

Catherine Dubois
Burkhart Wolff (Eds.)

LNCS 10889

Tests and Proofs

12th International Conference, TAP 2018
Held as Part of STAF 2018
Toulouse, France, June 27–29, 2018, Proceedings



Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, Lancaster, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Zurich, Switzerland

John C. Mitchell

Stanford University, Stanford, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

C. Pandu Rangan

Indian Institute of Technology Madras, Chennai, India

Bernhard Steffen

TU Dortmund University, Dortmund, Germany

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbrücken, Germany

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

Catherine Dubois · Burkhart Wolff (Eds.)

Tests and Proofs

12th International Conference, TAP 2018
Held as Part of STAF 2018
Toulouse, France, June 27–29, 2018
Proceedings

Editors
Catherine Dubois
ENSIE
Evry
France

Burkhart Wolff
Université Paris-Sud
Gif sur Yvette
France

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-92993-4 ISBN 978-3-319-92994-1 (eBook)
<https://doi.org/10.1007/978-3-319-92994-1>

Library of Congress Control Number: 2018944416

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer International Publishing AG, part of Springer Nature 2018

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

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

Foreword

Software Technologies: Applications and Foundations (STAF) is a federation of leading conferences on software technologies. It provides a loose umbrella organization with a Steering Committee that ensures continuity. The STAF federated event takes place annually. The participating conferences and workshops may vary from year to year, but they all focus on foundational and practical advances in software technology. The conferences address all aspects of software technology, from object-oriented design, testing, mathematical approaches to modeling and verification, transformation, model-driven engineering, aspect-oriented techniques, and tools. STAF was created in 2013 as a follow-up to the TOOLS conference series that played a key role in the deployment of object-oriented technologies. TOOLS was created in 1988 by Jean Bézivin and Bertrand Meyer and STAF 2018 can be considered its 30th birthday.

STAF 2018 took place in Toulouse, France, during June 25–29, 2018, and hosted: five conferences, ECMFA 2018, ICGT 2018, ICMT 2018, SEFM 2018, and TAP 2018, and the Transformation Tool Contest TTC 2018; eight workshops and associated events. STAF 2018 featured seven internationally renowned keynote speakers, welcomed participants from all around the world and had the pleasure to host a talk by the founders of the TOOLS conference, Jean Bézivin and Bertrand Meyer.

The STAF 2018 Organizing Committee would like to thank (a) all participants for submitting to and attending the event, (b) the Program Committees and Steering Committees of all the individual conferences and satellite events for their hard work, (c) the keynote speakers for their thoughtful, insightful, and inspiring talks, and (d) Ecole Nationale Supérieure d'Electrotechnique, Electronique, Hydraulique et Télécommunications (ENSEEIH), Institut National Polytechnique de Toulouse (Toulouse INP), Institut de Recherche en Informatique de Toulouse (IRIT), Occitanie, and all sponsors for their support. A special thanks goes to all the members of the Software and System Reliability Department of the IRIT Laboratory and the members of the INP-Act SAIC, coping with all the foreseen and unforeseen work so as to prepare a memorable event.

June 2018

Marc Pantel
Jean-Michel Bruel

Preface

This volume contains the papers presented at TAP 2018, the 12th International Conference on Tests and Proofs. The TAP conference promotes research in verification and formal methods that target the interplay of proofs and testing: the advancement of techniques of each kind and their combination, with the ultimate goal of improving software and system dependability. As in the five previous editions, TAP 2018 was part of STAF (Software Technologies: Applications and Foundations), a federation of leading conferences in software technology.

TAP 2018 took place in Toulouse in France during June 27–29, 2018. The Program Committee (PC) received 18 paper submissions, each reviewed by three PC members. After a lively discussion and careful deliberation, we selected ten contributions (seven regular papers, one tool demonstration paper, and two short papers) for inclusion in this proceedings volume and presentation at the conference. The combination of topics highlights how testing and proving are increasingly seen as complementary rather than mutually exclusive techniques, and confirms TAP’s commitment to bringing together researchers and practitioners from both areas of verification.

The program of TAP was nicely completed by a keynote talk by Dirk Beyer (Ludwigs-Maximilians-Universität München, Germany), who also contributed an invited paper for this volume, and a tutorial by Sébastien Bardin and Nikolai Kosmatov (CEA List, France). We would like to thank the invited speakers for contributing an exciting presentation to the participants of STAF 2018.

We also thank the PC members and the additional reviewers for their timely and thorough reviewing work, and for contributing to an animated and informed discussion. Their names are listed on the following pages. The EasyChair system provided flawless technical support.

The organization of STAF made for a successful and enjoyable conference in a wonderful location. We thank all the organizers, and in particular the general chair, Marc Pantel, and the organization chair, Jean-Michel Bruel, for their hard work; and we also thank ENSEEIHT (Ecole Nationale Supérieure d’Electrotechnique, Electronique, Hydraulique et Télécommunications) for hosting us.

June 2018

Catherine Dubois
Burkhart Wolff

Organization

Program Committee

Bernhard K. Aichernig	TU Graz, Austria
Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Jasmin Christian Blanchette	Vrije Universiteit Amsterdam, The Netherlands
Achim D. Brucker	The University of Sheffield, UK
Catherine Dubois (Co-chair)	ENSIIE-Samovar, France
Carlo A. Furia	Chalmers University of Technology, Sweden
Angelo Gargantini	University of Bergamo, Italy
Alain Giorgetti	FEMTO-ST Institute, University of Bourgogne Franche-Comté, France
Martin Gogolla	Database Systems Group, University of Bremen, Germany
Arnaud Gotlieb	SIMULA Research Laboratory, Norway
Klaus Havelund	Jet Propulsion Laboratory, USA
Rob Hierons	Brunel University, UK
Reiner Hähnle	TU Darmstadt, Germany
Moa Johansson	Chalmers University of Technology, Sweden
Thierry Jéron	Inria, France
Chantal Keller	LRI, Université Paris-Sud, France
Nikolai Kosmatov	CEA List, France
Laura Kovacs	Vienna University of Technology, Austria
Tanja Mayerhofer	Vienna University of Technology, Austria
Karl Meinke	KTH Royal Institute of Technology, Sweden
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Alexandre Petrenko	CRIM, Canada
Martina Seidl	Johannes Kepler University Linz, Austria
Helene Waeselynck	LAAS-CNRS, France
Burkhard Wolff (Co-chair)	LRI, Université Paris-Sud, France

Additional Reviewers

Arnaud, Mathilde	Marcozzi, Michaël
Avellaneda, Florent	Nguena Timo, Omer
Bannour, Boutheina	Paskevich, Andrei
Bubel, Richard	Smallbone, Nicholas
Desai, Nisha	Steinhöfel, Dominic
Lochbihler, Andreas	Voisin, Frederic

Specify and Measure, Cover and Unmask: A Proof-Friendly View of Test Coverage Criteria (Abstract of Invited Tutorial)

Sébastien Bardin and Nikolai Kosmatov

CEA, List, Software Reliability and Security Laboratory, PC 174, 91191
Gif-sur-Yvette, France
{Sébastien.Bardin,Nikolai.Kosmatov}@cea.fr

Tutorial Abstract. Automatic test data generation (ATG) is a major topic in software engineering. A large amount of research effort has been invested to automate white-box testing. While a wide range of different and sometimes heterogeneous code-coverage criteria have been proposed, testing techniques still lack a generic formalism to describe them all, and available test automation tools usually support only a small subset of them.

This tutorial brings participants to a journey into the world of white-box testing criteria and their automated support. We try to give a convenient proof-friendly view of coverage criteria and to bridge the gap between the coverage criteria supported by state-of-the-art white-box ATG technologies, such as Dynamic Symbolic Execution, and advanced coverage criteria found in the literature. The tutorial is articulated around *labels*, a recent specification mechanism for test objectives, and their *effective support in automated testing tools*. Labels are generic enough to **specify** many common testing criteria, and amenable to efficient automation making it possible to **cover** the corresponding test objectives and to **measure** the coverage level of a given test suite. We propose several optimization techniques resulting in an effective support for labels in ATG tools. We also show how a combination of static analysis techniques can be efficiently applied to detect — **unmask** — infeasible test objectives that are responsible for waste of test generation effort and imprecise coverage measurement. We demonstrate the LTest toolset for test automation with efficient support of labels. Finally, we present a recent *extension of labels*, called HTOL (Hyperlabel Test Objectives Language), capable to encode even most advanced test coverage criteria (such as variants of MCDC, dataflow criteria, non-interference properties, etc.), including *hyperproperties*. This tutorial is based on a series of recent research and tool implementation efforts [1–6].

References

1. Bardin, S., Kosmatov, N., Cheynier, F.: Efficient leveraging of symbolic execution to advanced coverage criteria. In: ICST (2014)
2. Bardin, S., Chebaro, O., Delahaye, M., Kosmatov, N.: An all-in-one toolkit for automated white-box testing. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570. Springer, Cham (2014)
3. Bardin, S., Delahaye, M., David, R., Kosmatov, N., Papadakis, M., Traon, Y.L., Marion, J.: Sound and quasi-complete detection of infeasible test requirements. In: ICST (2015)
4. Marcozzi, M., Delahaye, M., Bardin, S., Kosmatov, N., Prevosto, V.: Generic and effective specification of structural test objectives. In: ICST (2017)
5. Marcozzi, M., Bardin, S., Delahaye, M., Kosmatov, N., Prevosto, V.: Taming coverage criteria heterogeneity with LTest. In: ICST (2017)
6. Marcozzi, M., Bardin, S., Kosmatov, N., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: ICSE. ACM (2018, to appear)

Contents

Invited Contribution

- Tests from Witnesses: Execution-Based Validation of Verification Results . . . 3
*Dirk Beyer, Matthias Dangl, Thomas Lemberger,
and Michael Tautschnig*

Regular Contributions

- An Approximation-Based Approach for the Random Exploration
of Large Models 27
Julien Bernard, Pierre-Cyrille Héam, and Olga Kouchnarenko
- Static and Dynamic Verification of Relational Properties on
Self-composed C Code 44
*Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, Virgile Prevosto,
and Guillaume Petiot*
- Under-Approximation Generation Driven by Relevance Predicates
and Variants 63
J. Julliard, O. Kouchnarenko, P.-A. Masson, and G. Voiron
- Using Dependence Graphs to Assist Verification and Testing
of Information-Flow Properties 83
Mihai Herda, Shmuel Tyszberowicz, and Bernhard Beckert
- Tactic Program-Based Testing and Bounded Verification in Isabelle/HOL . . . 103
Chantal Keller
- Verification Coverage for Combining Test and Proof. 120
*Viet Hoang Le, Loïc Correnson, Julien Signoles,
and Virginie Wiels*
- Detection of Security Vulnerabilities in C Code Using Runtime
Verification: An Experience Report 139
Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles

Tool Demonstration and Short Papers

- Formalizing (Web) Standards: An Application of Test and Proof 159
Achim D. Brucker and Michael Herzberg

Automated Test Case Generation for Java EE Based Web Applications	167
<i>Andreas Fuchs</i>	
Ghosts for Lists: From Axiomatic to Executable Specifications	177
<i>Frédéric Louergue, Allan Blanchard, and Nikolai Kosmatov</i>	
Author Index	185

Invited Contribution



Tests from Witnesses

Execution-Based Validation of Verification Results

Dirk Beyer¹ , Matthias Dangl¹ , Thomas Lemberger¹ ,
and Michael Tautschnig² 

¹ LMU Munich, Munich, Germany

² Queen Mary University of London, London, UK

Abstract. The research community made enormous progress in the past years in developing algorithms for verifying software, as shown by international competitions. Unfortunately, the transfer into industrial practice is slow. A reason for this might be that the verification tools do not connect well to the developer work-flow. This paper presents a solution to this problem: We use verification witnesses as interface between verification tools and the testing process that every developer is familiar with. Many modern verification tools report, in case a bug is found, an error path as exchangeable verification witness. Our approach is to synthesize a test from each witness, such that the developer can inspect the verification result using familiar technology, such as debuggers, profilers, and visualization tools. Moreover, this approach identifies the witnesses as an interface between formal verification and testing: Developers can use arbitrary (witness-producing) verification tools, and arbitrary converters from witnesses to tests; we implemented two such converters. We performed a large experimental study to confirm that our proposed solution works well in practice: Out of 18 966 verification results obtained from 21 verifiers, 14 727 results were confirmed by witness-based result validation, and 10 080 of these results were confirmed alone by extracting and executing tests, meaning that the desired specification violation was effectively observed. We thus show that our approach is directly and immediately applicable to verification results produced by software verifiers that adhere to the international standard for verification witnesses.

1 Introduction

Automatic software verification, i.e., using methods from program analysis and model checking to find out whether a program satisfies or violates a given specification, is a successful and mature technology. The efficiency and effectiveness of the available verification tools for C programs is shown in the annual competition on software verification [5]. Despite this success story in research, the state-of-the-art in practice is that not many software projects have such verification tools incorporated into their software-development process. The reason for this gap between availability of technology on the one side and missed opportunities on the other side is perhaps twofold: (a) developers are frustrated by false alarms, i.e.,

in the past, static analyzers reported too many bugs that were not observable in a concrete program execution, and thus, developers have lost confidence in bug reports [20]; (b) there is a lack of appropriate interfacing, i.e., it is difficult for developers to leverage advantages of the verification tools because they are difficult to integrate and difficult to learn from [1].

To overcome these two problems, we propose (i) to use verifiers that produce verification witnesses, i.e., abstract descriptions of one or more paths to a specification violation (many such tools are already available¹), and (ii) to validate whether a real bug has been found by constructing a test from the produced verification witness and observing the execution of that test. This way, issue (a) above is solved because, if the test execution does show and thus confirm the reported specification violation, the verification result can be examined with high confidence and on a concrete, executable example (e.g., with a debugger), and issue (b) is solved because we bridge the gap between the, in most projects, unfamiliar domain of verification and the established domain of testing, which makes it easier to integrate verification into the development process.

Execution-Based Validation of Witnesses. Witness validation based on model-checking technology works well [4, 5, 9, 14], but the disadvantage is that due to over-approximation, the validation might be as imprecise as the verification step. A verification witness serves as a (potentially coarse) description of a part of the state space of a program that contains a specification violation, and the witness validators can confirm or reject the error report. We complement the witness-validation technology by direct test execution: A test case (e.g., unit-test code) is built from the violation witness, and this test case provides a precise and transparent way to confirm and examine it.² By observing and analyzing an execution that exposes undesirable behavior, developers can convince themselves that the error report is correct, and address the reported bugs without the risk of wasting time on a false alarm. If the execution does not violate the specification, the witness might have represented a false alarm and the developer can assign a lower priority to that report.

Witnesses as Communication Interface. One barrier for the adoption of verification technology is that developers have to spend considerable time on understanding a verification tool and on becoming familiar with it. Thus, we have to avoid the “lock-in” effect: people might not want to decide for one particular tool if they have to invest time again when they wish to change the decision later. If the developer constructs the integration on top of the exchangeable verification witnesses, i.e., using the witnesses as interface to the verification tools, the verification tool is exchangeable without any change to the testing process.³

¹ <https://sv-comp.sosy-lab.org/2017/systems.php>

² It has been shown that model checkers can be effective in constructing useful tests [12].

³ At least 21 verifiers are available that produce witnesses in the exchangeable format (cf. Table 1, which lists the verifiers that we use in our experiments).

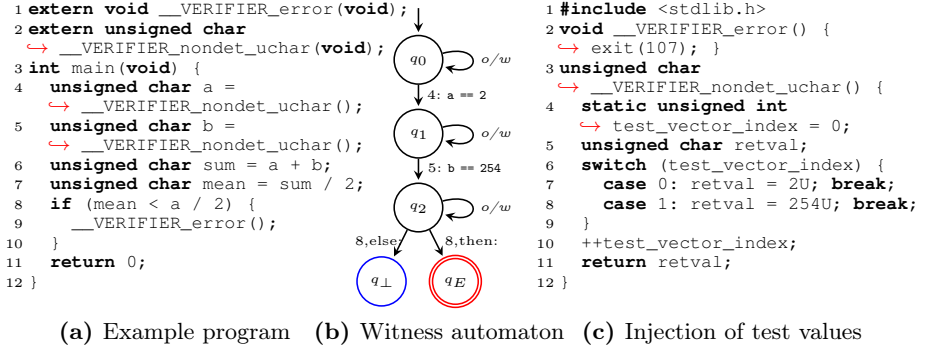


Fig. 1. An incorrect example C program (a), the corresponding violation witness produced by the verifier (b), and a code fragment used to inject the extracted test values for compilation (c)

Tests from Witnesses. In order to flexibly bridge the gap from witness to test, we provide two independently developed implementations of tools that take as input a program and a violation witness, and synthesize a test that is compilable and executable. This approach provides the following three features: (1) the result of a verification tool can be validated by compiling and executing the corresponding test—if the test violates the specification, the verification tool reported a correct alarm and the result can be handled appropriately; (2) the synthesized unit tests can be stored and maintained together with the other unit tests, but can also be re-constructed at any time on demand; (3) independently from the verification tool that produced the witness, the full repertoire for inspecting a failing program—such as debuggers, profilers, and visualization tools—can be used by the developer to understand the bug that the test represents.

Experimental Study. To evaluate our proposal, we performed experiments on thousands of witnesses. We took many C programs from the largest public repository of verification tasks and many witness-producing verification tools, and collected 13 200 witnesses of specification violations. We obtained another 5 766 refined witnesses using witness refinement, a procedure introduced in the original work on verification witnesses [9]. This technique is supposed to refine witnesses to be more concrete, so we should be able to generate better test cases from them. In conjunction with the two existing validators, CPACHECKER and ULTIMATE AUTOMIZER, our method significantly increases the confirmation rate: out of the total of 18 966 witnesses, we were able to extract test cases for 10 080 of them, meaning that we successfully created and executed the tests, and the specification violation was observed. Using the new approach, we increased the confirmed results from 12 821 to 14 727 in total.

Example. In the following, we illustrate the complete process from running a verification task using a verifier through synthesizing the test code from the violation witness to compiling the program and executing it.

