, concepts, and algorithms for fully automatic test generation. This competition report describes the (updated) rules and definitions, presents the competition results, and discusses some interesting facts about the execution of the competition experiments. We use BenchExec [20] to execute the benchmarks and the results are presented in tables and graphs on the competition web site (`https://test-comp.sosy-lab.org/2022/results`) and are available in the accompanying archives (see Table 3).

This report extends previous reports on Test-Comp [5, 6, 7, 9].
Reproduction packages are available on Zenodo (see Table 3).

✉ `dirk.beyer@sosy-lab.org`

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 321–335, 2022.
https://doi.org/10.1007/978-3-030-99429-7_18

Competition Goals. In summary, the goals of Test-Comp are the following [6]:

- Establish *standards* for software test generation. This means, most prominently, to develop a standard for marking input values in programs, define an exchange format for test suites, agree on a specification language for test-coverage criteria, and define how to validate the resulting test suites.
- Establish a set of *benchmarks* for software testing in the community. This means to create and maintain a set of programs together with coverage criteria, and to make those publicly available for researchers to be used in performance comparisons when evaluating a new technique.
- Provide an overview of *available tools* for test-case generation and a snapshot of the state-of-the-art in software testing to the community. This means to compare, independently from particular paper projects and specific techniques, different test generators in terms of effectiveness and performance.
- Increase the visibility and credits that *tool developers* receive. This means to provide a forum for presentation of tools and discussion of the latest technologies, and to give the participants the opportunity to publish about the development work that they have done.
- Educate PhD students and other participants on how to set up performance experiments, package tools in a way that supports reproduction, and how to perform *robust and accurate research experiments*.
- Provide *resources* to development teams that do not have sufficient computing resources and give them the opportunity to obtain results from experiments on large benchmark sets.

Related Competitions. In the field of formal methods, competitions are respected as an important evaluation method and there are many competitions [3]. We refer to the report from Test-Comp 2020 [6] for a more detailed discussion and give here only the references to the most related competitions [3, 10, 41, 43].

2 Definitions, Formats, and Rules

Organizational aspects such as the classification (automatic, off-site, reproducible, jury, training) and the competition schedule is given in the initial competition definition [5]. In the following, we repeat some important definitions that are necessary to understand the results.

Test-Generation Task. A *test-generation task* is a pair of an input program (program under test) and a test specification. A *test-generation run* is a non-interactive execution of a test generator on a single test-generation task, in order to generate a test suite according to the test specification. A *test suite* is a sequence of test cases, given as a directory of files according to the format for exchangeable test-suites.[1]

Execution of a Test Generator. Figure 1 illustrates the process of executing one test generator on the benchmark suite. One test run for a test generator gets

[1] https://gitlab.com/sosy-lab/software/test-format

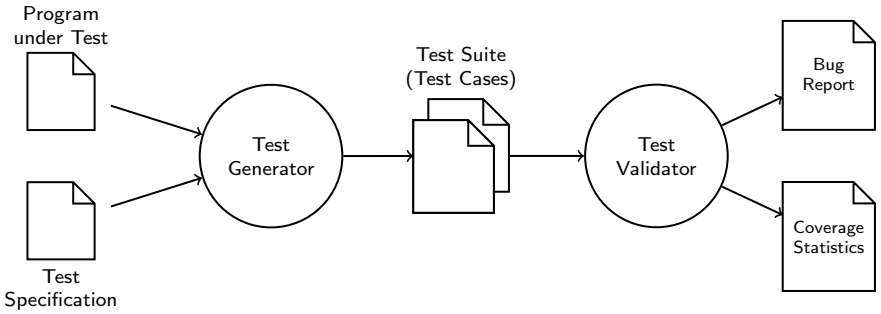


Fig. 1: Flow of the Test-Comp execution for one test generator (taken from [6])

as input (i) a program from the benchmark suite and (ii) a test specification (cover bug, or cover branches), and returns as output a test suite (i.e., a set of test cases). The test generator is contributed by a competition participant as a software archive in ZIP format. The test runs are executed centrally by the competition organizer. The test-suite validator takes as input the test suite from the test generator and validates it by executing the program on all test cases: for bug finding it checks if the bug is exposed and for coverage it reports the coverage. We use the tool TestCov [19] [2] as test-suite validator.

Test Specification. The specification for testing a program is given to the test generator as input file (either `properties/coverage-error-call.prp` or `properties/coverage-branches.prp` for Test-Comp 2022).

The definition `init(main())` is used to define the initial states of the program under test by a call of function `main` (with no parameters). The definition `FQL(f)` specifies that coverage definition `f` should be achieved. The FQL (FShell query language [30]) coverage definition `COVER EDGES(@DECISIONEDGE)` means that all branches should be covered (typically used to obtain a standard test suite for quality assurance) and `COVER EDGES(@CALL(foo))` means that a call (at least one) to function `foo` should be covered (typically used for bug finding). A complete specification looks like: `COVER(init(main()), FQL(COVER EDGES(@DECISIONEDGE)))`.

Table 1 lists the two FQL formulas that are used in test specifications of Test-Comp 2022; there was no change from 2020 (except that special function `__VERIFIER_error` does not exist anymore).

Task-Definition Format 2.0. Test-Comp 2022 used again the task-definition format in version 2.0.

License and Qualification. The license of each participating test generator must allow its free use for reproduction of the competition results. Details on qualification criteria can be found in the competition report of Test-Comp 2019 [7].

[2] https://gitlab.com/sosy-lab/software/test-suite-validator

Table 1: Coverage specifications used in Test-Comp 2022 (similar to 2019–2021)

Formula	Interpretation
`COVER EDGES(@CALL(reach_error))`	The test suite contains at least one test that executes function `reach_error`.
`COVER EDGES(@DECISIONEDGE)`	The test suite contains tests such that all branches of the program are executed.

3 Categories and Scoring Schema

Benchmark Programs. The input programs were taken from the largest and most diverse open-source repository of software-verification and test-generation tasks[3], which is also used by SV-COMP [8]. As in 2020 and 2021, we selected all programs for which the following properties were satisfied (see issue on GitHub[4] and report [7]):

1. compiles with `gcc`, if a harness for the special methods[5] is provided,
2. should contain at least one call to a nondeterministic function,
3. does not rely on nondeterministic pointers,
4. does not have expected result 'false' for property 'termination', and
5. has expected result 'false' for property 'unreach-call' (only for category *Error Coverage*).

This selection yielded a total of 4 236 test-generation tasks, namely 776 tasks for category *Error Coverage* and 3 460 tasks for category *Code Coverage*. The test-generation tasks are partitioned into categories, which are listed in Tables 6 and 7 and described in detail on the competition web site.[6] Figure 2 illustrates the category composition.

Category Error-Coverage. The first category is to show the abilities to discover bugs. The benchmark set consists of programs that contain a bug. Every run will be started by a batch script, which produces for every tool and every test-generation task one of the following scores: 1 point, if the validator succeeds in executing the program under test on a generated test case that explores the bug (i.e., the specified function was called), and 0 points, otherwise.

Category Branch-Coverage. The second category is to cover as many branches of the program as possible. The coverage criterion was chosen because many test generators support this standard criterion by default. Other coverage criteria can be reduced to branch coverage by transformation [29]. Every run will be started by a batch script, which produces for every tool and every

[3] https://github.com/sosy-lab/sv-benchmarks
[4] https://github.com/sosy-lab/sv-benchmarks/pull/774
[5] https://test-comp.sosy-lab.org/2022/rules.php
[6] https://test-comp.sosy-lab.org/2022/benchmarks.php

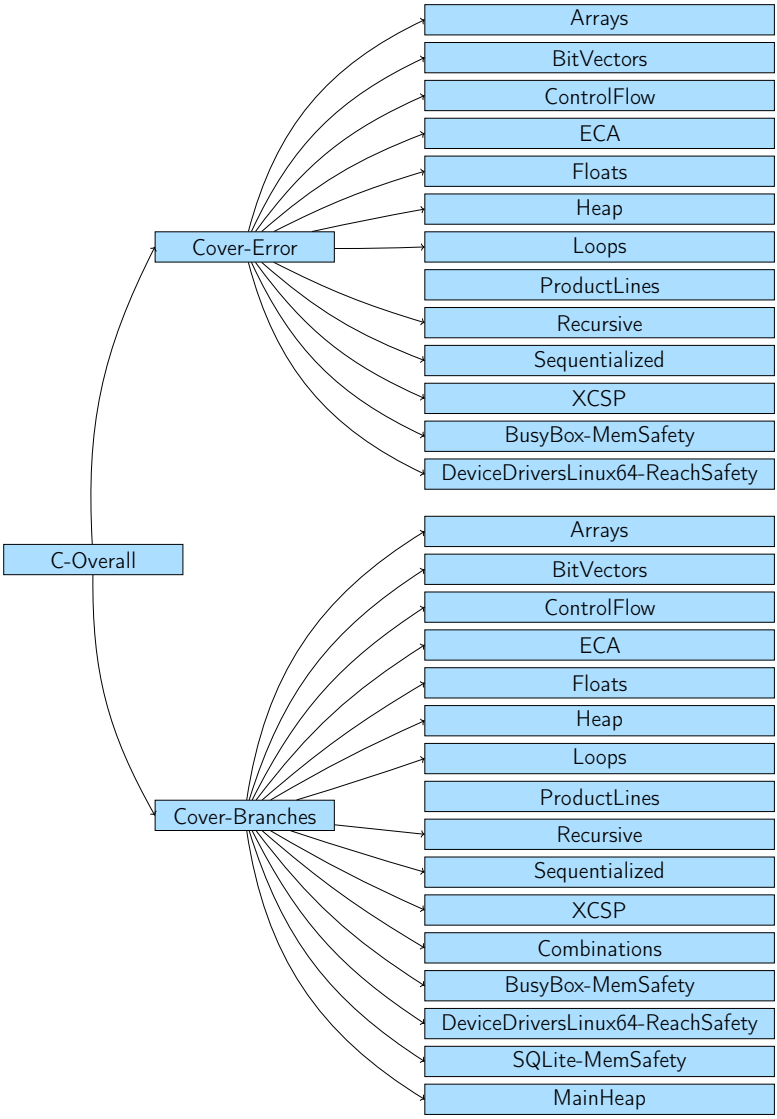


Fig. 2: Category structure for Test-Comp 2022; compared to Test-Comp 2021, sub-category *ProductLines* was added to both main categories *Cover-Error* and *Cover-Branches*

test-generation task the coverage of branches of the program (as reported by TestCov [19]; a value between 0 and 1) that are executed for the generated test cases. The score is the returned coverage.

Ranking. The ranking was decided based on the sum of points (normalized for meta categories). In case of a tie, the ranking was decided based on the run time,

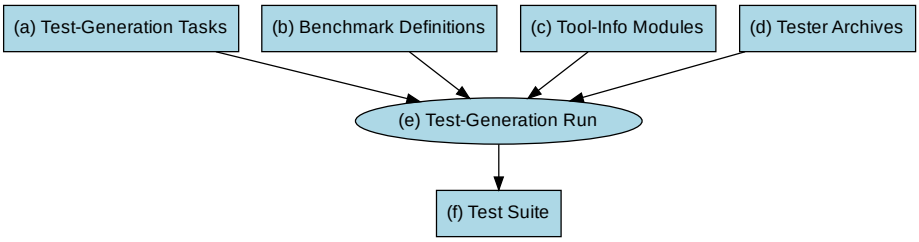


Fig. 3: Benchmarking components of Test-Comp and competition's execution flow (same as for Test-Comp 2020)

Table 2: Publicly available components for reproducing Test-Comp 2022

Component	Fig. 3	Repository	Version
Test-Generation Tasks	(a)	gitlab.com/sosy-lab/benchmarking/sv-benchmarks	testcomp22
Benchmark Definitions	(b)	gitlab.com/sosy-lab/test-comp/bench-defs	testcomp22
Tool-Info Modules	(c)	github.com/sosy-lab/benchexec	3.10
Test-Generator Archives	(d)	gitlab.com/sosy-lab/test-comp/archives-2022	testcomp22
Benchmarking	(e)	github.com/sosy-lab/benchexec	3.10
Test-Suite Format	(f)	gitlab.com/sosy-lab/software/test-format	testcomp22

which is the total CPU time over all test-generation tasks. Opt-out from categories was possible and scores for categories were normalized based on the number of tasks per category (see competition report of SV-COMP 2013 [4], page 597).

4 Reproducibility

We followed the same competition workflow that was described in detail in the previous competition report (see Sect. 4, [9]). All major components that were used for the competition were made available in public version-control repositories. An overview of the components that contribute to the reproducible setup of Test-Comp is provided in Fig. 3, and the details are given in Table 2. We refer to the report of Test-Comp 2019 [7] for a thorough description of all components of the Test-Comp organization and how we ensure that all parts are publicly available for maximal reproducibility.

In order to guarantee long-term availability and immutability of the test-generation tasks, the produced competition results, and the produced test suites, we also packaged the material and published it at Zenodo (see Table 3).

The competition used CoVeriTeam [17][7] again to provide participants access to the actual competition machines. The competition report of SV-COMP 2022 provides a description on reproducing individual results and on trouble-shooting (see Sect. 3, [10]).

[7] https://gitlab.com/sosy-lab/software/coveriteam

Table 3: Artifacts published for Test-Comp 2022

Content	DOI	Reference
Test-Generation Tasks	10.5281/zenodo.5831003	[12]
Competition Results	10.5281/zenodo.5831012	[11]
Test-Suite Generators	10.5281/zenodo.5959598	[13]
Test Suites (Witnesses)	10.5281/zenodo.5831010	[14]
BenchExec	10.5281/zenodo.5720267	[47]

Table 4: Competition candidates with tool references and representing jury members; new indicates first-time participants, ∅ indicates hors-concours participation

Tester	Ref.	Jury member	Affiliation
CMA-ES Fuzz∅	[34]	(hors concours)	–
CoVeriTest	[16, 33]	Marie-Christine Jakobs	TU Darmstadt, Germany
FuSeBMC	[1, 2]	Kaled Alshmrany	U. of Manchester, UK
HybridTiger∅	[22, 42]	(hors concours)	–
Klee∅	[23, 24]	(hors concours)	–
Legion	[38, 39]	Gidon Ernst	LMU Munich, Germany
Legion/SymCC new	[39]	Gidon Ernst	LMU Munich, Germany
LibKluzzer	[36]	Hoang M. Le	U. of Bremen, Germany
PRTest	[18, 37]	Thomas Lemberger	LMU Munich, Germany
Symbiotic	[25, 26]	Marek Chalupa	Masaryk U., Brno, Czechia
TracerX	[31, 32]	Joxan Jaffar	National U., Singapore
VeriFuzz	[40]	Raveendra Kumar M.	Tata Consultancy Services, India

5 Results and Discussion

This section represents the results of the competition experiments. The report shall help to understanding the state of the art and the advances in fully automatic test generation for whole C programs, in terms of effectiveness (test coverage, as accumulated in the score) and efficiency (resource consumption in terms of CPU time). All results mentioned in this article were inspected and approved by the participants.