Figure 1a shows a program that attempts to calculate the mean of two integer numbers, a computation that is often required in binary-search algorithms. In lines 4 and 5, two variables `a` and `b` of type `unsigned char`⁴ are initialized nondeterministically, for example from user input. The subsequent lines are supposed to calculate the mean of the two variables, by first computing their sum in line 6 and then dividing it by 2 in line 7. If the mean of `a` and `b` has been calculated correctly, it must not be less than half of either of the two values. This condition is asserted in lines 8 to 10. We can check whether the condition is satisfied by specifying that the function `_VERIFIER_error()` must not be reachable, and then running a verifier on this verification task. The verifier should detect and report that the assertion will be violated if the sum of `a` and `b` exceeds the range of the data type `unsigned char`, causing an overflow. Figure 1b shows a violation-witness automaton [9] that represents a counterexample to the specification. The automaton specifies that if we assume that `a` is assigned the value 2 in line 4 and `b` is assigned the value 254 in line 5, control will flow to the `then`-branch in line 8, causing a violation of the specification. To independently validate this witness, we can then extract the input values for `a` and `b`, and use them to provide an implementation of the input function `_VERIFIER_nondet_uchar()` and the `_VERIFIER_error()` function as depicted in Fig. 1c. After compiling Fig. 1a and 1c into an executable and running it, we can confirm that these input values trigger the call to `_VERIFIER_error()` by checking its return code. We can even use a debugger such as GDB to step through the compiled program and observe the faulty behavior directly. The debugger will show that the sum of `a` and `b`, respectively 2 and 254, computed in line 6 wraps around to 0. Therefore, the mean is incorrectly calculated as 0 in line 7. The condition in line 8 then evaluates to 1, because 0 is smaller than 1.

It must be noted that the witness depicted in Fig. 1b is very precise: it provides a concrete counterexample with explicit values for `a` and `b`. But in general, a violation witness may simply describe a part of the state space that contains a specification violation, i.e., an abstract counterexample. Suppose a verifier is only able to provide a witness that specifies that if `a + b` is greater than 255 in line 6, the specification will be violated. By using witness refinement [9], we can obtain from this abstract witness a concrete witness like Fig. 1b.

Contributions. Our approach features the following advantages:

- Verification tools sometimes produce false alarms, which can lead to severe waste of investigation time. We synthesize tests from verification witnesses, and consequently trust only verification results confirmed by test execution.
- There are several witness-based validators available, but our execution-based validation of the error path can be more precise and more efficient, compared to the previously available validators.
- Avoidance of technology lock-in: A developer’s work flow does not depend on a particular choice of verification tool, because the developer’s infrastructure hooks in at the witness. The developer may elect to use a different verifier, or even use multiple verifiers simultaneously—at no additional cost.

⁴ The example also works for larger data types, but for ease of presentation, we aim to keep the range of values small, so that all calculations can be followed by hand.

- Compared to working with witnesses, developers are more familiar with tests, and more supporting tools—such as profilers, memory analyzers, and visualization tools—are available to analyze the tests that correspond to the witnesses.
- The newly generated tests can complement the existing test suite, and the tests as well as the witnesses can be stored and maintained as first-class objects in the software life cycle.

Related Work. Our approach is based on a number of existing ideas, which we outline in the following.

Verification Witnesses. We build our contributions on top of existing work on violation witnesses [9], which we will describe in more detail in the background section. The problem that verification results are not treated well enough by the developers of verification tools is known and there are also other works that address the same problem, for example, the work on execution reports [18].

Test-Case Generation. The idea to generate test cases from verification counterexamples is more than ten years old [6, 48], has since been used to create debuggable executables [39, 42], and was extended and combined to various successful automatic test-case generation approaches [25, 27, 36, 46]. We complement existing techniques in the following ways: Our technique works on the flexible exchange format for violation witnesses. In case such a witness constitutes only an abstract counterexample, we can use witness refinement to efficiently obtain a concrete one [9]. Such a mechanism is not available for existing test-case generation tools.

Execution. Other approaches [16, 22, 35] focus on creating tests from concrete and tool-specific counterexamples. In contrast, our approach does not require full counterexamples, but works on more flexible, possibly abstract, violation witnesses.

Debugging and Visualization. Besides executing a test, it is important to understand the cause of the error path, and there are tools and methods to debug and visualize program paths [3, 7, 28].

2 Background

A verification witness is an exchangeable object that stores valuable information about the verification process and the verification result. The key is that the format is open and exchangeable, and that many verification tools support it.

Witness Construction. It has been commonly established practice for verifiers to provide a counterexample to witness a specification violation, in particular since counterexamples were used to refine abstract models [21]. The problem was that these counterexamples were more or less ‘dumps’ of paths through the state space, sometimes not human-readable, sometimes not machine-readable. Recent efforts of the software-verification community established a common exchange format for verification results as verification witnesses [9]. In this format, a so-called violation-witness automaton (as seen in Fig. 1b) describes a state space that contains the specification violation. This state space does not necessarily have to

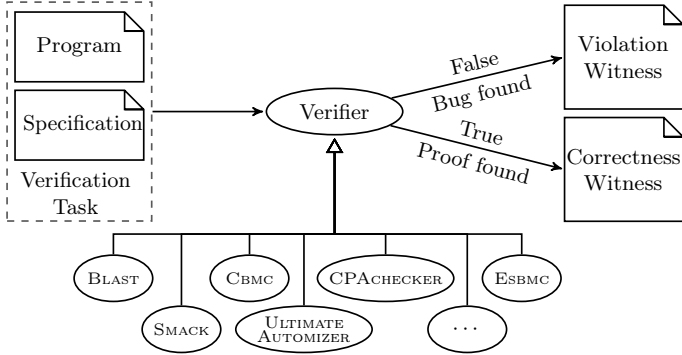


Fig. 2. Software verifiers produce witnesses

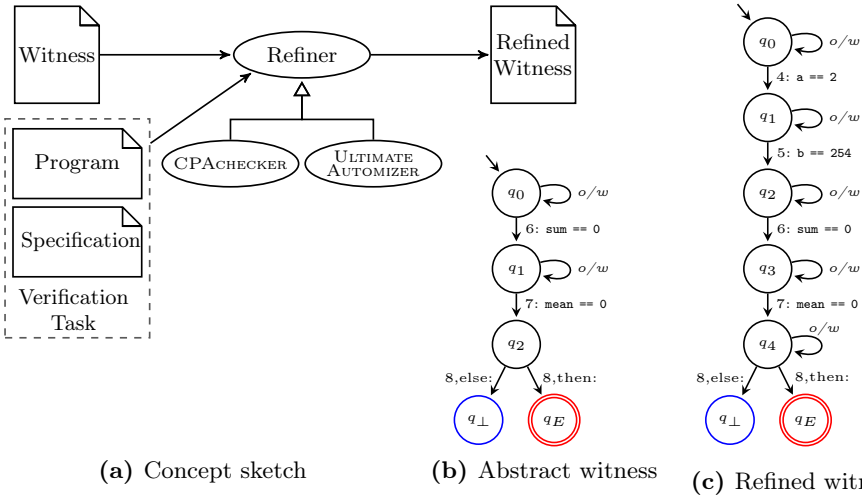


Fig. 3. Concept of witness refinement with example abstract and refined witnesses for the example program depicted in Fig. 1a from the introduction

represent just a single error path, but may contain multiple error paths and even paths without a specification violation. As an example for the use of verification witnesses, the International Competition on Software Verification (SV-COMP) applies this format and counts a report of a found bug only if a corresponding violation witness is reported and confirmed [4]. Figure 2 illustrates the process: the verifiers can be exchanged according to the needs of the user, there is no risk of technology lock-in. Figure 2 also shows that the exchange format for witnesses has recently been extended to correctness witnesses [8]. In the remainder of this paper, however, we will only consider violation witnesses.

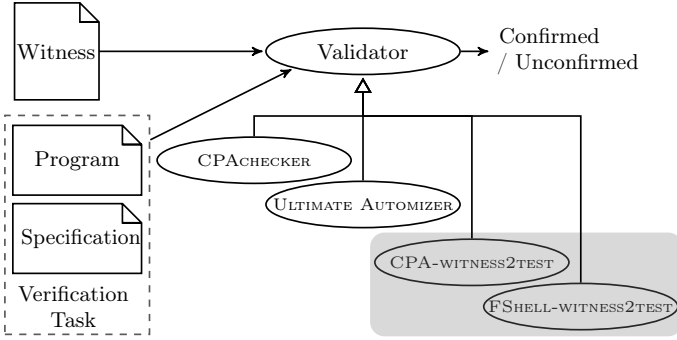


Fig. 4. Violation-witness validation

Witness Refinement. The original work on verification witnesses [9] contains the proposal to consider refinement of witnesses. The idea is to take a violation witness as input, replay it with a validating verifier, and produce a new witness that is more detailed. A more detailed violation witness is closer to a concrete program path and makes the validation process faster. We will later in this paper use an instance of a witness refiner to improve witnesses from other verification tools towards being able to successfully derive tests from witnesses. Figure 3a illustrates the optional step of using witness-refining validators to strengthen a witness. Figure 3b shows another, valid violation witness for the previously considered program from Fig. 1a. In contrast to the witness in Fig. 1b, this witness does not specify any concrete values for the two nondeterministic values of variables a and b , but specifies that a property violation occurs if the intermediate variables sum and $mean$ are both equal to 0. This witness automaton represents a set of 256 different counterexamples: every counterexample with values for a and b , so that $a + b == 0$ during execution. Figure 3c shows a violation witness that is a refinement of the more abstract witness in Fig. 3b that additionally specifies concrete values for the two variables a and b and thus restricts the search space in witness validation early on.

Witness Validation. Violation witnesses can be used to independently re-establish the verification result by using a witness-based result validator that takes the information from the witness to find a path through the state space of the program to a specification violation. Thus, a successful validation increases trust in the verification result, and developers no longer need to rely on the verifiers alone. Instead, they can focus their attention on the validated results and assign a lower priority to unconfirmed alarms. The existing witness-based result validators employ potentially-expensive model-checking techniques to replay error paths that are represented in the witness. While this is a powerful technique (it can reconstruct error paths even for abstract witnesses), the technique still has the limitations of common program-analysis and model-checking techniques, namely that the technique may over-approximate the semantics of the programming language,

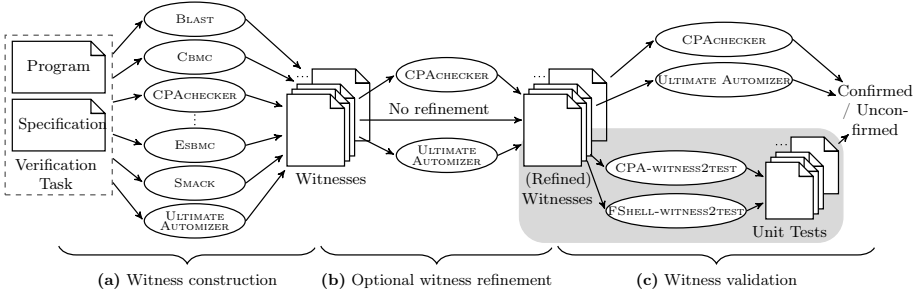


Fig. 5. Software verification with witnesses: construction, (optional) refinement, and validation work flow

thus potentially confirming false alarms or rejecting valid violation witnesses. As a solution to this, we propose an execution-based approach to witness-based result validation. Figure 4 shows the two existing validators CPAchecker and ULMATE AUTOMIZER together with the two new, execution-based validators that we introduce in this paper: CPA-WITNESS2TEST and FSHELL-WITNESS2TEST.

3 Tests from Witnesses

This section introduces a new, yet unexplored, application of witnesses that can easily be integrated into established processes for verification-result validation, as summarized by Fig. 5. The highlighted area in Fig. 5 outlines the goal: for a given violation witness, we want to construct a test that can be compiled and executed to check that the bug is realizable. In particular, driven by our desire to keep the work-flow independent from special verifiers, we want to have two independently developed implementations of such witness-to-test tools.

Our new, execution-based witness validator does not require the aid of model-checking techniques for validating verification results: we generate a test harness (test code for the program), which can be compiled and linked together with the original subject program and executed. If the execution does not trigger the described bug, the witness is deemed spurious, i.e., not realizable.

Adding this new tool to the pool of available witness-based result validators not only increases the diversity of validation techniques and its potential for establishing trust in verification results, but also adds novel features to the validation process: As a valuable by-product of a successful validation, the developers are able to obtain executable test code that is guaranteed to reproduce the bug in their system, and they can use all of the infrastructure for inspecting and debugging that they are trained and experienced in and that is already in place in their development environment. For example, a C developer might simply run GDB to step through the executable error path.

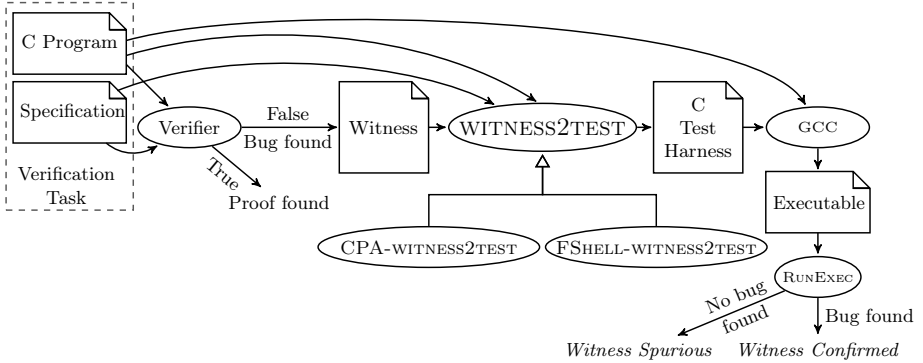


Fig. 6. Flow of execution-based result validation

Figure 6 shows the complete picture of execution-based witness validation. The verification task (a given program with a given specification) is verified by a chosen verifier. If the verifier reports a specification violation (FALSE, bug found) it also produces a violation witness. (Our work does not consider the outcome TRUE, for which the development of practical support, such as correctness witnesses [8] and compact proof witnesses [32], is also a subject of ongoing research.) The witness in GraphML format [15] is then given to WITNESS2TEST, which synthesizes a test harness that drives the program to the specification violation. In order to support our claim of independence from any particular tool implementation, we implement two completely different instances of WITNESS2TEST, namely CPA-WITNESS2TEST (based on open-source components from CPACHECKER) and FSHELL-WITNESS2TEST (based on ideas from FSHELL). The test-harness and the original (unchanged) program are then compiled and linked to obtain an executable program. The executable program is then executed in a safe execution container.⁵ If the reported specification violation is observed during this execution, the witness is confirmed. Otherwise the witness is not confirmed, most likely because the witness is not precise enough or even spurious.

3.1 CPA-WITNESS2TEST

One of our implementations for the WITNESS2TEST component of the architecture outlined in Fig. 6 is CPA-WITNESS2TEST, which is based on the CPACHECKER framework [11]. For our purpose of matching an input witness to the program source code of a verification task and generating a *test harness*, we configure CPACHECKER to use the witness automaton as a *protocol automaton* [9] to guide and restrict the state-space exploration to the program paths that the witness represents. Unlike observer automata [44], which we use to represent the specification and which can only monitor the state-space exploration of an analysis, *protocol automata* may also restrict the state-space exploration, for example to a specific program path,

⁵ We choose BENCHEXEC [13] as container solution, because it is also used by SV-COMP.

thereby guiding the analysis along that path. In our case, this path is the error path represented by the protocol automaton. We configure the analysis to only consider the (syntactical) branching information of the *protocol automaton* and to not semantically analyze the path. During this *protocol analysis*, we observe which input-value assumptions from the witness correspond to which input function or variable of the program. By collecting this information, we are able to construct a *test vector* for the program. The *test vector* maps an input value to each input variable and a list of input values to each external function. We synthesize a *test harness* from a test vector by providing initializations for input variables and definitions for external functions. An external function with a list (v_0, \dots, v_{n-1}) of $n \in \mathbb{N}$ input values is defined by using a `switch` statement with n cases over a static counter variable $0 \leq i < n$ that is initialized to 0 and incremented after each call to the function. Each case of the `switch` statement corresponds to an input value, such that `case i` selects v_i . We also inject a call to the `exit` function so that when we later execute the program, we can detect that the intended violation of the specification was triggered, i.e., the program crashed precisely due to the bug described by the witness, by checking for a specific execution return value. Figure 1c shows the `exit(107)`-call in line 2 and a definition of an input function `_VERIFIER.nondet_uchar()` in lines 3 to 12 as generated by CPA-WITNESS2TEST, where the counter variable `test_vector_index` represents i . The `switch` statement in this function definition provides sequential access to the two input values (2, 254) that CPA-WITNESS2TEST extracted from the witness of Fig. 1b for the program shown in Fig. 1a.

3.2 FSHELL-WITNESS2TEST

The key design principle of FSHELL-WITNESS2TEST is independence from existing verification infrastructure: FSHELL-WITNESS2TEST’s results shall—by design—be unbiased towards any existing software-analysis framework. While this does imply limitations on the class of witnesses that can be processed as discussed below, it does yield further advantages: FSHELL-WITNESS2TEST is easy to extend for prototyping, and does not require any background in software verification.

FSHELL-WITNESS2TEST comprises two major parts: (1) A Python-based processor of the witness and the input program, using `pycparser`⁶ to generate test vectors in a format compatible with FSHELL [31]. (2) A Perl script that translates such test vectors into a test harness.

For a given verification task and witness, FSHELL-WITNESS2TEST first parses the specification to restrict itself to reachability properties (call to error function should not be reachable). The witness and the C program are then handed to the Python-based processor. The specification defines the entry function to be used by the generated test harness.

As `pycparser` cannot handle various GCC extensions, input programs are preprocessed and sanitized by performing text replacement and removal. We then obtain the abstract syntax tree and iterate over its nodes to gather data types and

⁶ <https://github.com/eliben/pycparser>

source locations of (1) all procedure-local uninitialized variables, (2) all functions with prefix `_VERIFIER_nondet`, and (3) all uses of such functions. We refer to the locations of uninitialized variables and nondeterministic-input function uses as *watch points*.

Finally we build a linear sequence of nodes from the GraphML encoding of the witness. Traversing this sequence, any match of line numbers against the watch points triggers an attempt to extract values from assumptions in the witness. If parsing the C code that is contained in the assumption succeeds, then an input value is recorded.