Participating Test Generators. Table 4 provides an overview of the participating test generators and references to publications, as well as the team representatives of the jury of Test-Comp 2022. (The competition jury consists of the chair and one member of each participating team.) An online table with information about all participating systems is provided on the competition web site.[8] Table 5 lists the features and technologies that are used in the test generators.

There are test generators that did not actively participate (e.g., tester archives taken from last year) and that are not included in rankings. Those are called *hors-concours* participations and the tool names are labeled with a symbol (∅).

[8] https://test-comp.sosy-lab.org/2022/systems.php

Table 5: Technologies and features that the test generators used

Participant	Bounded Model Checking	CEGAR	Evolutionary Algorithms	Explicit-Value Analysis	Floating-Point Arithmetics	Guidance by Coverage Measures	Predicate Abstraction	Random Execution	Symbolic Execution	Targeted Input Generation	Algorithm Selection	Portfolio
CMA-ES Fuzz[∅]			✓	✓	✓	✓		✓				
CoVeriTest		✓		✓	✓		✓					✓
FuSeBMC	✓				✓	✓				✓		✓
HybridTiger[∅]		✓		✓	✓		✓					
Klee[∅]					✓				✓	✓		
Legion				✓	✓	✓		✓	✓	✓		
Legion/SymCC[new]				✓	✓	✓		✓	✓	✓		
LibKluzzer					✓	✓		✓	✓			
PRTest					✓			✓				
Symbiotic					✓	✓			✓	✓		✓
TracerX	✓				✓				✓	✓		
VeriFuzz	✓		✓	✓	✓	✓		✓				

Computing Resources. The computing environment and the resource limits were the same as for Test-Comp 2020 [6]: Each test run was limited to 8 processing units (cores), 15 GB of memory, and 15 min of CPU time. The test-suite validation was limited to 2 processing units, 7 GB of memory, and 5 min of CPU time. The machines for running the experiments are part of a compute cluster that consists of 167 machines; each test-generation run was executed on an otherwise completely unloaded, dedicated machine, in order to achieve precise measurements. Each machine had one Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 20.04 with Linux kernel 5.4). We used BenchExec [20] to measure and control computing resources (CPU time, memory, CPU energy) and VerifierCloud[9] to distribute, install,

[9] https://vcloud.sosy-lab.org

Table 6: Quantitative overview over all results; empty cells mark opt-outs; new indicates first-time participants, ∅ indicates hors-concours participation

Tester	Cover-Error 776 tasks	Cover-Branches 3460 tasks	Overall 4236 tasks
CMA-ES Fuzz∅	0	624	382
CoVeriTest	423	1860	2293
FuSeBMC	**628**	**2104**	**3003**
HybridTiger∅	355	1406	1830
Klee∅	500	1242	2125
Legion	57	1033	787
Legion/SymCC new		1487	
LibKluzzer	**528**	**1990**	**2658**
PRTest	145	896	945
Symbiotic	463	1802	2367
TracerX	0	1746	1069
VeriFuzz	**623**	**2075**	**2971**

run, and clean-up test-case generation runs, and to collect the results. The values for time and energy are accumulated over all cores of the CPU. To measure the CPU energy, we use CPU Energy Meter [21] (integrated in BenchExec [20]). Further technical parameters of the competition machines are available in the repository which also contains the benchmark definitions. [10]

One complete test-generation execution of the competition consisted of 50 056 single test-generation runs. The total CPU time was 339 days and the consumed energy 88 kWh for one complete competition run for test generation (without validation). Test-suite validation consisted of 50 832 single test-suite validation runs. The total consumed CPU time was 15 days. Each tool was executed several times, in order to make sure no installation issues occur during the execution. Including preruns, the infrastructure managed a total of 311 754 test-generation runs (consuming 4.9 years of CPU time). The CPU energy was not measured during preruns.

Quantitative Results. The quantitative results are presented in the same way as last year: Table 6 presents the quantitative overview of all tools and all categories. The head row mentions the category and the number of test-generation

[10] https://gitlab.com/sosy-lab/test-comp/bench-defs/tree/testcomp22

Table 7: Overview of the top-three test generators for each category (measurement values for CPU time and energy rounded to two significant digits)

Rank	Tester	Score	CPU Time (in h)	CPU Energy (in kWh)
Cover-Error				
1	FuSeBMC	**628**	22	0.28
2	VeriFuzz	623	3.5	0.039
3	LibKluzzer	528	140	1.5
Cover-Branches				
1	FuSeBMC	**2104**	850	11
2	VeriFuzz	2075	850	11
3	LibKluzzer	1990	760	8.3
Overall				
1	FuSeBMC	**3003**	870	11
2	VeriFuzz	2971	860	11
3	LibKluzzer	2658	900	9.8

tasks in that category. The tools are listed in alphabetical order; every table row lists the scores of one test generator. We indicate the top three candidates by formatting their scores in bold face and in larger font size. An empty table cell means that the test generator opted-out from the respective main category (perhaps participating in subcategories only, restricting the evaluation to a specific topic). More information (including interactive tables, quantile plots for every category, and also the raw data in XML format) is available on the competition web site [11] and in the results artifact (see Table 3). Table 7 reports the top three test generators for each category. The consumed run time (column 'CPU Time') is given in hours and the consumed energy (column 'Energy') is given in kWh.

Score-Based Quantile Functions for Quality Assessment. We use score-based quantile functions [20] because these visualizations make it easier to understand the results of the comparative evaluation. The web site [11] and the results artifact (Table 3) include such a plot for each category; as example, we show the plot for category *Overall* (all test-generation tasks) in Fig. 4. We had 11 test generators participating in category *Overall*, for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized [4]). A more detailed discussion of score-based quantile plots for testing is provided in the Test-Comp 2019 competition report [7].

Alternative Rankings. Table 8 is similar to Table 7, but contains the alternative ranking category *Green Testing*. Column 'Quality' gives the score in score points (sp), column 'CPU Time' the CPU usage in hours (h), column

[11] https://test-comp.sosy-lab.org/2022/results

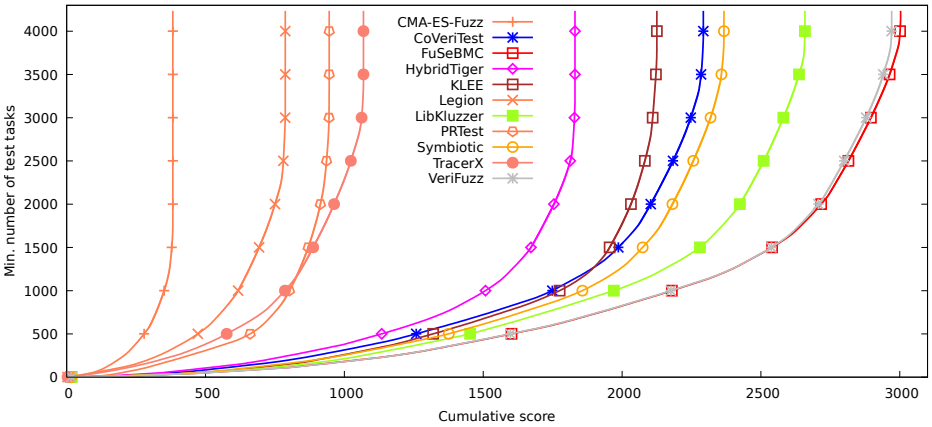


Fig. 4: Quantile functions for category *Overall*. Each quantile function illustrates the quantile (x-coordinate) of the scores obtained by test-generation runs below a certain number of test-generation tasks (y-coordinate). More details were given previously [7]. The graphs are decorated with symbols to make them better distinguishable without color.

Table 8: Alternative rankings; quality is given in score points (sp), CPU time in hours (h), energy in kilo-watt-hours (kWh), the first rank measure in kilo-joule per score point (kJ/sp), and the second rank measure in score points (sp); measurement values are rounded to 2 significant digits

Rank	Test Generator	Quality (sp)	CPU Time (h)	CPU Energy (kWh)	Rank Measure
Green Testing					**(kJ/sp)**
1	TracerX	1 069	120	1.4	4.8
2	Klee$^{\varnothing}$	2 125	310	3.5	6.0
3	Symbiotic	2 367	540	5.9	9.0
worst					41

'CPU Energy' the CPU usage in kilo-watt-hours (kWh), and column 'Rank Measure' reports the values for the rank measure.

Green Testing — Low Energy Consumption. Since a large part of the cost of test generation is caused by the energy consumption, it might be important to also consider the energy efficiency in rankings, as complement to the official Test-Comp ranking. This alternative ranking category uses the energy consumption per score point as rank measure: $\frac{\text{CPU Energy}}{\text{Quality}}$, with the unit kilo-joule per score point (kJ/sp).The energy is measured using CPU Energy Meter [21], which we use as part of BenchExec [20].

New Test Generators. To acknowledge the test generators that participated for the first time in Test-Comp, we list the test generators that participated for the first time. CMA-ES Fuzz$^{\varnothing}$ and FuSeBMC participated for the first time in

Table 9: New verifiers in Test-Comp 2021 and Test-Comp 2022; column 'Sub-categories' gives the number of executed categories

Verifier	Language	First Year	Sub-categories
Legion/SymCC[new]	C	2022	16
CMA-ES Fuzz[∅]	C	2021	30
FuSeBMC	C	2021	30

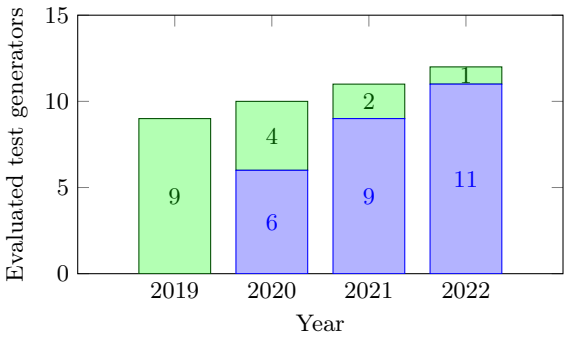


Fig. 5: Number of evaluated test generators for each year (top: number of first-time participants; bottom: previous year's participants)

Test-Comp 2021, and Legion/SymCC[new] participated first in Test-Comp 2022. Table 9 reports also the number of subcategories in which the tools participated.

6 Conclusion

For the 4th time, the Competition on Software Testing took place and provides an overview of test-generation tools for C programs. The competition event attracted 12 participating teams (see Fig. 5 for the participation numbers and Table 4 for the details). The competition is an off-site competition, the execution of the experiments is fully-automatatic and reproducible. To ensure transparency, all components are made available in public repositories and a jury (consisting of members from each team) oversees the process. The produced test suites are validated by the test-suite validator TestCov. The results of the competition are presented at the 25th International Conference on Fundamental Approaches to Software Engineering at ETAPS 2022.

Data-Availability Statement. The test-generation tasks and results of the competition are published at Zenodo, as described in Table 3. All components and data that are necessary for reproducing the competition are available in public version repositories, as specified in Table 2. For easy access, the results are presented also online on the competition web site `https://test-comp.sosy-lab.org/2022/results`.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 418257054 (Coop).

References

1. Alshmrany, K., Aldughaim, M., Cordeiro, L., Bhayat, A.: FuSeBMC v.4: Smart seed generation for hybrid fuzzing (competition contribution). In: Proc. FASE. LNCS 13241, Springer (2022)
2. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. In: Proc. TAP. pp. 85–105. Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6
3. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: TOOLympics 2019: An overview of competitions in formal methods. In: Proc. TACAS (3). pp. 3–24. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_1
4. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS. pp. 594–609. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_43
5. Beyer, D.: Competition on software testing (Test-Comp). In: Proc. TACAS (3). pp. 167–175. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_11
6. Beyer, D.: Second competition on software testing: Test-Comp 2020. In: Proc. FASE. pp. 505–519. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_25
7. Beyer, D.: First international competition on software testing (Test-Comp 2019). Int. J. Softw. Tools Technol. Transf. **23**(6), 833–846 (December 2021). https://doi.org/10.1007/s10009-021-00613-3
8. Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: Proc. TACAS (2). pp. 401–422. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_24
9. Beyer, D.: Status report on software testing: Test-Comp 2021. In: Proc. FASE. pp. 341–357. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_17
10. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. LNCS 13244, Springer (2022)
11. Beyer, D.: Results of the 4th Intl. Competition on Software Testing (Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831012
12. Beyer, D.: SV-Benchmarks: Benchmark set for softwware verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831003
13. Beyer, D.: Test-suite generators and validator of the 4th Intl. Competition on Software Testing (Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959598
14. Beyer, D.: Test suites from test-generation tools (Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831010
15. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
16. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23

17. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. Springer (2022)
18. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC. pp. 99–114. LNCS 10629, Springer (2017). https://doi.org/10.1007/978-3-319-70389-3_7
19. Beyer, D., Lemberger, T.: TestCov: Robust test-suite execution and coverage measurement. In: Proc. ASE. pp. 1074–1077. IEEE (2019). https://doi.org/10.1109/ASE.2019.00105
20. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
21. Beyer, D., Wendler, P.: CPU Energy Meter: A tool for energy-aware algorithms engineering. In: Proc. TACAS (2). pp. 126–133. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_8
22. Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., von Rhein, A., Apel, S., Beyer, D.: Facilitating reuse in multi-goal test-suite generation for software product lines. In: Proc. FASE. pp. 84–99. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_6
23. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
24. Cadar, C., Nowack, M.: Klee symbolic execution engine in 2019 (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 867 – 870 (December 2021). https://doi.org/10.1007/s10009-020-00570-3
25. Chalupa, M., Novák, J., Strejček, J.: Symbiotic 8: Parallel and targeted test generation (competition contribution). In: Proc. FASE. pp. 368–372. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_20
26. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
27. Cok, D.R., Déharbe, D., Weber, T.: The 2014 SMT competition. JSAT **9**, 207–242 (2016)
28. Godefroid, P., Sen, K.: Combining model checking and testing. In: Handbook of Model Checking, pp. 613–649. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_19
29. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Software Eng. **30**(1), 3–16 (2004). https://doi.org/10.1109/TSE.2004.1265732
30. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Proc. ASE. pp. 407–416. ACM (2010). https://doi.org/10.1145/1858996.1859084
31. Jaffar, J., Maghareh, R., Godboley, S., Ha, X.L.: TracerX: Dynamic symbolic execution with interpolation (competition contribution). In: Proc. FASE. pp. 530–534. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_28
32. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: Tracer: A symbolic execution tool for verification. In: Proc. CAV. pp. 758–766. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_61
33. Jakobs, M.C., Richter, C.: CoVeriTest with adaptive time scheduling (competition contribution). In: Proc. FASE. pp. 358–362. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_18