The test vector is compatible with the output of FSHELL; the program of Fig. 1 yields the following test vector:

```
IN:
ENTRY main()@[file mean.c line 1]
unsigned char _VERIFIER_nondet_uchar()@[file mean.c line 4]=2
unsigned char _VERIFIER_nondet_uchar()@[file mean.c line 5]=254
```

Such a test vector is translated to a Makefile that generates an actual test harness, which consists of invocation code and the implementation of various nondeterministic-input functions that are present in the program. FSHELL-WITNESS2TEST reports FALSE (confirming the violation) if, and only if, the property violation is detected in the output of the test execution.

4 Evaluation

We perform a large experimental study to demonstrate the general applicability and the advantages of our approach.

4.1 Evaluation Goals

The goal of our experimental evaluation is to collect experience with our new kind of result validation and to support the following claims with data for a large set of witnesses:

Claim 1: Execution-based validators can confirm violation witnesses that the existing validators (which are based on model-checking technology) can not validate. Thus, execution-based validation increases the overall effectiveness.

Claim 2: Result validation based on executable tests can be faster than result validation based on model-checking technology.

Claim 3: Violation witnesses in the common exchange format for verification results (cf. Sect. 2) are a valuable source to synthesize test code for specification violations to complement existing test suites.

4.2 Experiment Setup

We used the benchmarking framework BENCHEXEC (revision `fb32a3e7`) to conduct our experiments. In order to experimentally evaluate our approach, we first

construct a large set of witnesses that is diverse in terms of (a) subject programs and (b) verification tools that create witnesses.

Subject Programs. For (a), we consider the largest available set of verification tasks⁷ from the community of automatic software verification and select all 5 692 verification tasks with a reachability property⁸.

Verifiers. For (b), we use all verification tools that participated in SV-COMP 2017 for property *ReachSafety* and whose license allows us to use it⁹. Table 1 lists all verifiers that we executed to produce violation witnesses. The table lists in the first column the verifier name with a link to the project web site for more information, and a reference to the paper describing the corresponding verifier. For the experiments, we took the archives from the competition web site.¹⁰

Collection of Witnesses. From the given verification tasks and verifiers, we started verification runs and collected the obtained violation witnesses. For this replication of the SV-COMP experiments we followed thoroughly the description on the competition web site¹⁰ and in the report [4]. In particular, we started each verifier only on those verification tasks and with those parameters that were declared by the development teams of the verifiers¹¹. The number of witnesses that we obtained with this process is reported in Table 1 (col. ‘Unref.’). Because we use all available verifiers (not only those that performed well in the competition), the set of witnesses contains also bad witnesses (e.g., that are syntactically incorrect). We did not want to exclude them for external validity.

To further increase the external validity of our evaluation, we additionally produced witnesses by applying a witness-refinement technique (cf. Sect. 2) to 13 200 witnesses above. We used the witness-refiner from the CPACHECKER framework for this step. This refinement is often able to improve imprecise witnesses by adding concrete input values, and yields another 5 766 witnesses (col. ‘Ref.’) to a total of 18 966 witnesses (col. ‘Total’) that we will run our experiments on.

In order to highlight the differences between model-checking-based validation approaches and execution-based validation approaches, we manually crafted some verification tasks and corresponding witnesses. These witnesses allow us a more detailed discussion of some effects, but were not added to our set of automatically generated witnesses.

Computing Resources. Our experiments were conducted on machines with an 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 16.04 with Linux kernel 4.4). We limited the verification runs to four processing units (i.e., two physical cores), 7 GB of memory, and 15 min of CPU time, and the

⁷ <https://github.com/sosy-lab/sv-benchmarks/tree/423cf8c>

⁸ We have to restrict the experiments to property *ReachSafety* because there were no witness validators available for the other properties.

⁹ There are also two commercial verifiers that produce witnesses, but we cannot use them due to their proprietary license.

¹⁰ <https://sv-comp.sosy-lab.org/2017/systems.php>

¹¹ <https://github.com/sosy-lab/sv-comp/tree/svcomp17/benchmark-defs>

Table 1. Violation witnesses produced by verifiers and resulting tests

Verifier	Produced witnesses			Produced tests			
	Unref.	Ref.	Total	Count	kLOC	kB	# Inputs (Avg.)
2LS [45]	992	384	1376	1208	89.9	3999	7.57
BLAST [47]	778	202	980	327	29.0	938	0.271
CBMC [34]	831	467	1298	1249	67.7	2991	6.33
CEAGLE	619	426	1045	540	92.2	262	5.39
CPA-BAM-BNB [2]	851	175	1026	158	42.9	1114	0
CPA-KIND [10]	263	193	456	656	56.2	2967	14.9
CPA-SEQ [23]	883	767	1650	838	95.5	3895	1.79
DEPTHK [43]	1159	305	1464	1302	65.4	3170	2.96
ESBMC [37]	653	148	801	478	21.0	1983	2.53
ESBMC-FALSI [37]	981	395	1376	1133	53.7	1906	1.81
ESBMC-INCR [37]	970	392	1362	1126	53.5	1896	1.82
ESBMC-KIND [24]	847	352	1199	1028	48.9	1774	1.69
FORESTER [30]	51	0	51	0	0	0	-
PREDATORHP [33]	86	61	147	80	17.2	434	0
SKINK [17]	30	25	55	44	0.290	8	0
SMACK [41]	871	632	1503	1576	128	5654	6.09
SYMBIOTIC [19]	927	411	1338	589	38.1	1375	0
SYMDIVINE [38]	247	224	471	405	13.4	580	0
UAUTOMIZER [29]	514	70	584	121	2.24	59	0
UKOJAK [40]	309	67	376	116	2.15	55	0
UTAIPAN [26]	338	70	408	121	2.23	59	0
Total	13200	5766	18966	13095	920	35119	5.60

witness-refinement and validation runs to two processing units (i.e., one physical core), 4 GB of memory, and 1.5 min of CPU time. All CPU times are reported with two significant digits. The limits are inspired by SV-COMP.

Validators. We used CPA-WITNESS2TEST in version 1.6.14-tap18 from CPA-CHECKER and FSHELL-WITNESS2TEST in revision 2a76669f from the `test-gen` branch. We used the model-checking based witness validators CPACHECKER, version 1.6.14-tap18, and ULTIMATE AUTOMIZER 0.1.8.

4.3 Availability of Data and Tools

All tools and all data obtained in our experiments are available via our supplementary web page.¹² The verification tasks are also publicly available⁷.

4.4 Results

Claim 1: Effectiveness. Table 2 reports the number of witnesses that the individual validators were able to confirm. In the columns, it shows: the results of

¹² <https://www.sosy-lab.org/research/executionbasedwitnessvalidation/>

Table 2. Confirmed witnesses and verification results

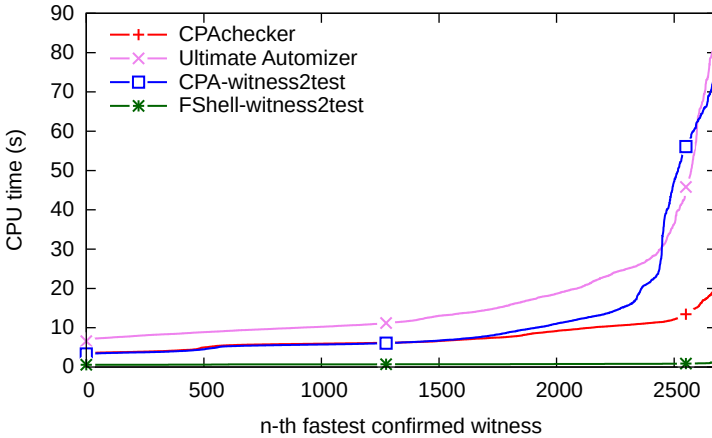
	Static validators			Dynamic validators			Union
	CPACHECKER	AUTOMIZER	Union	CPA-w2T	FSHELL-w2T	Union	
Confirmed witnesses	11 225	7 595	12 821	7 151	7 545	10 080	14 727
Unref. witnesses	5 750	3 450	7 214	3 506	3 459	5 082	9 056
Ref. witnesses	5 475	4 145	5 607	3 645	4 086	4 998	5 671
Incorrectly confirmed	18	7	25	6	0	6	31
Confirmed verif. results	5 751	5 643	7 215	5 377	5 755	7 292	9 057
Incorrectly confirmed	15	7	22	6	0	6	22

the static validators CPACHECKER and ULTIMATE AUTOMIZER, as well as the union of these two; the results of the dynamic validators CPA-w2T and FSHELL-w2T, as well as the union of these two; and the results of the union of all four validators. The union is the number of witnesses that at least one of the considered validators was able to confirm, i.e., one of CPACHECKER and ULTIMATE AUTOMIZER (col. 4), or one of CPA-w2T and FSHELL-w2T (col. 7), or any of the four (col. 8). In the rows, Table 2 is divided into confirmed witnesses (unrefined and refined witnesses, as well as incorrectly confirmed witnesses) and confirmed verification results. A witness is incorrectly confirmed if the verification result reported by a verifier is wrong and the validator reached the same, wrong conclusion using the verification-result witness that was provided by the verifier. Since for each unrefined witness from a verifier, a refined counterpart may exist, the number of confirmed witnesses is potentially double the number of verification results that were confirmed using these witnesses. Because of this, Table 2 also reports the number of confirmed verification results. We considered a verification result as confirmed if at least one of its witnesses is confirmed by the used validators. This can be the unrefined witness, or, if it exists, the refined one. The results of Table 2 show that the static validators together confirmed a total of 12 821 verification results, while the dynamic validators together confirmed a total of 10 080 results. Also, the two different validation techniques confirm different results: a union of 14 727 results were confirmed by both validation techniques together. Of the verification results that neither of the static validators was able to confirm, CPA-w2T was able to confirm 735 and FSHELL-w2T was able to confirm 1 488, meaning that the techniques complement each other well. Together, they were able to confirm 1 842 results that no static validator was able to confirm. This shows that the independently developed dynamic techniques complement each other because they are based on completely different technology. It is also interesting to consider wrong witnesses, i.e., violation witnesses that constitute false alarms. In our experiments, the verifiers produced 679 false alarms. Of these, the static approaches incorrectly confirmed 22 wrong witnesses (of different programs), while FSHELL-w2T did not wrongly confirm any false alarms. CPA-w2T confirmed 6 wrong witnesses incorrectly, all based on programs that contain floating-point arithmetic. For these, CPA-w2T has only limited support. Despite that, this highlights a high precision of our execution-based approach. In sum, using dynamic validators in addition to static validators can significantly increase the number of successfully validated verification results.

Table 3. Performance comparison for witnesses that all validators confirmed (CPU time for 2 685 witnesses)

	CPACHECKER	AUTOMIZER	CPA-w2T	FSHELL-w2T
Total time (s)	20 000	45 000	30 000	1 900
Average time (s)	7.4	17	11	0.72
Median time (s)	6.2	11	5.9	0.71

Claim 2: Efficiency. Table 3 considers only results that were confirmed by all validators, to compare the execution performance. For the dynamic validators, the reported run time contains all three steps: generating the test from the witness, compiling and linking, and executing the test. The results show that the static approaches are slow (CPACHECKER and ULTIMATE AUTOMIZER), that the approach that assembled a static analysis for test generation from CPACHECKER components is also slow (CPA-w2T), and that the light-weight implementation that is specifically tailored to generating tests from witnesses is extremely fast (FSHELL-w2T). Figure 7 displays quantile functions that show for each validator the necessary maximum CPU time (y-axis) for confirming a certain quantile of results (x-axis). We observe that FSHELL-w2T significantly outperforms all other validators.

**Fig. 7.** Quantile plot for CPU time consumed for validating witnesses accepted by all validators

Interestingly, in our validation we observed that the witnesses that require the most time to validate are witnesses that are large in size and that describe a long, detailed error path. Most of these are produced by verifiers that use bounded model checking, e.g., CBMC and CPA-KIND, or by our refinement step.

Claim 3: Test Generation. The last four columns of Table 1 relate the number of witnesses that we processed to the number of produced tests for which failing executions are realizable. With ‘produced tests’ we refer to the tests that were produced by any of the dynamic validators and for which the test execution lead to an observed specification violation. Note that because we collect tests from both dynamic validators, the numbers of produced tests exceed the number of witnesses in some rows. Since the tests are available in source code, and could be maintained and re-used by developers in practical application scenarios, we also report the size of these unit tests in lines of code, file size, and the average number of input values per generated unit test. The table shows that the number of unit tests and the accompanying size of test code that the approach can produce are significant. The results confirm that we are able to provide an interface to verification tools via witnesses and tests that avoids technology lock-in and which enables developers to explore the verification results using tools and techniques they are familiar with. The combination of software verification and execution-based result validation may also be used to automatically extend the existing test suites of a project.

4.5 Detailed Discussion of Synthetic Examples

Now we discuss a few effects in more detail on hand-crafted example witnesses. Bugs that occur after only few loop iterations are also known as *shallow* bugs, as opposed to *deep* bugs that occur after many loop iterations. One of the strengths of dynamic validation approaches is that long loops can simply be executed, while model checkers usually need to perform expensive symbolic unrolling to reveal deep bugs, which is therefore a more difficult task for them than discovering shallow bugs. Thus, we expect the set of witnesses obtained from model checkers to consist mostly of shallow bugs, while at the same time we must expect that the advantages of test-based validation become most apparent for witnesses for deeper bugs, which necessitate many unrollings. Therefore, we hand-crafted a small set of verification tasks and witnesses, including the example for computing the mean from Fig. 1a in the introduction, to exemplify the differences between the test-based approaches and those based on model checking.

Figure 8a shows an example program intended to compare the iterative sum of ascending values with the result of the Gauss sum formula, and a witness for a bug in the program. The bug is located in lines 10 to 12 and causes an error for inputs larger than or equal to 10 000. The depicted witness for this bug assigns an input value of 10 000. Figure 8b shows an example program that increments two variables x and y 1 000 000 times and then asserts their equality in line 12, and a witness for a violation of this assertion. Since y is initialized to $x + 1$ in line 5, the assertion will fail for any value of x . The depicted witness for this bug assigns an input value of 0. Figure 8c shows an example program with a variable n initialized with an input function in line 4 and copies its value to a variable x in line 5. In the same line, a variable y is initialized to 0. Then, in lines 6 to 9, x is decremented and simultaneously y is incremented, until x is 0, so essentially, y counts the loop iterations, and $n - x = y$ is a loop invariant. Consequently, y must be equal to n at the end of the loop, and therefore the call to

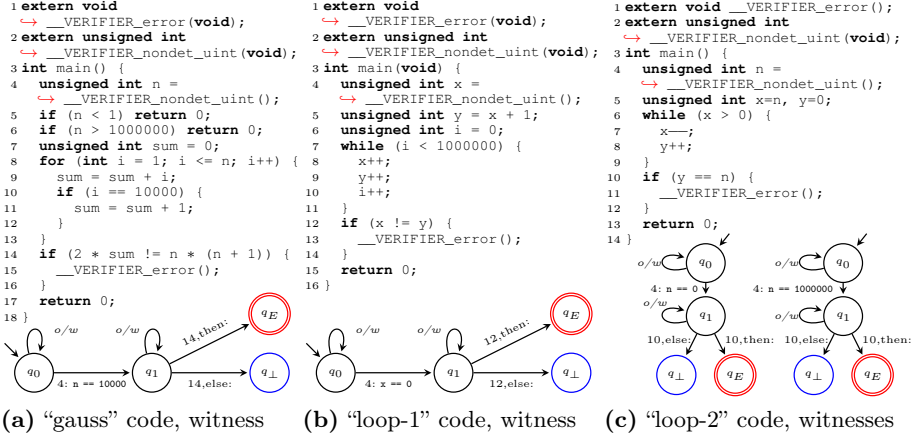


Fig. 8. Hand-crafted tasks and witnesses

the error function in line 11 is called for any input value, so that both witnesses in Fig. 8c are valid counterexamples. The first of these witnesses, however, describes a violation that skips the loop entirely with an input value of 0, while the second one, due to assigning an input value of 1 000 000, reaches the violation in line 11 only after 1 000 000 loop iterations. We expect all validators to quickly validate the witnesses for shallow bugs, i.e., the one depicted in Fig. 1a and the first witness in Fig. 8c, but we expect test-based validators to perform significantly better on the witnesses for deep bugs, i.e., those depicted in Fig. 8a and 8b, and the second witness in Fig. 8c. Table 4 reports the results for validating these tasks and largely confirms our expectations. While CPACHECKER exceeds its resource limitations (“M” for exceeding the memory limit, “T” for exceeding the CPU time limit) for all witnesses except for the two that represent shallow bugs, CPA-w2T and FSHELL-w2T quickly confirm all witnesses (✓). It is somewhat surprising to see that ULTIMATE AUTOMIZER is able to confirm the `loop-2/wit-2` of Fig. 8c. Checking the tool output, however, reveals that ULTIMATE AUTOMIZER ignored the input value of `n` specified by the witness and used 0 instead of 1 000 000. We were also surprised that the witnesses in the first two rows were rejected by ULTIMATE AUTOMIZER (✗), but since the confirmations of the execution-based validators along with their trustworthy executable tests give us confidence that the witnesses are correct, we assume that the rejections are either caused by the complexity of validating the witnesses or by an approximating behavior of ULTIMATE AUTOMIZER similar to the one leading to the rejection of `loop-2/wit-2`. Overall, we confirm that for this class of witnesses, dynamic approaches are more efficient and more effective than static approaches.

Table 4. Validation of hand-crafted witnesses

Witness	CPACHECKER		AUTOMIZER		CPA-w2T		FSHELL-w2T	
	Result	Time (s)	Result	Time (s)	Result	Time (s)	Result	Time (s)
gauss	M	-	✗	11	✓	3.4	✓	0.60
loop-1	T	-	✗	9.6	✓	3.4	✓	0.60
loop-2/wit-1	✓	3.8	✓	8.0	✓	3.4	✓	0.58
loop-2/wit-2	T	-	✓	7.5	✓	3.2	✓	0.58
mean	✓	3.5	✓	7.1	✓	3.6	✓	0.58

5 Conclusion

Developers are familiar with testing, and there are many tools available for bug analysis that are based on execution, such as debuggers. We try to close the gap between available verification tools and the desire for more precise bug finding by leveraging verification witnesses in an exchangeable standard format. We synthesize tests (test code) from verification results (witnesses) and check the tests for realizability by compiling them, linking them together with the original program, and executing the result in an isolating container. Prior to our work, developers would execute a verification tool and obtain the verification results, which include a violation witness in case a bug is found. Now, we can use the violation witness to obtain a test that drives the program to the specification violation (i.e., into the crash that the developer wants to investigate), while at the same time, we avoid verification-tool lock-in due to the exchangeable standard format. The approach reports only those tests to the developer that really expose the bug; any false alarms are suppressed. The results of our thorough experimental study are encouraging: We verified thousands of programs from the largest publicly-available collection of C verification tasks, consisting of 73 million lines of source code (2.3 GB), and synthesized tests that confirmed 7 286 verification results exposing known bugs in 974 different verification tasks.