34. Kim, H.: Fuzzing with stochastic optimization (2020), Bachelor's Thesis, LMU Munich
35. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
36. Le, H.M.: LLVM-based hybrid fuzzing with LibKluzzer (competition contribution). In: Proc. FASE. pp. 535–539. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_29
37. Lemberger, T.: Plain random test generation with PRTest (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 871–873 (December 2021). https://doi.org/10.1007/s10009-020-00568-x
38. Liu, D., Ernst, G., Murray, T., Rubinstein, B.: Legion: Best-first concolic testing (competition contribution). In: Proc. FASE. pp. 545–549. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_31
39. Liu, D., Ernst, G., Murray, T., Rubinstein, B.I.P.: Legion: Best-first concolic testing. In: Proc. ASE. pp. 54–65. IEEE (2020). https://doi.org/10.1145/3324884.3416629
40. Metta, R., Kumar, M.R., Karmarkar, H.: VeriFuzz: Fuzz centric test generation tool (competition contribution). In: Proc. FASE. LNCS 13241, Springer (2022)
41. Panichella, S., Gambi, A., Zampetti, F., Riccio, V.: SBST tool competition 2021. In: Proc. SBST. pp. 20–27. IEEE (2021). https://doi.org/10.1109/SBST52555.2021.00011
42. Ruland, S., Lochau, M., Jakobs, M.C.: HybridTiger: Hybrid model checking and domination-based partitioning for efficient multi-goal test-suite generation (competition contribution). In: Proc. FASE. pp. 520–524. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_26
43. Song, J., Alves-Foss, J.: The DARPA cyber grand challenge: A competitor's perspective, part 2. IEEE Security and Privacy **14**(1), 76–81 (2016). https://doi.org/10.1109/MSP.2016.14
44. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: Proc. IJCAR, pp. 367–373. LNCS 8562, Springer (2014). https://doi.org/10.1007/978-3-319-08587-6_28
45. Sutcliffe, G.: The CADE ATP system competition: CASC. AI Magazine **37**(2), 99–101 (2016)
46. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test-input generation with Java PathFinder. In: Proc. ISSTA. pp. 97–107. ACM (2004). https://doi.org/10.1145/1007512.1007526
47. Wendler, P., Beyer, D.: sosy-lab/benchexec: Release 3.10. Zenodo (2022). https://doi.org/10.5281/zenodo.5720267

FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing

(Competition Contribution)

Kaled M. Alshmrany(✉)[1,2]*, Mohannad Aldughaim[1], Ahmed Bhayat[1], and Lucas C. Cordeiro[1]

[1] University of Manchester, Manchester, UK
[2] Institute of Public Administration, Jeddah, Saudi Arabia
kaled.alshmrany@postgrad.manchester.ac.uk

Abstract. *FuSeBMC* is a test generator for finding security vulnerabilities in C programs. In Test-Comp 2021, we described a previous version that incrementally injected labels to guide Bounded Model Checking (BMC) and Evolutionary Fuzzing engines to produce test cases for code coverage and bug finding. This paper introduces an improved version of *FuSeBMC* that utilizes both engines to produce smart seeds. First, the engines run with a short time limit on a lightly instrumented version of the program to produce the seeds. The BMC engine is particularly useful in producing seeds that can pass through complex mathematical guards. Then, *FuSeBMC* runs its engines with extended time limits using the smart seeds created in the previous round. *FuSeBMC* manages this process in two main ways. Firstly, it uses *shared memory* to record the labels covered by each test case. Secondly, it evaluates test cases, and those of high impact are turned into seeds for subsequent test fuzzing. In this year's competition, we participate in the *Cover-Error*, *Cover-Branches*, and *Overall* categories. The Test-Comp 2022 results show that we significantly increased our code coverage score from last year, outperforming all tools in all categories.

Keywords: Automated Test-Case Generation · Symbolic Execution · Bounded Model Checking · Fuzzing · Security · Seed.

1 Overview

Software testing is one of the most crucial phases in software development [11]. Tests often expose critical bugs in software applications. In earlier work [4], we presented *FuSeBMC*, an automated test generation tool that exploits the combination of Fuzzing and BMC. *FuSeBMC* achieved second place in Test-Comp 2021 [5,3] and first place in the *Cover-Error* category. It ranked fourth in the *Cover-Branches* category. This year, we introduce a new version of *FuSeBMC* (v4) that adds smart seed generation and shared memory amongst other improvements and features. The new version significantly improves on the previous version, particularly relating to code coverage. One of the primary contributions of this paper is the linking of a grey-box fuzzer with a bounded model checker. A bounded model checker works by treating a program as a state transition system and then checking whether there exists a transition in this system of length less than a bound k that violates the property to be verified [6,8]. We leverage

*Jury Member

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 336–340, 2022.
https://doi.org/10.1007/978-3-030-99429-7_19

this power of model checkers as a method for smart seed generation. Here, we rate seeds on two metrics. First, the depth of the deepest goal covered by the seed. Second, the number of goals covered uniquely by the seed. Seeds that rate highly on these metrics are called *smart*. During grey-box fuzzing, if a particular branch has not been explored, BMC can be used to provide a model (set of assignments to input variables) that reaches the branch. This model is a smart seed since it covers a previously unexplored branch. It is then added to a seed store. Periodically seeds are selected from the store for further grey-box fuzzing based on the criteria as mentioned above. However, BMC can be slow and resource-intensive. As an alternative, we also carry out a lightweight static program analysis to recognize certain restricted forms of input verification. We analyze the code for conditions on the input variables and ensure that seeds are only selected if they pass these conditions. Together, these contributions turn *FuSeBMC* into a world-class fuzzer.

2 Test Generation Approach

Figure 1 provides an overview of the components within *FuSeBMC* and how these interact. *FuSeBMC* makes use of the Clang tooling infrastructure [1] to instrument programs. In addition, *FuSeBMC* employs three engines in its reachability analysis: one BMC and two fuzzing engines. ESBMC [9,10] is a state-of-the-art SMT-based bounded model checker. For the two fuzzers, one is based on the American Fuzzy Lop (AFL) [7,2], and the other is a custom fuzzer, which we refer here to as *selective fuzzer* (see [4] for details). In the sections below, we detail how these components work together.