References

1. Alglave, J., Donaldson, A.F., Kroening, D., Tautschnig, M.: Making software verification tools really work. In: Bultan, T., Hsiung, P.-A. (eds.) Proceedings of ATVA 2011. LNCS, vol. 6996, pp. 28–42. Springer, Heidelberg (2011)
2. Andrianov, P., Friedberger, K., Mandrykin, M., Mutilin, V., Volkov, A.: CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 355–359. Springer, Heidelberg (2017)
3. Artho, C., Havelund, K., Honiden, S.: Visualization of concurrent program executions. In: Belli, F., Chen, A., Lin, H., McMillin, B., Mei, H. (eds.) Proceedings of COMPSAC 2007, pp. 541–546. IEEE (2007)

4. Beyer, D.: Reliable and reproducible competition results with `BENCHEXEC` and witnesses (report on `SV-COMP 2016`). In: Chechik, M., Raskin, J.-F. (eds.) *Proceedings of TACAS 2016*. LNCS, vol. 9636, pp. 887–904. Springer, Heidelberg (2016)
5. Beyer, D.: Software verification with validation of results. In: Legay, A., Margaria, T. (eds.) *Proceedings of TACAS 2017*. LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017)
6. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) *Proceedings of ICSE 2004*, pp. 326–335. IEEE (2004)
7. Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Chaudhuri, S., Farzan, A. (eds.) *Proceedings of CAV 2016*. LNCS, vol. 9780, pp. 502–509. Springer, Cham (2016)
8. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Zimmermann, T., Cleland-Huang, J., Su, Z., (eds.) *Proceedings of FSE 2016*, pp. 326–337. ACM (2016)
9. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Di Nitto, E., Harman, M., Heymans, P. (eds.) *Proceedings of FSE 2015*, pp. 721–733. ACM (2015)
10. Beyer, D., Dangl, M., Wendler, P.: Boosting k -induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) *Proceedings of CAV 2015*. LNCS, vol. 9206, pp. 622–640. Springer, Cham (2015)
11. Beyer, D., Keremoglu, M.E.: `CPACHECKER`: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) *Proceedings of CAV 2011*. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
12. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. *Proceedings of HVC 2017*. LNCS, vol. 10629, pp. 99–114. Springer, Cham (2017)
13. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Technol. Transf.* (2017)
14. Beyer, D., Wendler, P.: Reuse of verification results. In: Bartocci, E., Ramakrishnan, C.R. (eds.) *Proceedings of SPIN 2013*. LNCS, vol. 7976, pp. 1–17. Springer, Heidelberg (2013)
15. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report structural layer proposal. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) *Proceedings of GD 2001*. LNCS, vol. 2265, pp. 501–512. Springer, Heidelberg (2002)
16. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: `EXE`: Automatically generating inputs of death. In: Juels, A., Wright, R.N., De Capitani di Vimercati, S. (eds.) *Proceedings of CCS 2006*, pp. 322–335. ACM (2006)
17. Cassez, F., Sloane, A.M., Roberts, M., Pigram, M., Suvanpong, P., de Aledo, P.G.: Skink: Static analysis of programs in LLVM intermediate representation. In: Legay, A., Margaria, T. (eds.) *Proceedings of TACAS 2017*. LNCS, vol. 10206, pp. 380–384. Springer, Heidelberg (2017)
18. Castaño, R., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Model checker execution reports. In: Rosu, G., Di Penta, M., Nguyen, T.N. (eds.) *Proceedings of ASE 2017*, pp. 200–205. IEEE (2017)
19. Chalupa, M., Vitovská, M., Jonáš, M., Slaby, J., Strejček, J.: Symbiotic 4: Beyond reachability. In: Legay, A., Margaria, T. (eds.) *Proceedings of TACAS 2017*. LNCS, vol. 10206, pp. 385–389. Springer, Heidelberg (2017)
20. Christakis, M., Bird, C.: What developers want and need from program analysis: An empirical study. In: Lo, D., Apel, S., Khurshid, S. (eds.) *Proceedings of ASE 2016*, pp. 332–343. ACM (2016)

21. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* **50**(5), 752–794 (2003)
22. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Roman, G.-C., Griswold, W.G., Nuseibeh, B. (eds.) *Proceedings of ICSE 2005*, pp. 422–431. ACM (2005)
23. Dangl, M., Löwe, S., Wendler, P.: CPACHECKER with support for recursive programs and floating-point arithmetic. In: Baier, C., Tinelli, C. (eds.) *Proceedings of TACAS 2015*. LNCS, vol. 9035, pp. 423–425. Springer, Heidelberg (2015)
24. Gadelha, M.Y.R., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. *STTT* **19**(1), 97–114 (2017)
25. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) *Proceedings of PLDI 2005*, pp. 213–223. ACM (2005)
26. Greitschus, M., Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Taipan: Trace abstraction and abstract interpretation. In: Legay, A., Margaria, T. (eds.) *Proceedings of TACAS 2017*. LNCS, vol. 10206, pp. 399–403. Springer, Heidelberg (2017)
27. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: A new algorithm for property checking. In: Young, M., Devanbu, P.T., (eds.) *Proceedings of FSE 2006*, pp. 117–127. ACM (2006)
28. Gunter, E.L., Peled, D.: Path exploration tool. In: Cleaveland, W.R. (ed.) *Proceedings of TACAS 1999*. LNCS, vol. 1579, pp. 405–419. Springer, Heidelberg (1999)
29. Heizmann, M., Chen, Y.-W., Dietsch, D., Greitschus, M., Nutz, A., Musa, B., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate automizer with an on-demand construction of Floyd-Hoare automata. In: Legay, A., Margaria, T. (eds.) *Proceedings of TACAS 2017*. LNCS, vol. 10206, pp. 394–398. Springer, Heidelberg (2017)
30. Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: FORESTER: From heap shapes to automata predicates. In: Legay, A., Margaria, T. (eds.) *Proceedings of TACAS 2017*. LNCS, vol. 10206, pp. 365–369. Springer, Heidelberg (2017)
31. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Pecheur, C., Andrews, J., Di Nitto, E. (eds.) *Proceedings of ASE 2010*, pp. 407–416. ACM (2010)
32. Jakobs, M.-C., Wehrheim, H.: Compact proof witnesses. In: Barrett, C., Davies, M., Kahsai, T. (eds.) *Proceedings of NFM 2017*. LNCS, vol. 10227, pp. 389–403. Springer, Cham (2017)
33. Kotoun, M., Peringer, P., Šoková, V., Vojnar, T.: Optimized PredatorHP and the SV-COMP heap and memory safety benchmark. In: Chechik, M., Raskin, J.-F. (eds.) *Proceedings of TACAS 2016*. LNCS, vol. 9636, pp. 942–945. Springer, Heidelberg (2016)
34. Kroening, D., Tautschnig, M.: CBMC: C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) *Proceedings of TACAS 2014*. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)
35. Li, K., Reichenbach, C., Csallner, C., Smaragdakis, Y.: Residual investigation: Predictive and precise bug detection. In: Heimdahl, M.P.E., Su, Z., (eds.) *Proceedings of ISSTA 2012*, pp. 298–308. ACM (2012)
36. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Emmerich, W., Knight, J., Rothermel, G. (eds.) *Proceedings of ICSE 2007*, pp. 416–426. IEEE (2007)

37. Morse, J., Ramalho, M., Cordeiro, L., Nicole, D., Fischer, B.: ESBMC 1.22. In: Ábrahám, E., Havelund, K. (eds.) Proceedings of TACAS 2014. LNCS, vol. 8413, pp. 405–407. Springer, Heidelberg (2014)
38. Mrázek, J., Jonáš, M., Štill, V., Lauko, H., Barnat, J.: Optimizing and caching SMT queries in SymDIVINE. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 390–393. Springer, Heidelberg (2017)
39. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: Butler, M., Schulte, W. (eds.) Proceedings of FM 2011. LNCS, vol. 6664, pp. 73–87. Springer, Heidelberg (2011)
40. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks. In: Baier, C., Tinelli, C. (eds.) Proceedings of TACAS 2015. LNCS, vol. 9035, pp. 458–460. Springer, Heidelberg (2015)
41. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) Proceedings of CAV 2014. LNCS, vol. 8559, pp. 106–113. Springer, Cham (2014)
42. Rocha, H., Barreto, R., Cordeiro, L., Neto, A.D.: Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) Proceedings of IFM 2012. LNCS, vol. 7321, pp. 128–142. Springer, Heidelberg (2012)
43. Rocha, W., Rocha, H., Ismail, H., Cordeiro, L., Fischer, B.: DepthK: A k -induction verifier based on invariant inference for C programs. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 360–364. Springer, Heidelberg (2017)
44. Schneider, F.B.: Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* **3**(1), 30–50 (2000)
45. Schrammel, P., Kroening, D.: 2LS for program analysis. In: Chechik, M., Raskin, J.-F. (eds.) Proceedings of TACAS 2016. LNCS, vol. 9636, pp. 905–907. Springer, Heidelberg (2016)
46. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Wermelinger, M., Gall, H.C. (eds.) Proceedings of FSE 2005, pp. 263–272. ACM (2005)
47. Shved, P., Mandrykin, M., Mutilin, V.: Predicate analysis with BLAST 2.7. In: Flanagan, C., König, B. (eds.) Proceedings of TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012)
48. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Avruin, G.S., Rothermel, G. (eds.) Proceedings of ISSTA 2004, pp. 97–107. ACM (2004)

Regular Contributions



An Approximation-Based Approach for the Random Exploration of Large Models

Julien Bernard, Pierre-Cyrille Héam^(✉), and Olga Kouchnarenko

FEMTO-ST Institute, CNRS, Univ. Bourgogne Franche-Comté,
Besançon, France

{jbernard,pheam,okouchna}@femto-st.fr

Abstract. System modeling is a classical approach to ensure their reliability since it is suitable both for a formal verification and for software testing techniques. In the context of model-based testing an approach combining random testing and coverage based testing has been recently introduced [9]. However, this approach is not tractable on quite large models. In this paper we show how to use statistical approximations to make the approach work on larger models. Experimental results, on models of communicating protocols, are provided; they are very promising, both for the computation time and for the quality of the generated test suites.

1 Introduction

Many critical tasks are now assigned to automatic systems. In this context, producing trusted software is a challenging problem and a central issue in software engineering. Recent decades have witnessed the strengthening of many formal approaches to ensure software reliability, from verification (model-checking, automatic theorem proving, static analysis) to testing, which remains an inescapable step to ensure software quality. A great effort has been made by the scientific community in order to upgrade hand-made testing techniques to scalable and proven framework.

Experience shows that random testing is a very efficient technique for detecting bugs, especially at the first stages of testing activities. The strength of random testing consists of its independence on tester's priority and choices. However, the nature of random testing is *to draw randomly a test rather than choosing it*, and it is therefore inefficient to detect behaviour of a program occurring with a very low probability. In [9], a random testing approach consisting of the exploration of large graph based models has been proposed. In order to tackle the problem of low probabilistic behaviour, the authors have also suggested to bias the random generation, by combining it with a coverage criterion, in order to optimize the probability to meet system' features described by this criterion. It however requires the computation of large linear systems, which becomes rapidly intractable in practice for large graphs.

In this paper we propose a sampling-based approach in order to compute approximated values of the system' solutions, deeply improving the efficiency of

the computation. Experimental results on various graphs provided in the paper show a very significant time computation improvement while keeping similar covering statistical properties.

1.1 Related Work

A prevailing methods in model-based testing consists in designing the system under test by a graph-based formal model [18,26] on which different algorithms may be used to generate the test suites. This approach has been used for a large class of applications from security of Android systems [23] to digital ecosystems [19]. A large variety of models can be used for model-based testing such as Petri nets [24], timed automata [27], pushdown automata [11], process algebra [2], etc. Moreover, a strength of model-based testing is that it can be combined with several verification approaches, such as model-checking [8] or those using SMT-solvers [1]. A general taxonomy with many references on model-based testing approach can be found in [25].

Random testing approaches have been introduced in [12] and are widely used in the literature, either for generating data [13,16] or for generating test suites [21]. As far as we know, the first work combining random testing and model-based testing has been proposed in [14] as a combination of model-checking and testing. In [9] the authors have proposed an improved approach to explore the models at random. This technique has been extended to pushdown models [11,15] and to grammar-based systems [10].

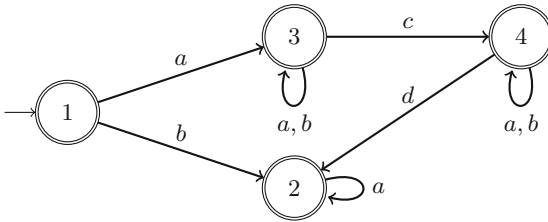


Fig. 1. Illustrating example.

1.2 Formal Background

For a general reference on probability theory, see [20].

Finite Automata. Models considered in this article are finite automata, that are labelled graphs. More precisely, a finite automaton \mathcal{A} is a tuple (Q, Σ, E, I, F) , where Q is a finite set of states, Σ is a finite alphabet, $E \subseteq Q \times \Sigma \times Q$ is the set of transitions, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states. A path σ in a finite automaton is a sequence $(p_0, a_0, p_1) \dots (p_{N-1}, a_{N-1}, p_N)$ of transitions. The integer N is the length of the path. If $p_0 \in I$ and $p_N \in$

F , σ is said successful. The path σ visits a state q if there exists i such that $p_i = q$. An automaton is trim if every state is visited by at least one successful path. All automata considered throughout this paper are trim. An example of an automaton is depicted in Fig. 1: its set of states is $\{1, 2, 3, 4\}$, the alphabet is $\{a, b, c, d\}$, its set of transitions is

$$\{(1, a, 3), (3, a, 3), (3, b, 3), (3, c, 4), (4, a, 4), (4, b, 4), (1, b, 2), (2, a, 2), (4, d, 2)\},$$

its set of initial states is reduced to $\{1\}$ and all its states are final.

Let $\mathcal{A} = (Q, \Sigma, E, I, F)$ be a n -state automaton and $q \in Q$. We denote by \mathcal{A}_q the automaton on the alphabet Σ whose set of states is $Q \times \{0, 1\}$ (two copies of Q) and:

- Its set of initial states is $I \times \{0\}$,
- Its set of final states is $F \times \{1\} \cup (F \cap \{q\}) \times \{0\}$,
- Its set of transitions is $E' = \{((p, 0), a, (p', 0)) \mid (p, a, p') \in E \text{ and } p \neq q\} \cup \{((p, 1), a, (p', 1)) \mid (p, a, p') \in E\} \cup \{((q, 0), a, (p', 1)) \mid (q, a, p') \in E\}$.

Intuitively, a successful path in \mathcal{A}_q starts with an initial state of the form $(q_0, 0)$ and remains in a state of the form $(p, 0)$ until it visits q . Then, if q is final in \mathcal{A} it may ends or continue with states of the form $(p, 1)$. One can easily show that there is a bijection between the set of successful paths of \mathcal{A}_q of length N and the set of successful paths of \mathcal{A} of length N visiting q . denoted Let us consider for instance the automaton depicted in Fig. 1 and state 3. The corresponding automaton is depicted in Fig. 2 (states $(4,0)$, $(1,1)$ and $(2,0)$ have to be removed to make the resulting automaton trim).

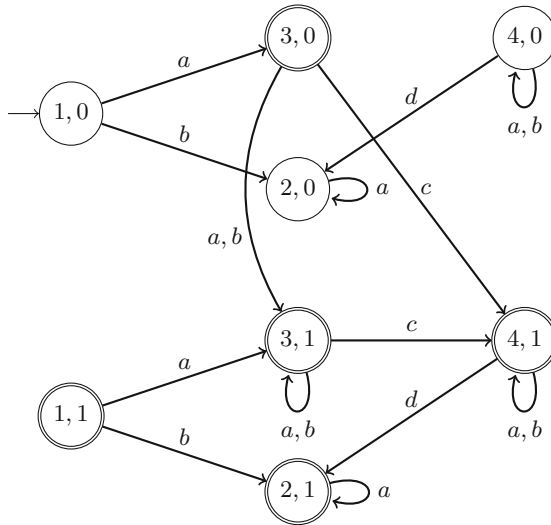


Fig. 2. Illustrating example for constrained paths.

Automata used in many testing applications have a bounded outgoing degree. Throughout this paper, we consider that $|E| = O(|Q|)$. Note that it is not a theoretical requirement: it is only used for the complexity issues. Indeed, all proposed algorithms work for any automata. Under this hypothesis, computing \mathcal{A}_q can be done in time $O(n^2)$ and the resulting automaton has at most twice the number of states (regardless of the fact that $|E| = O(|Q|)$).

Counting Paths. We call NUMPATHS an algorithm that, given a finite automaton \mathcal{A} and a positive integer N , computes the number of successful paths of length N in \mathcal{A} . We call RANDOMPATH an algorithm that, given a finite automaton \mathcal{A} and positive integers N, k , randomly, uniformly and independently generates k successful paths of length N in \mathcal{A} . Several algorithms have been developed for processing NUMPATHS and RANDOMPATH [22], whose complexities depend on several parameters. Let us observe, without going into details, using floating point arithmetics, that NUMPATHS can be performed in $O(nN \log N)$, where n is the number of states of \mathcal{A} . And RANDOMPATH can be performed in time $O(knN \log^2 N)$. Note that the different approaches may have different meanings of time/space complexities, both for the preprocessing step and the generation step. The reader can see [22, Table 4] and [4, Table 1] for more details.