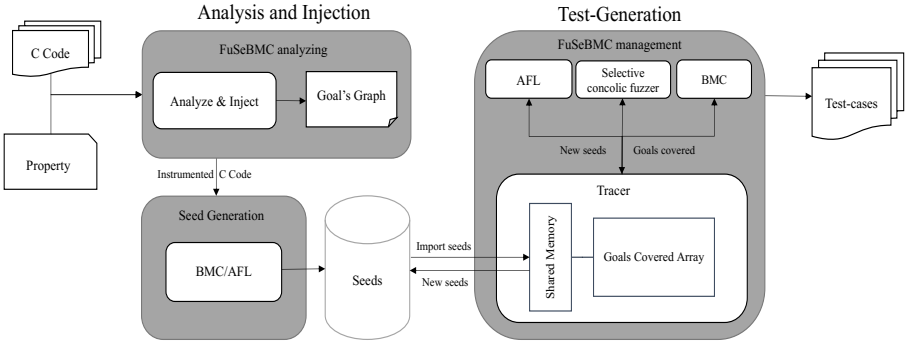


Fig. 1. *FuSeBMC* v4 Framework. This figure illustrates the major components of the *FuSeBMC* test generator and how they interact. Note in particular the seed store, which interacts with the BMC/AFL and the shared memory to produce test cases.

Code Instrumentation *FuSeBMC* front-end uses Clang tooling infrastructure [1] to parse a C program and produce an Abstract Syntax Tree (AST). While traversing the AST, *FuSeBMC* injects labels into each branch, including every conditional statement, loop, and function. Using these labels, *FuSeBMC* can measure the code coverage.

Reachability Graph Analysis After instrumenting the C program, *FuSeBMC* analyzes it and produces a reachability graph. The graph assigns each goal label to the code block it is located in. Then, *FuSeBMC* ranks goals depending on the strategy chosen. For example, one strategy, which we used in Test-Comp 2022, is to prefer deeper goals over

shallower goals. This strategy improves the performance of *FuSeBMC* since a test case that covers a deep goal will also cover shallower goals on the path to it. *FuSeBMC* also ranks coverage metrics over others, such as conditional coverage over loop coverage.

Seed Generation A unique aspect of the latest version of *FuSeBMC* is a seed generation phase that is run prior to the start of the principal reachability analysis. In this phase, *FuSeBMC* first lightly instruments the code under test by limiting loop bounds and assuming a narrow range of values for input variables. The bounds on input variables are further limited by carrying out a lightweight static analysis to recognize code that applies verification conditions to input variables. After instrumenting the code, *FuSeBMC* runs its fuzzing and BMC engines with concise time limits (60 s for Test-Comp 2022). The test cases generated by these engines are ranked, and the highest impact test cases are selected as smart seeds for the next round. The selected seeds are added to the *seed store*. The impact of a test case is measured using two metrics.

1. The number of labels covered uniquely by that test case.
2. The maximum program depth achieved by the test case.

ESBMC is particularly effective at seed generation as its underlying SMT solvers can be used to discover test cases that circumvent complex mathematical guards. Note that we do not rely on any specific features of the models returned by the SMT solvers. Instead, the strength of the method lies in the solvers' ability to return *some* model that can satisfy a guard and cover goals lying beyond. A fuzzer on its own, randomly mutating a seed, struggles to explore program sections occurring behind complex guards [12].

Reachability Analysis Engines In its primary phase, *FuSeBMC* carries out reachability analysis. Essentially, this involves running the engines in parallel with longer timeouts on the original, non-instrumented code with the fuzzer making use of the smart seeds. ESBMC is run using an incremental BMC strategy with some fixed time limit for each goal it attempts. *FuSeBMC*'s *Tracer* component coordinates the various engines through the use of *shared memory*. In this shared memory, we have two components. The first component is a "goals covered array" that stores the goals covered so far during the execution. Its purpose is to ensure there is no wasting effort through duplication of work. Secondly, the *Tracer* maintains a set of the currently most effective seeds for the fuzzer to use.

As the engines run and produce new test cases, the *Tracer* monitors these and evaluates them, adding those with the highest impact, as measured by the metrics above, to the seed store. Thus, the seed store is dynamically updated as the analysis progresses. Periodically, it selects a number of the most effective seeds from the store and adds them to shared memory for the fuzzers to use in their next fuzzing round. In parallel, ESBMC uses the "goals covered array" to select an as yet uncovered goal and attempts to find a test case that covers it. Test cases produced by ESBMC are passed directly to the store because they are likely to be beneficial for future fuzzing attempts.

For example, assume that the fuzzers are unable to cover some goal L due to a complex condition guarding it. ESBMC can be used to create a seed that covers L. This seed is then passed to the store and later selected for fuzzing. The fuzzers, armed with a seed that covers L, may well now be able to reach goals deeper than L along L's path. Thus, *FuSeBMC* combines the strengths of both types of engines. The BMC engine produces seeds that bypass complex guards and thereby help the fuzzers explore paths deep within the program.

3 Strengths and Weaknesses

The strengths of the latest version of *FuSeBMC* are as follows. It runs a dedicated seed generation phase to start the main fuzzing effort with high-quality, high-impact seeds. Furthermore, these seeds are constantly being updated during the main test-generation phase. Beyond this, it incorporates a dedicated subsystem, the *Tracer*, that uses a shared memory store to manage the various engines. By combining the engines, the *Tracer* ensures that *FuSeBMC* far outperforms the individual engines or even the running of the engines in parallel, but isolated. The outcome of these improvements can be seen in the ECA and Combination benchmark sets. Previously, these posed a challenge to *FuSeBMC*. With the latest changes, *FuSeBMC* achieved first place in the Combination subcategory and took second place in the ECA subcategory of the 2022 Test-Comp competition. Since the benchmarks in the ECA category have remained stable between last year's and this year's competitions, we can measure FuSeBMC's improvement in terms of the combined coverage it achieves across the 29 tasks. This improvement stands at a remarkable 60%. The 2022 Test-Comp results also show that *FuSeBMC* has achieved first place in the *Cover-Branches* category with high coverage and validation statistics. However, one of the weaknesses of *FuSeBMC* that we plan to work on is that for large programs, particularly for programs that redefine C library functions, seed generation can be slow and consume too much of the tool's time.

4 Tool Setup and Configuration

FuSeBMC can be run using the command below. The user is required to set the architecture, the property file path, the competition strategy, and the benchmark path, as:

```
fusebmc.py [-a {32, 64}] [-p PROPERTY_FILE]
               [-s {kinduction,falsi,incr,fixed}]
               [BENCHMARK_PATH]
```

where `-a` sets the architecture to 32 or 64, `-p` sets the property file to `PROPERTY_FILE`, where it has a list of all the properties to be tested. `-s` sets the BMC strategy to one of the listed strategies`{kinduction,falsi,incr,fixed}`. For Test-Comp'22, *FuSeBMC* uses `incr` for incremental BMC, which relies on the ESBMC's symbolic execution engine to increasingly unwind the program loops using an iterative technique. The `incr` strategy verifies the program for each unwind bound up to a maximum default value of 50 or indefinitely (until it exhausts the time or memory limits). The Benchexec tool info module is `fusebmc.py` and the benchmark definition file is `FuSeBMC.xml`.

5 Software Project

FuSeBMC is implemented using C++, and it is publicly available under the terms of the MIT License at GitHub[1]. The repository includes the latest version of *FuSeBMC* (version 4.1.14). *FuSeBMC* dependencies and instructions for building from source code are all listed in the `README.md` file. Test-Comp 2022 provides the script, benchmarks, and *FuSeBMC* binary to reproduce the competition's results[2].

[1] https://github.com/kaled-alshmrany/FuSeBMC
[2] https://test-comp.sosy-lab.org/2022/

Acknowledgment

The Institute of Public Administration - IPA - Saudi Arabia [1] supports the FuSeBMC development. The work in this paper is also partially funded by the EPSRC grants EP/T026995/1, EP/V000497/1, EU H2020 ELEGANT 957286, and Soteria project awarded by the UK Research and Innovation for the Digital Security by Design (DSbD) Programme.