Random Biased Exploration of Finite Automata. The objective is here to biased the random generation of paths (i.e. not use a uniform random generation) in order to improve the state coverage of the automata. It is necessary to provide a quite detailed description of the algorithms in [22]. The first approach, denoted later **Uniform**, consists in uniformly picking up a given number of paths from the set of successful paths of a given length. The approach can be applied to very large graphs with hundreds of nodes (see [9, Sect. 6]). However, rare events can be missed up, and in order to optimize¹ the coverage criterion (let us present it here for nodes coverage²) of the graph, the following approach, denoted later **Exact**, is proposed to produce k successful paths of an automaton \mathcal{A} whose set of states is $\{1, \dots, n\}$:

1. Choose a set S of successful paths (for instance those of length less than or equal to a constant N),
2. For each pair of nodes, compute the probability $\alpha_{i,j}$ that a path of S visiting j also visits i ,
3. Solve the linear programming system whose variables are $p_{\min}, \pi_1, \dots, \pi_n$:

$$\begin{aligned} & \text{maximize } p_{\min}, \text{ under the constraints} \\ & \left\{ \begin{array}{l} \text{for all } j, p_{\min} \leq \sum_{i=1}^n \alpha_{i,j} \pi_i \\ 1 = \sum_{i=1}^n \pi_i \end{array} \right. \end{aligned} \quad (1)$$

Solution is a distribution $\pi = (\pi_1, \dots, \pi_n)$ of probabilities over the states of the automaton,

¹ Computing test suites of a reduced size is a major issue in the testing process, since executing test on the system is frequently a complex issue (not addressed in this paper).

² The approach can easily be adapted for transitions coverage.

4. Repeat k times: pick a node i up at random according to the distribution π . Pick up at random (uniformly) a path visiting i .

The goal of the linear programming system is to optimize the minimal probability p_{\min} of a state to be visited by a random path.

Let us illustrate this approach on the example depicted in Fig. 1. Note that if the goal is to cover a given proportion of the set of states (for instance) Step 4. can be replaced by: generate paths as soon as the wanted proportion of states are visited by these paths. There are 16 successful paths of strictly positive length less than or equal to 3 reported in Table 1. Since the automaton is deterministic, one can identify successful paths with their labels. Let S_{exa} be this set of paths.

Table 1. Successful paths of length less than or equal to 3 for Example 1.

Length	Paths	Number of paths
1	a, b	2
2	aa, ab, ac, ba	4
3	$aaa, aab, aba, abb, aac, abc, aca, acb, acd, baa$	10

There are 4 out of 16 paths of S_{exa} visiting state 2. Therefore, the probability of visiting state 2 by uniformly generated paths of S_{exa} is $\frac{1}{4}$. In order to generate a path visiting 2, one has to generate averagely 4 tests. Moreover, for this example $\alpha_{i,j}$'s matrix is

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0.25 & 1 & \frac{1}{13} & \frac{1}{6} \\ 0.825 & 0.25 & 1 & 1 \\ 0.375 & 0.25 & \frac{6}{13} & 1 \end{pmatrix}.$$

For instance, $\alpha_{1,i} = 1$ for every i since all paths visit 1. Similarly, $\alpha_{3,4} = 1$ since all paths visiting 4 also visits 3. There are four paths (b, ba, baa and acd) of needed length visiting 2 and, among these paths, only acd visits 4. Therefore $\alpha_{4,2} = \frac{1}{4}$. The resolution³ of linear programming systems (1) provides in this context the solution: $\pi_1 = 0$, $\pi_2 = 0.526315$, $\pi_3 = 0$, and $\pi_4 = 0.473685$. In this context, the biased approach covers all states averagely with less than 3 generated paths.

The bottleneck of this approach is Step 2. since computing the $\alpha_{i,j}$ s requires many manipulations on the graphs (it requires to compute the $(\mathcal{A}_i)_j$): for each $i \neq j$, Algorithm NUMPATHS has to be applied to graphs 4 times larger than the initial ones. The complexity is in $O(n^3N \log N)$ with quite large involved constants, making the approach intractable for big n 's.

³ Resolutions have been performed using the `lp_solve` solver.

1.3 Contributions

In this paper we propose to not exactly compute the $\alpha_{i,j}$ s but instead to approximate them by using statistical sampling, as described in Sect. 2. Experimental results on several examples of communication protocols models are provided in Sect. 3. The paper reports on very promising experimental results: the computation time is significantly better for a similar quality of the large graphs coverage.

2 Approximating the Linear Programming Systems

In this section, we propose to approximate the coefficients $\alpha_{i,j}$ by $\alpha_{i,j}^{\text{approx}}$ by using classical sampling techniques. Using m times Algorithm `RandomPath`, one can count as m_i the number of paths visiting i , and $m_{i,j}$ the number of paths visiting both i and j . If $m_i \neq 0$ then $\alpha_{i,j}^{\text{approx}} = \frac{m_{i,j}}{m_j}$.

2.1 Approximation Algorithm

More precisely, let there be a trim finite automaton $\mathcal{A} = (Q, A, E, I, F)$, a strictly positive integer m , a strictly positive integer N and a strictly positive integer r (the parameter r is used to provide some bounds on the precision of the approximation: each evaluation of a parameter is estimated using a sample of size at least r).

(Step 1): Generate m successful paths in \mathcal{A} of length less than or equal to N uniformly. For each $i \in Q$, let m_i^{approx} be the number of these paths visiting i , and $m_{i,j}^{\text{approx}}$ be the number of these paths visiting both i and j .

(Step 2): For each $i, j \in Q$, $i \neq j$

(a) If $r = 0$ and $m_j^{\text{approx}} = 0$, then let $\alpha_{i,j}^{\text{approx}} = 0$,

(b) If $m_j^{\text{approx}} > r$, let $\alpha_{i,j}^{\text{approx}} = m_{i,j}^{\text{approx}} / m_j^{\text{approx}}$,

(c) If $m_j^{\text{approx}} \leq r$, generate r paths visiting i and set $\alpha_{i,j}^{\text{approx}}$ as the proportion of these paths visiting j .

(Step 2): For each $i \in Q$, $\alpha_{i,i}^{\text{approx}} = 1$.

Let us illustrate the approach on the example depicted in Fig. 1, with $N = 4$ and $r = 0$. Rather than compute exactly the $\alpha_{i,j}$'s, we randomly and uniformly generate 1000 paths of length less than or equal to 3. We obtain the following matrix for the $\alpha_{i,j}^{\text{approx}}$:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0.243 & 1 & 0.0835 & 0.1778 \\ 0.826 & 0.284 & 1 & 1 \\ 0.288 & 0.284 & 0.4697 & 1 \end{pmatrix}.$$

The resolution of systems provides the solution $\pi_1 = 0$, $\pi_2 = 0.538019$, $\pi_3 = 0$ and $\pi_4 = 0.461981$.

In this example, there are 243 paths visiting 2, 826 paths visiting 3 and 288 paths visiting 4. Therefore, running the algorithm with $r = 250$ will change the second column of the matrix since $m_2^{\text{approx}} < 250$. In this case, the automaton for the paths visiting 2 is computed. Generating 250 paths visiting state 2 provides the following matrix:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0.243 & 1 & 0.0835 & 0.1778 \\ 0.826 & 0.256 & 1 & 1 \\ 0.288 & 0.256 & 0.4697 & 1 \end{pmatrix}.$$

The resolution of systems provides the solution $\pi_1 = 0$, $\pi_2 = 0.524965$, $\pi_3 = 0$ and $\pi_4 = 0.475035$.

Section 3 describes more experiments and provides details, both on the quality of the results and on the time to compute the $\alpha_{i,j}$'s.

Notice too that, as mentioned in [9], the optimal solution leads to a loss of randomness: many π_i 's are null. It is proposed in [9] to fix minimal probability to the π_i 's. It can be directly adapted in our approach by adding, in the programming linear system, some inequations of the form $\pi_i \geq \varepsilon$. This situation would not be considered in the experiments developed in this paper.

2.2 Complexity

We investigate in this section the worst case complexity of the proposed algorithm. Step (1) can be performed in time $O(mnN \log^2 N + mn^2)$: first the m paths are generated in time $O(mnN \log^2 N)$. These paths are not stored but a table t of size $m \times n$ is filled in the following way: $t[i][j] = 1$ if the i -th path visits state j , and $t[i][j] = 0$ otherwise. It is done on the fly and in time $O(nm)$. The m_i^{approx} are calculated by computing columns sums in time $O(nm)$ too. Similarly, each $m_{i,j}^{\text{approx}}$ can be computed in time $O(m)$. Therefore, computing all of them is performed in time $O(mn^2)$.

Step 2-(a) is performed in time $O(1)$ as well as Step 2-(b). Step 2-(c) is performed in time $O(rnN \log^2 N)$: computing the specific automaton is done in time $O(n)$ (under the hypothesis that the number of transitions is in $O(n)$).

Step 3 is performed in time $O(n)$.

In conclusion, if we denote by s the number of calls to Step 2-(c), the complexity is: $O(((sr + m)nN \log^2 N + mn^2))$.

A small r (for instance $r = 0$) will provide a small s ($s = 0$), but a coarser approximation, as exposed in the next section.

2.3 Precision of the $\alpha_{i,j}^{\text{approx},s}$

Each $\alpha_{i,j}$ is the parameter of Bernoulli's Law (see [20, Sect. 2.2]). The precision of the estimation can classically be obtained using either Bienaymé-Chebyshev's Inequality [5, 7] or Hoeffding's Inequality [17].

First, assuming that $m_j^{\text{approx}} > r$, then Bienaymé-Chebyshev's Inequality provides for any $\varepsilon > 0$:

$$\mathbb{P}(|\alpha_{i,j}^{\text{approx}} - \alpha_{i,j}| \geq \varepsilon) \leq \frac{\alpha_{i,j}(1 - \alpha_{i,j})}{\varepsilon^2 m_j^{\text{approx}}} \leq \frac{1}{4\varepsilon^2 r},$$

and if $m_j^{\text{approx}} \leq r$, it provides:

$$\mathbb{P}(|\alpha_{i,j}^{\text{approx}} - \alpha_{i,j}| \geq \varepsilon) \leq \frac{\alpha_{i,j}(1 - \alpha_{i,j})}{\varepsilon^2 r^2} \leq \frac{1}{4\varepsilon^2 r}.$$

In order to have an $\varepsilon = 0.1$ precision with a 0.95 confidence level, r has to be fixed to 500 (this is an upper bound).

Secondly, one can have another evaluation using Hoeffding's Inequality (better in most of cases): for any $0 < \rho < 1$,

$$\mathbb{P}(|\alpha_{p,q}^{\text{approx}} - \alpha_{p,q}| \geq \varepsilon) \leq 2e^{-2r\varepsilon^2}.$$

In order to have an $\varepsilon = 0.1$ precision with a 0.95 confidence level, r has to be fixed to 185 (this is also an upper bound).

Let us note that the two above inequalities provide upper bounds that are not very tight: for states j frequently visited by random paths, m_j will be significantly greater than r , and the estimation of the algorithm will be very precise. As it is shown in the next section, running the algorithm with $r = 0$ frequently provides very acceptable solutions and very good solutions with $r = 10$. For $r = 10$ the two bounds above do not ensure precise estimations: Hoeffding's Inequality states that with a 0.8 confidence level we have an estimation of $\alpha_{i,j}$ with $\varepsilon = 0.34$. An hypothesis explaining why $r = 10$ works is that it is important to detect whether while visiting j the probability to also visit i is significant. But it's not critical to know how significant it is, for instance if $\alpha_{i,j} = 0.1$ or 0.4 ; it is important to know that generating a path visiting j will quite frequently provide a path visiting i .

Finally, other statistical tools can be used to obtain bounds on r , for instance the well-known central limit theorem.

3 Experiments

This section is dedicated to an experimental evaluation of the proposed approximation-based approach. In Sect. 3.1 the set of used automata is described. Section 3.2 explains the experimental protocol. Finally, the obtained experimental results are provided in Sect. 3.3, both for the quality of the approach and for computation time.

3.1 Benchmark

Experiments have been done on several automata modeling communication protocols designed for the FAST tool [3] available⁴ online as a library of parametric

⁴ <http://www.lsv.fr/Software/fast/examples/examples.tgz>.

r=0	90%	95%	99%	100%
RW	4.38 2-10	4.38 2-10	4.38 2-10	4.38 2-10
Uniform	4.22 2-11	4.22 2-11	4.22 2-11	4.22 2-11
Approx 10	4.65 2-13	4.65 2-13	4.65 2-13	4.65 2-13
Approx 1000	4.18 2-12	4.18 2-12	4.18 2-12	4.18 2-12
Exact	4.42 2-12	4.42 2-12	4.42 2-12	4.42 2-12

Barber1
15 states
18 transitions

r=0	90%	95%	99%	100%
RW	428 18-1724	790 159-2062	2451.4 599-7727	2451.4 599-727
Uniform	22.58 9-45	35.45 14-79	92.36 32-216	92.36 32-216
Approx 10	11.7 7-21	17.2 10-52	39.3 16-181	39.35 16-181
Approx 1000	10.46 7-20	14.9 8-31	30.9 11-73	30.9 11-73
Exact	11.23 7-24	15.5 8-30	29.8 13-71	29.8 13-71

Dekker1, 86 states, 178 transitions

r=0	90%	95%	99%	100%
RW	15 9-27	21.5 13-37	37.8 18-70	50.5 19-130
Uniform	31.2 16-51	47.7 22-88	87.6 40-166	115.5 45-278
Approx 10	19.5 13-34	28.1 17-51	48.6 27-146	67.9 30-146
Approx 1000	18 10-29	24.5 13-38	40.3 20-74	50.0 26-86
Exact	18.4 12-30	24.8 18-38	39.5 23-63	49.6 26-102

Fsm1, 120 states, 582 transitions

r=0	90%	95%	99%	100%
RW	102.1 8-430	178.3 16-735	330.6 16-1125	330.6 16-1125
Uniform	9.7 2-35	13.1 2-39	21.9 2-90	21.9 2-90
Approx 10	2.9 2-11	3.5 2-11	5.1 2-23	5.1 2-23
Approx 1000	3.0 2-8	3.0 2-8	3.5 2-8	3.5 2-8
Exact	3.3 2-9	3.3 2-9	3.7 2-9	3.7 2-9

Moesi2, 22 states, 43 transitions

r=0	50%	90%	95%	99%	100%
RW	11.5 6-18	80.6 45-153	122.8 73-247	260.8 120-514	342.4 156-649
Uniform	8.78 7-13	54.4 36-82	80.9 51-12	180.972-333	277.3 105-776
Approx 10	7.49 5-10	37.6 27-56	53.3 38-91	102.6 56-205	140.6 59-347
Approx 1000	7.6 6-11	35.8 24-51	50.9 32-80	88.11 58-147	106.8 62-179
Exact	7.6 5-10	34.9 24-46	47.6 33-63	82.3 55-121	101.5 56-165

Kanban1, 160 states, 1151 transitions

Fig. 3. Comparative results (1) for number of generated tests

counter automata (the parameter can be, for instance, the number of communicating processes). For several examples and parameters, the counter automaton has been filtered into a classical finite automaton. The list is given in Table 2: first column contains the name of the protocol with the value of the parameter.

r=0	90%		95%	
RW	54991.6 39414-69917		155803 117044-214680	
Uniform	937.1 827-1044		1493 1240-176	
Approx 10	789..9 718-884		1129.7 980-1292	
Approx 1000	704.7 651-759		974.1 892-1066	
Exact	708 655-769		698.8 867-1101	
	99%		100%	
RW	$10^6 10^5 - -10^7$		$10^7 10^6 - -10^7$	
Uniform	3405.7 2760-4186		11049 6563-22480	
Approx 10	1998.8 1707-2282		11962 3336-62763	
Approx 1000	1625.3 1441-1841		3410.8 2265-7144	
Exact	1610.5 1402-1843		3268.8 2369-4738	

Ttp8, 3201 states,
6765 transitions

r=0	90%	95%	99%
RW	505 248-850	1073 460 - 1777	5702.6 1395-16035
Uniform	67.6 47-92	106.6 73-160	226.6 115-471
Approx 10	53.05 40-73	73.3 55-106	134.1 90-226
Approx 1000	49.1 39-64	67.6 51.95	115.1 79-199
Exact	49.7 39-72	66.8 51-95	114.4 76-168

	100%
RW	21750 2579 - 119811
Uniform	372.3 165-809
Approx 10	193.3 97-439
Approx 1000	158.1 85-300
Exact	157.4 91-270

Prodcons10, 286 states, 600 transitions

Fig. 4. Comparative results (2) for number of generated tests

Table 2. Graphs used for benchmarking.

Name	States	Transitions	Eccentricity	Nb. of paths
Barber1	15	18	5	74
Berkeley3	1376	3974	51	1,33 10^{39}
Consistency3	806	1206	600	5,63 10^{153}
Csm1	24	57	8	934000
Dekker1	86	178	17	8,80 10^{11}
Dragon3	103	696	50	2,34 10^{93}
Fms1	120	582	14	1,41 10^{20}
Illinois3	103	307	100	2,23 10^{90}
Kanban1	160	1151	14	3,31 10^{20}
Lift3	499	587	302	7,24 10^{59}
Moesi2	22	43	11	3,84 10^8
Prodcons10	286	660	20	3,51 10^7
Ttp8	3201	6765	32	4,30 10^7

The second and the third columns respectively report on the number of states of the automaton and the number of transitions. The fourth column provides the eccentricity⁵ of the automaton, that is the maximal distance of an edge to the initial states. Finally, the last column gives the approximate number of successful paths in the automaton of length less than or equal to twice the eccentricity. Note that in these graphs all states are final.

3.2 Experimental Protocol

For each protocol, we have measured the number of tests/generated paths required to cover either 50%, or 90%, or 95%, or 99%, or 100% of the states. Several values close to 100% have been chosen since many biased approaches have been introduced to handle rare events, and many methods will efficiently cover 50% or 70% of the graph. It is harder to cover the remaining last states. We have compared 5 different approaches. First, the **RW** Approach consists in performing isotropic random walks in the automaton: once in a state, the next one is picked up uniformly among its neighbours. The path ends either when it reaches a dead-end state, or when its length is twice the eccentricity. The second approach, denoted **Uniform**, is the one introduced in [9]: paths of length bounded by twice the eccentricity are uniformly generated. The approach denoted **Exact** is the biased approach proposed in [9], where the linear system is exactly computed. The **Approx 10** and **Approx 1000** approaches are the ones proposed in this article: for 10 [resp. 1000] the $\alpha_{i,j}$'s are approximated using $10n$ [res. $1000n$] randomly generated paths, where n is the number of states.