References

1. Clang documentation. http://clang.llvm.org/docs/index.html.
2. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.
3. Kaled Alshmrany et al. FuSeBMC: A white-box fuzzer for finding security vulnerabilities in C programs. In FASE, pages 363–367, 2021.
4. Kaled Alshmrany et al. FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. International Conference on TAP, pages 85-105, 2021.
5. Beyer, D.: Status report on software testing: Test-Comp 2021. In FASE, pages 341–357, 2021.
6. Armin Biere. Bounded model checking. Frontiers in Artificial Intelligence and Applications. In Handbook of satisfiability, pages 457–481, 2009.
7. Böhme et al. Directed greybox fuzzing. In CCS, pages 2329–2344, 2017.
8. Lucas C. Cordeiro et al. SMT-Based bounded model checking for embedded ANSI-C software. IEEE Trans. Software Eng. 38(4): 957–974, 2012.
9. Gadelha, M.R. et al. ESBMC: scalable and precise test generation based on the floating-point theory:(Competition Contribution). In FASE, pages 525–529, 2020.
10. Gadelha, M.R. et al. ESBMC v6.0: verifying C programs using *k*-induction and invariant inference - (Competition Contribution). In TACAS, pages 209–213, 2019.
11. Nicha Kosindrdecha and Jirapun Daengdej: A test case generation process and technique. Journal of Software Engineering, 4(4):265-287, 2010.
12. Stephens, Nick et al. Driller: Augmenting fuzzing through selective symbolic execution. In NDSS, pages 1–16, 2016.

[1] https://www.ipa.edu.sa/en-us/Pages/default.aspx

VeriFuzz: Good Seeds for Fuzzing (Competition Contribution)

Ravindra Metta, Raveendra Kumar Medicherla*(✉), and Hrishikesh Karmarkar

TCS Research, Tata Consultancy Services, India
r.metta,raveendra.kumar,hrishikesh.karmarkar@tcs.com

Abstract. We present VeriFuzz 1.2 with two new enhancements: (1) unroll the given program to a short depth and use BMC to produce *incomplete* test inputs, which are extended into *complete* inputs, and (2) if BMC fails for this short unrolling, automatically identify the reason and rerun BMC with a corresponding remedial strategy.

Keywords: Coverage Guided Fuzzing · Bounded Model Checking · Scalable Model Checking

1 Introduction

VeriFuzz 1.0 [5] is an automated test input generation tool built on top of AFL [11], a Coverage Guided Fuzzing (CGF) engine, and the PRISM [7] program analysis framework. CGF requires initial test inputs (seeds) to generate newer inputs in order to build a test suite for coverage. In VeriFuzz 1.0, the seeds are generated as follows: (1) random seeds are either generated dynamically, or picked from a small set of unbiased inputs, and (2) for sequentialized concurrent programs that have deep nesting, generate test inputs using BMC by unrolling all the loop bodies once. However, a direct application of CGF or BMC to a given program may not yield required coverage as CGF may get *stuck* in "complex conditions" [10], while BMC does not scale well for programs with "complex loops". To address these issues, we implemented two key enhancements to VeriFuzz 1.0. The first is to generate *incomplete* test seeds using BMC and complete them using random inputs. The second is to automatically identify the cause if BMC fails, and re-run BMC with an appropriate *remedial strategy*. These enhancements, implemented in VeriFuzz 1.2 [8], are described below.

1.1 Enhancement 1 : New Seed Generation Approach

Instead of generating a complete test seed using BMC, which scales poorly, for a given program P, we use CBMC [6] to generate an incomplete program P_u by unwinding P only to a "short" depth d, which is heuristically guessed

* Jury member

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 341–346, 2022.
https://doi.org/10.1007/978-3-030-99429-7_20

to be small enough for BMC to scale. This short unwinding makes rest of P unreachable due to the incompleteness of the unwinding. But it allows BMC to scale much better to P_u. We then use CBMC to produce test input sequences that cover the branches of P_u, using the cbmc options "–cover branches". Each of these inputs forms a *valid* prefix of a *complete* test input for P. We denote the set of such prefixes with T_p. When P is executed with a prefix t_p in T_p, P may still require additional inputs to complete it's execution. We randomly generate such additional inputs, from the value ranges respecting the input types. We append each of these random inputs to the corresponding t_p to form a *complete* input for P. Our experimentation showed that this approach helped VeriFuzz 1.2 to cover many more branches, which could not be covered with VeriFuzz 1.0.

1.2 Enhancement 2: Remedying A Stuck or Failed BMC

We observed that often times, even for short unwindings on complex programs, BMC either gets stuck (i.e., does not terminate in the given time budget) or fails with some error. We investigated this problem and found that BMC may get stuck/fail in any of its internal phases during the translation of a given program into a SAT/SMT formula. Some times the formula gets generated, but the backend SAT/SMT solver times out due to the complexity of the formula. Some of the common BMC failure causes and the remedial actions are:-

1. *Large number of unwindings due to loops or recursive calls:* As each unwinding causes an exponential increase in the formula size, BMC may get stuck in unwinding the program even for an unwinding depth as small as 10. In such cases, we rerun BMC with an even smaller unwinding. If the BMC still gets stuck in unwinding, instead of trying to generate a formula for the entire program, we try to generate one formula per path and solve each such formula separately. CBMC supports this with the option "–paths".
2. *Large arrays:* Large arrays require too many boolean variables (equal to the number of bits) to encode them into a SAT formula, which requires too much memory and solving time. In such cases, to use a SAT solver backend, we use the CBMC option "–arrays-uf-always" that translates arrays as uninterpreted functions, thereby avoiding the bit blasting. Alternatively, depending on the program features, we use the Z3 SMT solver [9] as it supports array theory and hence does not require bit blasting of arrays.
3. *Quadratic constraints*: To ensure functional consistency of translation of features such as array indexing operations, techniques like Ackermann expansion are used, which lead to a quadratic number of constraints. In some cases, this causes more than a billion constraints to be added to the SAT formula, and BMC gets stuck in adding these constraints. One remedy in such cases is to abstract out the array operations, for example by havocing the enclosing functions. This remedy is currently under investigation.
4. *Timeout trap*: Sometimes, the SAT formula generation goes through, but the solver times out. We trap the timeout, and output the test inputs corresponding to the goals that have already been covered.

2 Tool Architecture and Flow

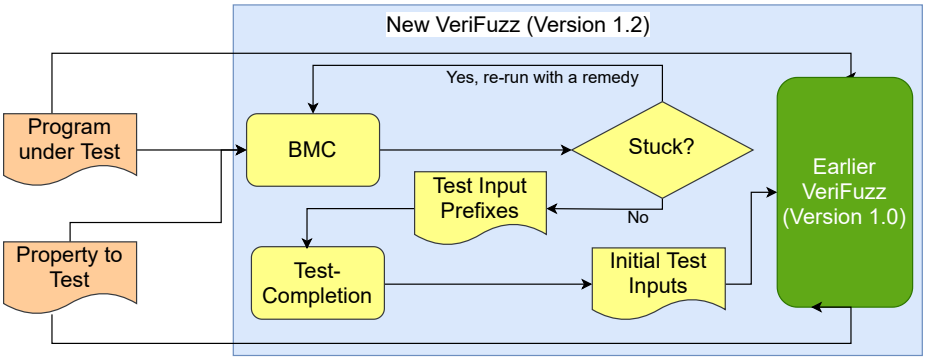


Fig. 1. VeriFuzz architecture.