Note that comparing the distribution π given by the exact approach and the approximation-based approaches is not easy. Indeed, a linear programming system may have different optimal solutions. Let us consider for instance the example depicted in Fig. 5. The set of successful paths visiting 3 is the same as the set of successfully paths visiting 4. Therefore, in any optimal solutions of the linear programming system given $\pi_3 = x$ and $\pi_4 = y$, one can do the following changes: $\pi_3 = z$ and $\pi_4 = t$ with $z + t = x + y$, and we also obtain an optimal solution.

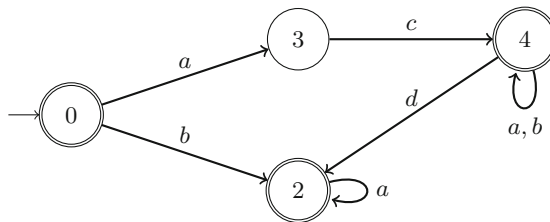


Fig. 5. Illustrating example with different optimal solutions.

⁵ Eccentricity is an important parameter since it is the minimal length required for paths to have a chance to visit each state.

3.3 Qualitative Experimental Results

Since the test generation procedures are randomized, performance is stochastic. For each example and each coverage proportion, each approach has been experimented 100 times. For each case, we report on the average number of tests obtained in order to cover the wanted proportion, but also the minimum number of tests (the best case), and the maximum number of tests (the worst case).

Results presented in Figs. 3 and 4 are obtained with $r = 0$: there is no a priori guarantee on the precision of the approximations. Results presented in Fig. 6 are obtained with $r = 0$ and with $r = 10$ (and in one case with $r = 50$). For instance, the second table in Fig. 3 reports on the result for Dekker1: in order to cover 95% of the set of the states, the RW approach requires on average 790 paths. In the best case (of the experiments), it only requires 159 paths, and in the worst case 2062 paths have been generated. For the same coverage, the Uniform approach requires 35.45 paths in average. The Exact approach only requires 15.5 paths in average.

Relatively to the other approaches, the performance of RW deeply depends on the topology of the automaton. For instance, for Prodcons10 or Ttp8 or Moesi2, RW is ugly, and requires much more tests to (partially) cover the set of states. For Fms1, RW is as efficient as Exact. For some automata, there is no result for RW: after hours of computation, the approach was not able to cover 90% of the set of states. In these cases, some states occur with a so low probability on random walks, that in practice it is not possible to generate a path visiting them.

One can see that for Barber1, Dekker1, Fms1, Moesi2, Kanban1, Ttp8 and Prodcons10, all biased approaches are better (cover the set of states with less paths) than the uniform one. Moreover, the Exact approach is better than the approximate ones, but not significantly with the Approx 1000. Consider for instance Fms1: the Uniform approach requires on average 87.6 paths to cover 99% of the states. With Approx 10 this number falls to 48.1, and it falls to 40.3 with Approx 1000. The Exact approach requires 29.8 paths on average.

The results for Lift3 are similar but the Approx 1000 is not so close to the Exact approach. For Berkeley3, Illinois3 and Dragon3, the Exact approach is clearly more efficient to cover the set of states. It is similar for Consistency3, but only for the 100% coverage criterion. A significant case is Illinois3: the Exact approach requires on average a unique path to cover all states, while the Approx 1000 approach requires 47 paths. For all these examples there is a huge number of paths, and many states j are visited with a very low probability by a path: the corresponding $\alpha_{i,j}$'s are set to zero since $r = 0$, thus providing a very bad approximation. For instance, for Illinois3, 84 states over the 103 states are not visited by any random paths. We run the experiment with Approx 1000 and $r = 10$. The obtained results are presented in Fig. 6: these results are much better and close to the ones of the Exact approach.

In conclusion, for the quality of the coverage, running Approx 10 with $r = 10$ seems to be an efficient solution.

r=0	90%	95%	99%	100%
Uniform	176 153-240	315 258-375	630 486-834	1203 837-2376
Approx 10	166 142-212	301 253-377	648 431-884	2127 797-5611
Approx 1000	171 259-381	307 259-381	673 485-1070	1973 807 - 4485
Exact	166 143 - 203	294 256-346	589 440-816	1084 689-1783
Approx 10 ($r = 10$)	168 140-231	297 256-376	626 479-780	1474.9 730 - 4749
Approx 1000 ($r = 10$)	168 133-211	300 258-359	624 489-828	1390 765 - 4042
Approx 10 ($r = 50$)	168 141-208	300 250-373	610 480-848	1292 710 - 2715

Consistency3, 806 states, 1206 transitions

r=0	90%	95%	99%	100%
Uniform	10.56 4-30	23 6-100	64.7 10-400	85.3 10-400
Approx 10	5.4 3-14	9 4-46	24.7 8-107	34.9 9-137
Approx 1000	4.4 3-7	6.6 4-23	13.9 6-37	18.9 7-51
Exact	3.9 3-6	5.5 4-9	9.5 6-20	13.1 6-24
Approx 10 ($r = 10$)	3.8 3-4	5.4 4-7	8.9 6-15	12.9 8-23
Approx 1000 ($r = 10$)	3.7 3-5	5.3 4-7	9.0 7-14	12.4 7-20

Lift3, 499 states, 587 transitions

r=0	90%	95%	99%	100%
Uniform	22.5 7-101	35.8 12-135	69.6 24-267	112.1 34-317
Approx 10	18.3 9-47	29.1 12-76	63.2 15-231	98.5 34-267
Approx 1000	14.7 6-29	25.2 11-55	56.7 13-184	87.2 13-319
Exact	10.8 6-17	15 7-25	22.8 10-41	29.9 10-57
Approx 10 ($r = 10$)	10.6 5-17	14.6 6-29	23.5 12-47	30.7 16-79
Approx 1000 ($r = 10$)	10.1 6-18	14.3 8-26	22.6 14-42	29.5 14-79

Dragon3, 103 states, 696 transitions

r=0	90%	95%	99%	100%
Uniform	337 235-448	594 400-863	1551 993-2262	4470 2316-9974
Approx 10	222 167-286	384.8 258-536	1014.6 655-1382	2886.6 1504-5251
Approx 1000	189 134-247	323 232-498	854 467-1302	2116 1108-2886
Exact	103 60-185	137.0 84-210	199.6 109-364	224.1 128-510
Approx 10 ($r=10$)	102 66-149	138 80-280	207.0 107-399	233.6 140-412
Approx 1000 ($r=10$)	102 67-202	137 86-256	206 112-423	231 122-508

Berkley3, 1376 states, 3974 transitions

r=0	90%	95%	99%	100%
Uniform	8.8 1-46	16 1-73	40.1 1-209	68.43 1-338
Approx 10	7.5 1-27	12.1 1-46	30.1 1-150	56.9 1-314
Approx 1000	6.2 1-38	11.25 1-61	29.1 1-210	48.3 1-234
Exact	1.0 1-2	1.0 1-2	1.0 1-2	1.1 1-2
Approx 10 ($r = 10$)	1.0 1-2	1.0 1-2	1.1 1-2	1.2 1-3
Approx 1000 ($r = 10$)	1.0 1-2	1.0 1-2	1.1 1-2	1.2 1-2

Illinois3, 1524 states, 307 transitions

Fig. 6. Comparative results (3)

Table 3. Time to compute the linear programming system.

(Seconds)	Berkeley3	Consistency3	Dragon3	Lift3	Illinois3
Exact	26401	40964	29	4337	18
Approx 10 ($r = 0$)	1	58	1	8	0.4
Approx 1000 ($r = 0$)	186	5794	21	813	39
Approx 10 ($r = 10$)	16	110	1	12	1
Approx 1000 ($r = 10$)	208	5890	25	862	41
Approx 10 ($r = 50$)	–	124	–	–	–

3.4 Computation Time

Let us note first that for all approaches, generating paths is done practically in a very efficient way. As mentioned before, the bottleneck step is the computation of the linear programming system. In Table 3, the time (in seconds) used to compute the linear programming system is given for the protocols Berkeley3, Consistency3, Dragon3, Lift3 and Illinois3. The results are similar for the other protocols. For Illinois3, using Approx 1000 is less efficient than using the Exact approach. The reason is that the automaton is quite small. However, for other cases, using the approximation-based approaches is faster. And it is significantly faster for large automata. For instance, for Consistency3, while the Exact approach requires more than 11 h, and only about 90 min are needed for Approx 1000 (with $r = 0$).

In all cases, the best compromise seems to use Approx 10 with $r = 10$: the computation time is strongly better, and the quality of the biased approach is similar to the Exact approach, except for Consistency. For this protocol, we run the Approx 10 with $r = 50$ and we obtain better results, closer to the Exact approach, with a very short computation time (about 2 min, in comparison to 11 h for the Exact approach).

3.5 Experiments on Large Graphs

We have experimented the approaches on a model of the Centralserver2 protocol, which has 2523 states and 18350 transitions, an eccentricity of 63, and about $8,04 \cdot 10^{113}$ successful paths of length less or equal to 126. By computing the first $\alpha_{i,j}$'s, we estimate that the computation time of the linear programming system with Exact will require about 200 days. The linear programming system with Approx 10 and Approx 1000 ($r = 10$) has been computed respectively in 81 s and in 24 min. The obtained qualitative results compared to Uniform are given in Table 4.

We also used the algorithm proposed in [6] to randomly generate two trim automata with respectively 5659 states (with 17007 transitions) and 11251 states (with 33753 transitions). The approximated linear programming system obtained by the Approx 10 and Approx 1000 approaches (with $r = 10$) has been computed in respectively 5.5 s and 613 s for the first graph, and in respectively 26.3 s and 1162 s for the second graph.

Table 4. Results for Centralserver2.

	90%	95%	99%	100%
Uniform	680 573–786	1198 947–1492	2942 2358–3612	9413 5487–19533
Approx 10 ($r = 10$)	316 287–349	476 437–529	878 775–1037	2065 1223–4337
Approx 1000 ($r = 10$)	313 281–345	467 415–515	864 729–1037	1926 1252–3514

4 Conclusion

In this paper we proposed an approximation-based approach for the random biased exploration of large models. It has been experimented on several examples: in practice the approximation is not too coarse, and the quality of the generated test suites to cover the states of the model is excellent compared to the exact approach and to the other random approaches. For computation time, using approximation is significantly better since the approach can be used on graphs with more than 10000 states. In the future we plan to investigate recent advances in optimization in order to improve the computation time.

References




1. Aichernig, B.K., Jöbstl, E., Kegele, M.: Incremental refinement checking for test case generation. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 1–19. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38916-0_1
2. Alberto, A.D.B., Cavalcanti, A., Gaudel, M.-C., Simão, A.: Formal mutation testing for circus. *Inf. Softw. Technol.* **81**, 131–153 (2017)
3. Bardin, S., Leroux, J., Point, G.: FAST extended release. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 63–66. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_9
4. Bernardi, O., Giménez, O.: A linear algorithm for the random sampling from regular languages. *Algorithmica* **62**(1–2), 130–145 (2012)
5. Bienaymé, I.-J.: Considérations à l’appui de la découverte de Laplace. *Comptes Rendus de l’Académie des Sciences* **37**, 309–324 (1853)
6. Carayol, A., Nicaud, C.: Distribution of the number of accessible states in a random deterministic automaton. In: 29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, Paris, France, 29th February–3rd March 2012. LIPIcs, pp. 194–205 (2012)
7. Chebishev, P.: Des valeurs moyennes. *Journal de mathématiques pures et appliquées* **12**, 177–184 (1867)
8. Dadeau, F., Héam, P.-C., Kheddami, R., Maatoug, G., Rusinowitch, M.: Model-based mutation testing from security protocols in HLPSP. *Softw. Test. Verif. Reliab.* **25**(5–7), 684–711 (2015)
9. Denise, A., Gaudel, M.-C., Gouraud, S.-D., Lassaingne, R., Oudinet, J., Peyronnet, S.: Coverage-biased random exploration of large models and application to testing. *STTT* **14**(1), 73–93 (2012)

10. Dreyfus, A., Héam, P.-C., Kouchnarenko, O.: Random grammar-based testing for covering all non-terminals. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, 18–22 March 2013, pp. 210–215. IEEE Computer Society (2013)
11. Dreyfus, A., Héam, P.-C., Kouchnarenko, O., Masson, C.: A random testing approach using pushdown automata. *Softw. Test. Verif. Reliab.* **24**(8), 656–683 (2014)
12. Duran, J.W., Ntafos, S.C.: A report on random testing. In: Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, 9–12 March 1981, pp. 179–183. IEEE Computer Society (1981)
13. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005, pp. 213–223. ACM (2005)
14. Groce, A., Joshi, R.: Random testing and model checking: building a common framework for nondeterministic exploration. In: Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), WODA 2008, Seattle, Washington, USA, 21 July 2008, pp. 22–28. ACM (2008)
15. Héam, P.-C., Masson, C.: A random testing approach using pushdown automata. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 119–133. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21768-5_10
16. Héam, P.-C., Nicaud, C.: Seed: an easy-to-use random generator of recursive data structures for testing. In: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, 21–25 March 2011, pp. 60–69. IEEE Computer Society (2011)
17. Hoeffding, W.: Probability inequalities for sums of bounded random variables. *J. Am. Stat. Assoc.* **58**(301), 13–30 (1963)
18. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines—a survey. *Proc. IEEE* **84**, 1090–1123 (1996)
19. Lima, B., Faria, J.P.: A model-based approach for product testing and certification in digital ecosystems. In: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, 11–15 April 2016, pp. 199–208. IEEE Computer Society (2016)
20. Mitzenmacher, M., Upfal, E.: Probability and Computing - Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005)
21. Oriat, C.: Jartège: a tool for random generation of unit tests for Java classes. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) QoSA/SOQUA - 2005. LNCS, vol. 3712, pp. 242–256. Springer, Heidelberg (2005). https://doi.org/10.1007/11558569_18
22. Oudinet, J., Denise, A., Gaudel, M.-C.: A new dichotomic algorithm for the uniform random generation of words in regular languages. *Theor. Comput. Sci.* **502**, 165–176 (2013)
23. Tang, J., Cui, X., Zhao, Z., Guo, S., Xu, X.-S., Hu, C., Ban, T., Mao, B.: NIVAnalyzer: a tool for automatically detecting and verifying next-intent vulnerabilities in android apps. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, 13–17 March 2017, pp. 492–499. IEEE Computer Society (2017)
24. Thummala, S., Offutt, J.: Using Petri nets to test concurrent behavior of web applications. In: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, 11–15 April 2016, pp. 189–198. IEEE Computer Society (2016)

25. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.* **22**(5), 297–312 (2012)
26. How Tai Wah, K.S.: Black-box testing: techniques for functional testing of software and systems. *Softw. Test. Verif. Reliab.* **6**(1), 49–50 (1996). By Beizer, B. Wiley (1995) (book review)
27. Wang, C., Pastore, F., Briand, L.C.: System testing of timing requirements based on use cases and timed automata. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, 13–17 March 2017, pp. 299–309. IEEE Computer Society (2017)



Static and Dynamic Verification of Relational Properties on Self-composed C Code

Lionel Blatter^{1,2} , Nikolai Kosmatov¹ , Pascale Le Gall²,
Virgile Prevosto¹ , and Guillaume Petiot¹

¹ CEA, LIST, Software Reliability and Security Lab,
PC 174, 91191 Gif-sur-Yvette, France

{lionel.blatter, nikolai.kosmatov, virgile.prevosto,
guillaume.petiot}@cea.fr

² CentraleSupélec, Université Paris-Saclay, 91190 Gif-sur-Yvette, France

{lionel.blatter, pascale.legall}@centralesupelec.fr

Abstract. Function contracts are a well-established way of formally specifying the intended behavior of a function. However, they usually only describe what should happen during a single call. Relational properties, on the other hand, link several function calls. They include such properties as non-interference, continuity and monotonicity. Other examples relate sequences of function calls, for instance, to show that decrypting an encrypted message with the appropriate key gives back the original message. Such properties cannot be expressed directly in the traditional setting of modular deductive verification, but are amenable to verification through *self-composition*. This paper presents a verification technique dedicated to relational properties in C programs and its implementation in the form of a FRAMA-C plugin called RPP and based on self-composition. It supports functions with side effects and recursive functions. The proposed approach makes it possible to prove a relational property, to check it at runtime, to generate a counterexample using testing and to use it as a hypothesis in the subsequent verification. Our initial experiments on existing benchmarks confirm that the proposed technique is helpful for static and dynamic analysis of relational properties.

Keywords: Relational properties · Specification · Self-composition
Deductive verification · Dynamic verification · Frama-C

1 Introduction

Context. Deductive verification techniques provide powerful methods for formal verification of properties expressed in Hoare Logic [11, 12]. In this formalization, also known as axiomatic semantics, a program is seen as a predicate transformer, where each instruction S executed on a state verifying a property P leads to a state verifying another property Q . This is summarized in the form of *Hoare*

triples as $\{P\}S\{Q\}$. In this setting, P and Q refer to states before and after a single execution of a program S . It is possible in Q to refer to the initial state of the program, for instance to specify that S has increased the value stored in variable x , but one cannot express properties that refer to two distinct executions of S , even less properties relating executions of different programs S_1 and S_2 . As will be seen in the next sections, such properties, that we will call *relational properties* in this paper, occur quite regularly in practice. Hence, it is desirable to provide an easy way to specify them and to verify that implementations are conforming to such specification. A simple example of a relational property is monotonicity of a function $f: x < y \Rightarrow \mathbf{f}(x) < \mathbf{f}(y)$.

Several theories and techniques exist for handling relational properties. First, Relational Hoare Logic [6] is mainly used to show the correctness of program transformations, *i.e.* the fact that the result of the transformation preserves the original semantics of the code. Then, Cartesian Hoare Logic [19] allows for the verification of k -safety properties, that is, properties over k calls of a function. The DESCARTES tool is based on Cartesian Hoare Logic and has been used to verify anti-symmetry, transitivity and extensionality of various comparison functions written in Java. A decomposition technique using abstract interpretation is presented in [1] for verification of k -safety properties. The method is implemented in a tool called BLAZER and used for verification of non-interference and absence of timing channel attacks. A relational program reasoning based on an intermediate program representation in LLVM is proposed by [13]. The method supports loops and recursive functions and is used for checking program equivalence. Finally, self-composition [3] and its refinement Program Products [2] propose theoretical approaches to prove relational properties by reducing the verification of relational properties to a standard deductive verification problem.