Figure 1 shows the architecture of VeriFuzz 1.2. The yellow boxes show the enhancements to VeriFuzz 1.0. For BMC, we use CBMC 5.42.0, with z3 4.8.12 and Glucose Syrup 4.0 [1]. VeriFuzz 1.2 takes two inputs: (1) a program to test, say P, and (2) a property to test, such as branch or error coverage. First, the module "BMC" invokes CBMC on P for a short unwinding, typically with a timeout of 1 minute, to generate the incomplete test inputs. If CBMC times out, then: if any of the incomplete test inputs have been generated by CBMC, then output those, else identify the phase where CBMC is stuck and re-run CBMC with a corresponding remedial strategy. Then, the "Test-Completion" module extends these incomplete test inputs to form *complete* test inputs. These complete tests are then passed to VeriFuzz 1.0, which fuzzes them using AFL to produce more test inputs.

3 Strengths and Weaknesses

VeriFuzz 1.0 was enhanced with minor optimizations into VeriFuzz 1.1, which does not contain Enhancement 1 and Enhancement 2 (see Sec. 1). VeriFuzz 1.1 participated in Test-Comp 2020 [2], while VeriFuzz 1.2 participated in Test-Comp 2021 [3] and Test-Comp 2022 [4]. Here, we compare VeriFuzz 1.2's results against VeriFuzz 1.1's results, for all the categories common to Test-Comp 2022 and 2020, except ECA (to avoid any bias due to the fixed-seeds).

Performance: In Cover-Error, VeriFuzz 1.2 detected 93% of the errors (387 out of 415) with an average time of 17 seconds, while VeriFuzz 1.1 detected 91% (262 out of 287) of the errors with an average time of 33 seconds. In Cover-Branches, VeriFuzz 1.2 covered 68% (scored 1626 out of 2378)) of the branches, with an average of 14.7 minutes per benchmark, while VeriFuzz 1.1

covered 59% (scored 872 out of 1485) of the branches, with an average of 13.3 minutes per benchmark. The higher time taken by VeriFuzz 1.2 directly corresponds to the increase in coverage. On device drivers in BusyBox (MemSafety) and LDV(ReachSafety) VeriFuzz 1.2 scored substantially better: 29 of 75 and 57 of 290 respectively, while VeriFuzz 1.1 scored only 19 of 72 and 23 of 290 respectively.

Usefulness of the enhancements: We analyzed the results of VeriFuzz 1.2, and noticed that in some cases Enhancement 1 was sufficient, while Enhancement 2 was also necessitated in other cases. For instance, in loop-floats-scientific-comp/loop2-1.c and ntdrivers-simplified/cdaudio_simpl1.cil-1.c, VeriFuzz 1.1 was unable to detect the error, which VeriFuzz 1.2 could detect with the seeds generated by Enhancement 1 alone. In cases like array-examples/sorting_bubblesort_2_ground.i and array-tiling/mlceu.c, Enhancement 2 was also required, in addition to Enhancement 1, to generate a seed that allowed VeriFuzz 1.2 to detect the error, which VeriFuzz 1.1 could not. In Cover-Branches, on benchmarks like loop-industry-pattern/mod3.c (2 seeds generated by BMC), and bitvector/s3_srvr_1a_alt.BV.c.cil.c (73 seeds generated by BMC), VeriFuzz 1.2 could cover 90% of the branches, while VeriFuzz 1.1 could cover none.

Weaknesses: (1) In some cases, e.g. array-multidimensional/copy-2-u.c, BMC is running out of memory, leading to the termination of entire VeriFuzz process. In some other cases, e.g. float-benchs/inv_square-1.c, the floating point interpretation mismatches between CBMC and VeriFuzz lead to unintended behavior. These issues indicate that our tooling needs improvement. (2) In Arrays subcategory of Cover-Error, VeriFuzz 1.2 took more than twice the time of VeriFuzz 1.1. This is because, many array benchmarks contain *for-loops* that iterate over large arrays. In such cases, *short* unwindings of BMC do not go past the array initialization itself and hence the seeds generated by BMC were ineffective, adding to the elapsed time.(3) In BusyBox and LDV drivers, there are many benchmarks that VeriFuzz is unable to solve due to issues like complex loops and quadratic constraints, which we are currently working on.

4 VeriFuzz Tool Configuration and Setup

The tool is available at git@gitlab.com:sosy-lab/test-comp/archives-2022.git. To install and run the tool, follow the instructions in the README.txt provided with the tool. The `benchexec` tool-info module is verifuzz.py and the benchmark description file is verifuzz.xml. A sample run command is as follows: `./scripts/verifuzz.py --propertyFile coverage-error.prp example.c`. In 2022, VeriFuzz 1.2 participated in *Cover-Branches* and *Cover-Error* categories.

5 Software Project and Contributors

VeriFuzz is developed and maintained by the authors at TCS Research. They can be contacted at `VeriFuzz.Tool@tcs.com`. We thank everyone who has contributed to VeriFuzz, AFL, PRISM, CBMC, Glucose Syrup and Z3.

References

1. Audemard, G., Simon, L.: On the glucose SAT solver. Int. J. Artif. Intell. Tools **27**(1), 1840001:1–1840001:25 (2018). https://doi.org/10.1142/S0218213018400018
2. Beyer, D.: Second competition on software testing: Test-comp 2020. In: Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020, Dublin, Ireland, April 25-30). pp. 505–519. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_25
3. Beyer, D.: Status report on software testing: Test-Comp 2021. In: Proceedings of the 24th International Conference on Fundamental Approaches to Software Engineering (FASE 2021, Luxembourg, Luxembourg, March 27 - April 1). pp. 341–357. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_17
4. Beyer, D.: Advances in automaticl software testing: Test-Comp 2022. In: Proceedings of the 25th International Conference on Fundamental Approaches to Software Engineering - 25th International Conference, (FASE 2022, Munich, Germany, April 2-7, 2022). LNCS 13241, Springer (2022)
5. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: Verifuzz: Program aware fuzzing - (competition contribution). In: Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019, Prague, Czech Republic, April 6-11). pp. 244–249. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22
6. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004, Barcelona, Spain, March 29 - April 2). pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
7. Khare, S., Saraswat, S., Kumar, S.: Static program analysis of large embedded code base: an experience. In: Proceeding of the 4th Annual India Software Engineering Conference, (ISEC 2011, Thiruvananthapuram, India, February 24-27). pp. 99–102. ACM (2011). https://doi.org/10.1145/1953355.1953368
8. Metta, R., Medicherla, R.K., Chakraborty, S.: Bmc+fuzz : Efficient and effective test generation to appear. In: Design, Automation & Test in Europe Conference & Exhibition, (DATE 2022, Antwerp, Belgium, March 14-23). IEEE (2022)
9. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008, Budapest, Hungary, March 29 - April 6). pp. 337–340. LNCS 4963, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
10. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: 23rd Annual Network and Distributed System Security Symposium, (NDSS 2016, San Diego, California, USA, February 21-24). The Internet Society (2016). https://doi.org/10.14722/ndss.2016.23368
11. Zalewski, M.: American fuzzy lop, http://lcamtuf.coredump.cx/afl/

Author Index