Motivation. In the context of the ACSL specification language [5] and the deductive verification plugin WP of FRAMA-C [14], the necessity to deal with relational properties has been faced in various verification projects. For example, we can extract the following quote from a work on verification of continuous monotonic functions in an industrial case study on smart sensor software [7] (emphasis ours):

After reviewing around twenty possible code analysis tools, we decided to use FRAMA-C, which fulfilled all our requirements (*apart from the specifications involving the comparison of function calls*).

The authors attempt to prove the monotonicity of some functions (*i.e.*, if $x \leq y$ then $f(x) \leq f(y)$) using FRAMA-C/WP plugin. To address the absence of support for relational properties in ACSL and WP, they perform a manual transformation [7] consisting in writing an additional function simulating the call to the related functions in the property. Broadly speaking, this amounts to manually perform self-composition. This technique is indeed quite simple and expressive enough to be used on many relational properties. However, applying it manually is relatively tedious, error-prone, and does not provide a completely automated link between three key components: (i) the specification of the property, (ii) the

proof that the implementation satisfies the property, and (iii) the ability to use the property as hypothesis in other proofs (of relational as well as non-relational properties). Thus, the lack of support for relational properties can be a major obstacle to a wider application of deductive verification in academic and industrial projects. Finally, another motivation of this work was to obtain a solution compatible with other techniques than deductive verification, notably dynamic analysis.

Contributions. To address the absence of support for expressing relational properties in ACSL and for verifying such properties in the FRAMA-C platform, we implemented a new plugin called RPP. This plugin allows the specification and verification of properties invoking any (finite) number of calls of possibly dissimilar functions with possibly nested calls, and to use the proved properties as hypotheses in other proofs. A preliminary version of RPP has been described in a previous short paper [8]. However, it suffered from major limitations. Notably, it could only handle pure, side-effect free functions, which in the context of the C programming language is an extremely severe constraint. Similarly, the original syntax to express relational properties is not expressive enough and requires some additional constructs, in order to properly specify relational properties of functions with side-effects. The previous work [8] did not address dynamic analysis of relational properties either.

The current paper will thus focus on the extensions that have been made to the original RPP design and implementation, as well as its evaluation. Its main contributions include:

- a new syntax for relational properties;
- handling of side effects;
- handling of recursive functions;
- evaluation of the approach over a suitable set of illustrative examples;
- experiments with runtime checking of relational properties and counterexample generation when a property cannot be proved in the context of RPP.

Outline. The remainder of this paper is organized as follows. First, in Sect. 2 we briefly recall the general idea of relational property verification with RPP in the case of pure functions using self-composition. Then, in Sect. 3, we show how to extend this technique to the verification of relational properties over functions with side effects (access to global variables and pointer dereference). Another extension, described in Sect. 4 allows considering recursive functions. We demonstrate the capacities of RPP by using it on the adaptation to C of the benchmark proposed for Java in [19] and our own set of test examples (Sect. 5). Finally, we show in Sect. 6 that RPP can also be used to check relational properties at runtime and/or to generate a counterexample using testing, and conclude in Sect. 7.

2 Context and Main Principles

RPP (Relational Property Prover) is a solution designed and implemented as a plugin of FRAMA-C [14], an extensible framework dedicated to the analysis of

C programs. FRAMA-C offers a specification language, called ACSL [5], and a deductive verification plugin, WP [4], that allow the user to specify the desired program properties as function contracts and to prove them. A typical ACSL function contract may include a precondition (**requires** clause stating a property that must hold each time the function is called) and a postcondition (**ensures** clause that must hold when the function returns), as well as a frame rule (**assigns** clause indicating which parts of the global program state the function is allowed to modify). **assigns** clauses may be refined by **\from** directives, indicating for each memory location l potentially modified by the function the list of memory locations that are read in order to compute the new value of l . Finally, an assertion (**assert** clause) can also specify a local property at any function statement.

WP is based on Hoare logic and generates Proof Obligations (POs) using Weakest Precondition calculus: given a property Q and a fragment of code S , it is possible to compute the minimal (weakest) condition P such that $\{P\}S\{Q\}$ is a valid Hoare triple. When S is the body of a function f , POs are formulas expressing that the precondition of f implies the weakest condition necessary for the postcondition (or assertion) to hold after executing S . POs can then be discharged either automatically by automated theorem provers (e.g. Alt-Ergo, CVC4, Z3¹) or with some help from the user *via* a proof assistant (e.g. Coq²).

FRAMA-C also offers an executable subset of ACSL, called E-ACSL [10, 18], that can be transformed into executable C code. It is thus compatible with dynamic analysis, such as runtime assertion checking of annotations using the E-ACSL plugin [10, 20] or with counterexample generation (in case of a proof failure) using the STADY plugin [16, 17].

Function contracts allow specifying the behavior of a single function call, that is, properties of the form “If $P(s)$ is verified when calling f in state s , $Q(s')$ will be verified when f returns with state s' ”. However, it is not possible to specify *relational properties*, that relate several function calls. Examples of such properties include monotonicity ($x < y \Rightarrow \mathbf{f}(x) < \mathbf{f}(y)$), anti-symmetry ($\mathbf{compare}(x, y) = -\mathbf{compare}(y, x)$) or transitivity ($\mathbf{compare}(x, y) \leq 0 \wedge \mathbf{compare}(y, z) \leq 0 \Rightarrow \mathbf{compare}(x, z) \leq 0$). RPP addresses this issue by providing an extension to ACSL for expressing such properties and a way to prove them. More specifically, RPP works like a preprocessor for WP: given a relational property and the definition of the C function(s) involved in the property, it generates a new function together with plain ACSL annotations whose proof (using the standard WP process) implies that the relational property holds for the original code. As we show below, this encoding of a relational property is also compatible with dynamic analysis (runtime verification or counterexample generation).

¹ See, resp., <https://alt-ergo.ocamlpro.com>, <http://cvc4.cs.nyu.edu>, <https://z3.codeplex.com/>.

² See <http://coq.inria.fr/>.

2.1 Original Relational Specification Language

For the specification of a relational property, we initially proposed an extension [8] of the ACSL specification language with a new clause, **relational**. These clauses are attached to a function contract. A property relating calls of different functions, such as R1 in Fig. 1a, must appear in the contract of the last function involved in the property, *i.e.* when all relevant functions are in scope. In this new clause we introduced a new construct `\call(f, <args>)` denoting the value returned by the call `f(<args>)` to `f` with arguments `<args>`. This allows relating several function calls in a **relational** clause. `\call` can be used recursively, *i.e.* a parameter of a called function can be the result of another function call. In Fig. 1a, properties R1 and R2 at lines 7–9 and 15–17 specify properties of functions `max` and `min` respectively.

Note however that the `\call` construct only allows speaking about the return value of a C function. If the function has some side effects, there is no way to express a relation between the values of memory locations that are modified by distinct calls. Section 3 describes the improvements that have been made to the initial version of the relational specification language in order to support side effects. To ensure that a function has no side effects, an **assigns \nothing** clause can be used.

```

1 /*@ requires x > INT_MIN;
2   assigns \nothing;
3   behavior pos:
4     assumes x ≥ 0;
5     ensures \result == x;
6   behavior neg:
7     assumes x < 0;
8     ensures \result == -x; */
9 int abs(int x){
10  return (x ≥ 0) ? x : (-x);
11 }
12
13 /*@ requires INT_MIN < x+y < INT_MAX;
14   assigns \nothing;
15   relational R1: ∀ int x,y;
16     \call(max,x,y) ==
17       (x+y+\call(abs,x - y))/2; */
18 int max(int x,int y){
19  return (x ≥ y) ? x : y;
20 }

```

(a) Original source code

```

1 /*@ axiomatic Relational_axiom {
2   logic int max_acsl(int x, int y);
3   logic int abs_acsl(int x);
4   lemma Relational_lemma{L}:
5     ∀ int x, int y;
6     max_acsl(x, y) ==
7       ((x + y) + abs_acsl(x - y)) / 2;
8 } */
9
10 void relational_wrapper(int x, int y){
11  int ret_var_1, ret_var_2;
12  ret_var_1 = (x ≥ y) ? x : y;
13  ret_var_2 = (x-y ≥ 0) ? x-y : -(x-y);
14  /*@ assert
15     ret_var_1 ==
16       ((x + y) + ret_var_2) / 2; */
17  return;
18 }
19
20 /*@ assigns \nothing;
21   behavior Relational_behavior:
22     ensures
23       \result ==
24         max_acsl(\old(x), \old(y));
25  */
26 int max(int x, int y){ ... }

```

(b) Excerpt of the code generated by RPP

Fig. 1. Pure function with relational properties

2.2 Preprocessing of a Relational Property

The previous work [8] also proposed a code transformation whose output can be analyzed with standard deductive verification tools. This is materialized in the

RPP plugin of FRAMA-C, that relies then on WP to prove the resulting standard ACSL annotations.

Going back to our example, applying the transformation to property R1 over function `max` gives the code of Fig. 1b. The generated code can be divided into three parts. First, a new function, called *wrapper*, is generated. The wrapper function is inspired by the workaround proposed in [7] and self-composition [3]. As in self-composition, this wrapper function inlines the calls occurring in the relational property under analysis, with a suitable renaming of local variables to avoid interferences between the calls.

In addition, the wrapper records the results of the calls in fresh local variables. Then, in the spirit of calculational proofs [15], we state an assertion equivalent to the relational property (lines 14–16 in Fig. 1b). The proof of such an assertion is possible with a classic deductive verification tool (WP with Alt-Ergo as back-end prover in our case).

However, the wrapper function only provides a solution to prove relational properties. The ability to use these properties as hypotheses in other proofs (relational or not) must be reached otherwise. For this purpose, RPP generates an ACSL axiomatic definition (cf. **axiomatic** section at lines 1–8 in Fig. 1b) introducing a logical reformulation of the relational property as a lemma (cf. lines 4–7) over otherwise unspecified logic functions (`max_acsl` and `abs_acsl`

```

1 /*@ assigns \nothing; */
2 int Crypt(int m, int key) {
3     return m + key;
4 }
5
6 /*@ assigns \nothing;
7     relational R3:
8     ∀ int m, key;
9     \call(Decrypt,
10          \call(Crypt, m, key),
11              key)
12     == m; */
13 int Decrypt(int m, int key) {
14     return m - key;
15 }
16
17 /*@ assigns \nothing;
18     ensures \result == m;
19     relational R4:
20     ∀ int m, key;
21     \call(run,
22          \call(run, m, key),
23              key)
24     == m; */
25 int run(int m, int key) {
26     int crypt, decrypt;
27     crypt = Crypt(m, key);
28     decrypt = Decrypt(crypt, key);
29     return decrypt;
30 }
    
```

(a) Original source code

```

1 /*@ axiomatic Relational_axiom {
2     logic int run_acsl(int m, int key);
3
4     lemma Relational_lemma{L}:
5     ∀ int m, int key;
6     run_acsl(
7         run_acsl(m, key),
8         key)
9     == m; } */
10
11 void relational_wrapper(int m, int key) {
12     int tmp_1, tmp_2, tmp_3, tmp_4;
13     tmp_1 = Crypt_aux_2(m, key);
14     tmp_2 = Decrypt_aux_2(tmp_1, key);
15     tmp_3 = Crypt_aux_2(tmp_2, key);
16     tmp_4 = Decrypt_aux_2(tmp_3, key);
17     /*@ assert tmp_4 == m; */
18     return; }
19
20 /*@ ensures \result == \old(m);
21     assigns \nothing;
22     behavior Relational_behavior:
23     ensures \result ==
24         run_acsl(\old(m), \old(key)); */
25 int run(int m, int key) {
26     int crypt;
27     int decrypt;
28     crypt = Crypt(m, key);
29     decrypt = Decrypt(crypt, key);
30     return decrypt; }
    
```

(b) Transformed code

Fig. 2. Functions `Crypt` and `Decrypt`, used by function `run`.

in the example). Furthermore, new postconditions are generated in the contracts of the C functions involved in the relational property. They specify that there is an exact correspondence between the original C function and its newly generated logical ACSL counterpart. Thanks to this axiomatic, POs over functions calling `max` and `abs` will have the lemma in their environment and thus will be able to take advantage of the proven relational property. Note that the correspondence between `max` and `max_acsl` (respectively `abs` and `abs_acsl`) can only be done because `max` and `abs` do not access global memory (neither for writing nor for reading). Indeed, since `max_acsl` and `abs_acsl` are pure logic functions, they do not have side effects and their result only depends on their parameters.

To illustrate the use of relational properties in the proof of other specifications, we can consider the postcondition and property R4 of function `run` of Fig. 2a (inspired by the PISCO project³) whose proof needs to use property R3. Thanks to their reformulation as lemmas and to the link between ACSL and C functions, WP automatically proves the assertion at line 17 (for property R4) and the postcondition at line 20 of Fig. 2b.

2.3 Soundness of the Transformation

Since our transformation is introducing an ACSL **axiomatic**, care must be taken to avoid introducing inconsistencies in the specification. More precisely, the **axiomatic** specifies the intended behavior of the ACSL counterpart of the C functions under analysis. The corresponding ACSL functions are then only used in the contracts of those C functions. In particular, since the wrapper is inlining the body of the functions concerned by the relational property, the **lemma** of the **axiomatic** cannot be used to prove the **assert** annotation inside the wrapper.

3 Functions with Side Effects

As mentioned above, the initial RPP approach only works for relational properties over pure functions. More precisely, it allows proving relational properties of the form:

$$\forall \langle \text{args1} \rangle, \dots, \forall \langle \text{argsN} \rangle, \\ P(\langle \text{args1} \rangle, \dots, \langle \text{argsN} \rangle, \backslash \text{call}(f_1, \langle \text{args1} \rangle), \dots, \backslash \text{call}(f_N, \langle \text{argsN} \rangle))$$

for an arbitrary predicate P invoking $N \geq 1$ calls of non-recursive functions without side effects. In the context of the C programming language, handling only pure functions is a major limitation. We thus propose an extension of both the specification language and the transformation technique in order to let RPP tackle a wider, more representative, class of C functions.

³ See <http://www.projet-pisco.fr/>.

3.1 New Grammar for Relational Properties

Relational properties are still introduced by a **relational** clause inside an ACSL contract. However, since we might now refer to memory locations in either the pre- or the post-state of any call implied in the relational property, we need to be able to make explicit references to these states, and not only to the value returned by a given call. Although more verbose, the new syntax can also be used for pure functions. For instance, property R1 of Fig. 1a can be rewritten as shown in Fig. 3.

More generally, we introduce the grammar shown in Fig. 4. A relational clause is composed of three parts. First, we declare a set of universally quantified variables, that will be used to express the arguments of the calls that are related

```

1 /*@ assigns \result \from x, y;
2   relational R1:
3     \forall int x1, y1;
4       \callset(\call(max, x1, y1, id1), \call(abs, x1 - y1, id2)) ==>
5         \callresult(id1) == (x1 + y1 + \callresult(id2)) / 2;
6 */
7 int max(int x, int y) { ... }

```

Fig. 3. Annotated C function with **relational** annotations

$\langle \text{call-id} \rangle ::= \text{id}$	$\langle \text{literal} \rangle ::= \backslash \text{true} \mid \backslash \text{false} \mid \text{int} \mid \text{float}$
$\langle \text{bin-rel} \rangle ::= == \mid != \mid < \mid > \mid < = \mid > = \mid < \mid >$	$\langle \text{relational-label} \rangle ::= \text{Post_} \langle \text{call-id} \rangle$ $\mid \text{Pre_} \langle \text{call-id} \rangle$
$\langle \text{function-parameter} \rangle ::= \langle \text{relational-call-terms} \rangle +$	$\langle \text{bin-op} \rangle ::= + \mid - \mid * \mid / \mid \backslash$
$\langle \text{function-name} \rangle ::= \text{poly-id}$	$\langle \text{result-reference} \rangle ::= \backslash \text{callresult} (\langle \text{call-id} \rangle)$
$\langle \text{function-call} \rangle ::= \backslash \text{call} (\langle \text{inlining-option} \rangle ,$ $\langle \text{function-name} \rangle ,$ $\langle \text{function-parameter} \rangle ,$ $\langle \text{call-id} \rangle)$	$\langle \text{pure-function-parameter} \rangle ::= \langle \text{relational-call-terms} \rangle +$
$\langle \text{call-parameter} \rangle ::= \langle \text{function-call} \rangle +$	$\langle \text{inlining-option} \rangle ::= \text{int}$
$\langle \text{relational-def} \rangle ::= \backslash \text{callset} (\langle \text{call-parameter} \rangle)$	$\langle \text{pure-function-name} \rangle ::= \text{poly-id}$
$\langle \text{relational-pred} \rangle ::= \backslash \text{true} \mid \backslash \text{false}$ $\mid \langle \text{relational-terms} \rangle \langle \text{bin-rel} \rangle \langle \text{relational-terms} \rangle$ $\mid \langle \text{relational-pred} \rangle \ \&\& \ \langle \text{relational-pred} \rangle$ $\mid \langle \text{relational-pred} \rangle \ \parallel \ \langle \text{relational-pred} \rangle$ $\mid \langle \text{relational-pred} \rangle ==> \langle \text{relational-pred} \rangle$ $\mid ! \langle \text{relational-pred} \rangle$ $\mid \backslash \text{forall} \langle \text{binders} \rangle ; \langle \text{relational-pred} \rangle$ $\mid \backslash \text{exists} \langle \text{binders} \rangle ; \langle \text{relational-pred} \rangle$	$\langle \text{pure-function-call} \rangle ::=$ $\backslash \text{callpure} (\langle \text{inlining-option} \rangle ,$ $\langle \text{pure-function-name} \rangle , \langle \text{pure-function-parameter} \rangle)$
$\langle \text{relational-annot} \rangle ::= \text{relational} \langle \text{relational-clause} \rangle$	$\langle \text{relational-call-terms} \rangle ::= \langle \text{literal} \rangle$ $\mid \langle \text{pure-function-call} \rangle$ $\mid \langle \text{relational-call-terms} \rangle \langle \text{bin-op} \rangle \langle \text{relational-call-terms} \rangle$
$\langle \text{relational-clause} \rangle ::=$ $\backslash \text{forall} \langle \text{binders} \rangle ;$ $\langle \text{relational-def} \rangle ==> \langle \text{relational-pred} \rangle$	$\langle \text{relational-terms} \rangle ::= \langle \text{literal} \rangle$ $\mid \langle \text{relational-terms} \rangle \langle \text{bin-op} \rangle \langle \text{relational-terms} \rangle$ $\mid \langle \text{result-reference} \rangle$ $\mid \backslash \text{at} (\langle \text{poly-id} \rangle , \langle \text{relational-label} \rangle)$ $\mid \langle \text{pure-function-call} \rangle$

(a) Grammar of relational predicates

(b) Grammar of relational terms

Fig. 4. Grammar for relational properties

by the clause. Then, we specify the set of calls on which we will work in the *relational-def* part. As shown in Fig. 4a, each call is then associated to an identifier *call-id*. In the property R1 of Fig. 3, two function calls are explicitly specified in the `\callset` construct and not directly in the predicate. Each call has its own identifier (`id1` and `id2` respectively). Finally, the relational property itself is given as an ACSL predicate in the *relational-pred* part. As described in Fig. 4a, in addition to standard ACSL constructs, three new terms can be used. First, `\callpure` can be used to indicate the value returned by a pure function as was done with the `\call` built-in in the original version of RPP. This allows specifying relational properties over pure functions without the overhead required for handling side-effects. As before, nested `\callpure` are allowed. Second, `\callresult`, as used in Fig. 3, takes a *call-id* as parameter and refers to the value returned by the corresponding call in *relational-def*. Finally, each such *call-id* gives rise to two logic labels. Namely, `Pre_call-id` refers to the pre-state of the corresponding call, and `Post_call-id` to its post-state. These labels can in particular be used in the ACSL term `\at(e,L)` that indicates that the term `e` must be evaluated in the context of the program state linked to logic label `L`. Figure 5a below shows an example of their use.

3.2 Global Variables Accesses

As said before, the new syntax for relational properties enables us to speak about the value of global variables at various states of the execution, thanks to the newly defined logic labels bound to each call involved in the `\callset` of the property. This is for instance the case in the relational property of Fig. 5a, which indicates that `h` is monotonic with respect to `y`, in the sense that if a first call to `h` is done in a state `Pre_id1` where the value of `y` is strictly less than in the pre-state `Pre_id2` of a second call, this will also be the case in the respective post-states `Post_id1` and `Post_id2`.

Generation of the wrapper function is more complicated in presence of side-effects. As presented in [3], each function call must operate on its own memory state, separated from the other calls in order for self-composition to work. We thus create as many duplicates of global variables as needed to let each part of the wrapper use its own set of copies. However, to avoid useless copies, RPP requires that each function involved in a relational property has been equipped with a proper set of ACSL `assigns` clauses, including `\from` components. This constraint is similar to what is proposed in [9], and ensures that only the parts of the global state that are accessed (either for writing or for reading) by the functions under analysis are subject to duplication. As an example, the wrapper function corresponding to our `h` function of Fig. 5a is shown in lines 24–33 of Fig. 5b.

Finally, the generated axiomatic definition enabling the use of the relational property in other POs must also be modified. The original transformation uses a logic function that is supposed to return the same `\result` as the C function. However, since logic functions are always pure, this mechanism is not sufficient to characterize side effects in the logic world. Instead, we declare a predicate

```

1 int y;
2
3 /*@ assigns y \from y;
4   relational R1:
5     \callset (\call(h, id1),
6               \call(h, id2))
7     ==>
8     \at(y, Pre_id1) < \at(y, Pre_id2)
9     ==>
10    \at(y, Post_id1) < \at(y, Post_id2);
11 */
12 void h() {
13   int a = 10;
14   y = y + a;
15   return;
16 }

```

(a) Annotated C function with **relational** annotations

```

1 int y;
2
3 /*@ axiomatic Relational_axiom_1 {
4   predicate
5     h_acsl(int y_pre, int y_post);
6
7   lemma Relational_lemma_1:
8     \forall int y_id2_pre, y_id2_post,
9           y_id1_pre, y_id1_post;
10    h_acsl(y_id2_pre, y_id2_post)
11    ==> h_acsl(y_id1_pre, y_id1_post)
12    ==> y_id1_pre < y_id2_pre
13    ==> y_id1_post < y_id2_post; }*/
14
15 /*@ assigns y \from y;
16   behavior Relational_behavior_1:
17     ensures h_acsl(\at(y, Pre),
18                  \at(y, Post)); */
19 void h(void) { ... }
20
21 int y_id1;
22 int y_id2;
23
24 void relational_wrapper_1(void) {
25   int a_1 = 10;
26   y_id1 = y_id1 + a_1;
27   int a_2 = 10;
28   y_id2 += y_id2 + a_2;
29   /*@ assert Rpp:
30     \at(y_id1, Pre) < \at(y_id2, Pre) ==>
31     \at(y_id1, Here) < \at(y_id2, Here); */
32   return;
33 }

```

(b) Transformed code for verification and use of relational properties with side effect

Fig. 5. Relational property on a function with side-effect

that takes as parameters not only the returned value and the formal parameters of the C function, but also the relevant parts of the program states that are involved in the property. As for the wrapper function, these additional parameters are inferred from the **assigns ... \from ...** clauses of the corresponding C functions. For instance, predicate `h_acsl`, on line 5 of Fig. 5b, takes two arguments representing the values of `y` before and after and execution of `h`. This link between the ACSL predicate and the C function is again materialized by an **ensures** clause (lines 17–18). The lemma defining the ACSL predicate is more complex too, since we have to quantify over the values of all the global variables at all relevant program states. In the example, this is shown on lines 7–13, where we have 4 quantified variables representing the value of global variable `y` before and after both calls involved in the relational property.

3.3 Support of Pointers

In the previous section, we have shown how to specify relational properties in presence of side effects over global variables, and how the transformations for both proving and using a property are performed. However, support of pointer

```

1 /*@ assigns *y \from *y;
2 relational R1:
3 \callset (
4   \call(k, id1),
5   \call(k, id2))
6 ==>
7   \at(*y, Pre_id1) <
8   \at(*y, Pre_id2)
9 ==>
10  \at(*y, Post_id1) <
11  \at(*y, Post_id2);
12 */
13 void k(int *y) {
14   *y = *y + 1;
15   return;
16 }

17 /*@ axiomatic Relational_axiom_1 {
18 predicate
19 k_acsl{pre, post}(int *y)
20 reads \at(*y, post), \at(*y, pre);
21
22 lemma Relational_lemma_1
23 {pre_id2, post_id2, pre_id1, post_id1}:
24 \forall int *y_id2, int *y_id1;
25 \separated(y_id1, y_id2)
26 ==> k_acsl{pre_id2, post_id2}(y_id2)
27 ==> k_acsl{pre_id1, post_id1}(y_id1)
28 ==> \at(*y_id1, pre_id1) < \at(*y_id2, pre_id2)
29 ==> \at(*y_id1, post_id1) < \at(*y_id2, post_id2);
30 }*/
31
32 /*@ assigns *y \from *y;
33 behavior Relational_behavior_1:
34 ensures k_acsl{Pre, Post}(y);*/
35 void k(int *y) { ... }
36
37 /*@ requires \separated(y_id1, y_id2);*/
38 void relational_wrapper_1(int *y_id1, int *y_id1) {
39   *y_id1 = *y_id1 + 1;
40
41   *y_id2 = *y_id2 + 1;
42
43   /*@ assert Rpp:
44   \at(*y_id1, Pre) < \at(*y_id2, Pre) ==>
45   \at(*y_id1, Here) < \at(*y_id2, Here);*/
46   return;
47 }

```

(a) Original annotated C function

(b) Code transformation

Fig. 6. Relational property in presence of pointers

dereference is more complicated. Again, as proven in [3] Self-Composition works if the memory footprint of each call is separated from the others. Thus, in order to adapt our method, we must ensure that pointers that are accessed during two distinct calls point to different memory locations. As above, such accesses are given by **assigns ... \from ...** clauses in the contract of the corresponding C functions. An example of a relational property on a function `k` using pointers (monotonicity with respect to the content of a pointer) is given in Fig. 6a, where `k` is specified to assign `*y` using only its initial content.

Memory separation is enforced using ACSL’s built-in predicate **\separated**. For the wrapper function, we add a **requires** clause stating the appropriate **\separated** locations. This can be seen on Fig. 6b, line 20, where we request that the copies of pointer `y` used for the inlining of both calls to `k` points to two separated area in the memory. Similarly, in the axiomatic part, the lemma adds separation constraints over the universally quantified pointers (line 9 in the Fig. 6b).

We also need to refine the declaration of the predicate in presence of pointer accesses. First, the predicate now needs to explicitly take as parameters the pre- and post-states of the C function. In ACSL, this is done by specifying *logic labels* as special parameters, surrounded by braces, as shown in line 3 of Fig. 6b. Second, a **reads** clause allows one to specify the footprint of the predicate, that is, the set of memory accesses that the validity of the predicate depends on

(line 4). Similarly, the lemma on lines 6–13 takes 4 logic labels as parameters, since it relates two calls to `k`, each of them having a pre- and a post-state.

It should be noted that the memory separation assumption makes the tool verify relational properties without pointer aliasing. Support of properties with pointer aliasing is left as future work.

4 Recursive Functions

We have shown in the previous section how we handle functions with side effects. Let us now focus on another class of functions, namely recursive functions. Support for recursive functions in RPP is interesting because it is very natural to specify such functions with relational properties. For example, a naive specification of a `fact` function computing the factorial of an integer can be written as

$$\left\{ \begin{array}{l} \forall x. x \leq 1 \implies \text{fact}(x) = 1, \\ \forall x. x > 1 \implies \text{fact}(x) = \text{fact}(x - 1) * (x) \end{array} \right.$$

The corresponding relational properties are given in Fig. 7a. The proof of the Induction property requires a modification to the generation of the wrapper function, that can be observed in Fig. 7b. Indeed, we do not want to inline the second call to `fact` on line 12, in order to take advantage of the fact that, since `fact` is a pure function that does not read anything from the global environment, this call returns the same value as the one of line 9, obtained by inlining the call to `fact(x1)`. This is why, as was indicated on Fig. 4, there is an optional argument to the `\callpure` construct, that indicates the maximal depth that the inlining can reach in the wrapper. The default value of 1, which is also used

<pre> 1 /*@ assigns \result \from x; 2 relational Base: 3 \forall int x1; 4 x1 <= 1 ==> 5 \callpure(1, fact, x1) == 1; 6 relational Induction: 7 \forall int x1; 8 x1 > 1 ==> 9 \callpure(1, fact, x1) == 10 \callpure(0, fact, x1-1)*x1; 11 */ 12 int fact(int x) { 13 if(x <= 1) { 14 return 1; 15 } 16 else { 17 return fact(x-1)*x; 18 } 19 }</pre>	<pre> 1 void relational_wrapper_2(int x1){ 2 int return_var_rela_2; 3 int return_var_rela_3; 4 { 5 if (x1 <= 1) { 6 return_var_rela_2 = 1; 7 } 8 else { 9 return_var_rela_2 = fact(x1-1)*x1; 10 } 11 } 12 return_var_rela_3 = fact(x1-1); 13 /*@ assert Rpp: 14 x1 > 1 ==> 15 return_var_rela_2 == 16 return_var_rela_3*x1; 17 */ 18 return; 19 }</pre>
---	---

(a) Annotated recursive C function with relational clauses

(b) Code transformation for the proof of the second relational property

Fig. 7. Relational property on recursive C function without side effects


```

1 int r;
2
3 /*@ requires x >= 0;
4   assigns r \from r,x;
5   relational \forallall int x1;
6   \callset(\call(1, fact, x1, id1)) ==> x1 <= 1 ==> \at(r, Post_id1) == 1;
7   relational \forallall int x1;
8   \callset(\call(1, fact, x1, id2), \call(1, fact, x1-1, id3))
9   ==> x1 > 1 ==> \at(r, Post_id2) == \at(r, Post_id3)*x1;
10 */
11 void fact(int x) {
12   if(x <= 1){
13     r = 1;
14     return;
15   }
16   else{
17     fact(x-1);
18     r = r * x;
19     return;
20   }
21 }

```

Fig. 8. Relational property on recursive C function with side effects

explicitly in our example for the first call, on line 9 of Fig. 7a, means that we inline the body of the function once (i.e. if the function calls other functions, including itself, these calls themselves will not be inlined). When this parameter is set to 0, as is the case for the second call in our example (line 10), we keep the call as such in the wrapper.

Support for recursive functions is not limited to pure functions. Recursive functions with side effects can also be handled. In particular, as shown in the grammar, each `\call` appearing in a `\callset` can also have an inlining directive. For instance, we can consider another implementation of the factorial, whose result is this time recorded in a global variable `r` (Fig. 8). The corresponding relational properties (lines 5–9) are similar to the pure case. However, the proof is slightly different, since the function has side effects, we cannot use logic function equality. Instead, we use the relational property as an induction hypothesis and inline both functions.

Note that in this case, a call to the function itself appears in the wrapper, contrarily to the situation detailed in Sect. 2.3. However, under the assumption that the function always terminates, this call is performed on arguments that are strictly smaller than the ones of the wrapper itself. Hence, the **axiomatic** can be used as an induction hypothesis in the sense that the wrapper allows us to prove that if the relational property holds for arguments smaller than `x`, then it holds for `x`.

5 Illustrative Examples

We have seen how to express relational properties over a large class of C functions and how RPP can generate C code and plain ACSL specifications for proving and using these properties through a standard WP process. To check that this

approach works in practice, we have tested our tool on different benchmarks. These tests aim at confirming:

- the ability to specify various relational properties over a large class of functions;
- the capacity to prove and use such properties using the generated transformation;
- the support of a large range of function implementations;
- the ability to use other techniques (runtime checks, test generation for invalidating the property) when WP fails to discharge a corresponding PO.

The first subsection will present our own benchmark composed of a mix of different types of relational properties. This benchmark is mainly designed to validate the two first items. The second subsection will show how RPP has performed on the benchmark proposed in [19]. This will confirm the second and third points. Finally, we will present in Sect. 6 our use of the E-ACSL and STADY plugins assessing the last point.

5.1 Internal Examples

As stated previously, we have tested RPP on a set of relational properties extracted from real case studies. This includes in particular encryption, as presented in Sect. 2, monotonicity (Sect. 3) or the factorial of Sect. 4, but also properties found in map/reduce, as the one in row 6 in Fig. 9, stating that the choice of the partitioning for the initial set of data should not play a role in the final result. The benchmark is also composed of more academic examples like linear algebraic properties of matrices, over functions containing loops

Num	Relational Property	Specified / Generated	Verified	Used	Side effect	Loop	Recursive
1	$\forall x1, x2 \in \mathbb{Z} : x1 < x2 \Rightarrow f(x1) < f(x2)$	✓	✓	✓	✓	✗	✗
2	$\forall x; f(x + 1) = f(x) * (x + 1)$	✓	✓	✓	✓	✗	✓
3	$\forall x, f_1(x) \leq f_2(x) \leq f_3(x)$	✓	✓	✗	✗	✗	✗
4	$\forall x, f(f(x)) = f(x)$	✓	✓	✗	✗	✓	✗
5	$\forall Msg, Key; Decrypt(Encrypt(Msg, Key), Key) = Msg$	✓	✓	✓	✓	✓	✗
6	$\forall t, sub_{t1}, \dots, sub_{tn}; t = sub_{t1} \cup \dots \cup sub_{tn} \Rightarrow max(t) = max(max(sub_{t1}), \dots, max(sub_{tn}))$	✓	✓	✗	✓	✓	✗
7	$\forall A, B; (A + B)^T = (A^T + B^T)$	✓	✓	✗	✗	✓	✗
8	$\det(A) = \det(A^T)$	✓	✓	✗	✗	✓	✗
9	$\forall x1, x2, y, f(x1 + x2, y) = f(x1, y) + f(x2, y)$	✓	✓	✓	✗	✗	✓
10	$\forall a, b, c, Med(a, b, c) = Med(a, c, b)$	✓	✓	✗	✗	✗	✗

Fig. 9. Summary of relational properties considered by RPP

(rows 7 and 8), or the property of row 10, that states the symmetry of the median of three numbers.

Figure 9 summarizes the results obtained on the benchmark. The first three columns indicate respectively whether the corresponding property could be specified and the corresponding code transformation generated, proved and used as an hypothesis in other proofs. The last three columns show what kind of C constructs are used in the implementation of the functions under analysis, namely side effects, presence of loops (which are always difficult for WP-related verification techniques, due to the need for loop invariants), and presence of recursive functions.

5.2 Comparator Functions

We also evaluated RPP on the benchmark proposed in [19]. It is composed of a collection of flawed and corrected implementations of comparators over a variety of data types written in Java, inspired from a collection of Stackoverflow⁴ questions. Translating the Java code into C was straightforward and fully preserved the semantics of the functions. We focused on the same properties as [19], that is anti-symmetry (P1), transitivity (P2) and extensionality (P3). Mathematically, these properties can be expressed as such:

$$P1 : \forall s1, s2. \text{compare}(s1, s2) = -\text{compare}(s2, s1)$$

$$P2 : \forall s1, s2, s3. \text{compare}(s1, s2) > 0 \wedge \text{compare}(s2, s3) > 0 \\ \Rightarrow \text{compare}(s1, s3) > 0$$

$$P3 : \forall s1, s2, s3. \text{compare}(s1, s2) = 0 \Rightarrow (\text{compare}(s1, s3) = \text{compare}(s2, s3))$$

Results are depicted in Fig. 10. For each comparator, we indicate whether the properties P1, P2 and P3 hold according to RPP (✓ and ✗ show whether the property was proved valid by WP). We get similar results as [19], with the exception of PokerHand, for which the generated wrapper function seems currently out of reach for WP (limits of scalability due to the combinatorial explosion of self-composition). However, by rewriting the function in a more modular way, WP was able to handle the example.

6 Dynamic Verification

6.1 Counterexample Generation

For the properties that do not hold in the comparator benchmark, we have been able to find counterexamples thanks to the proposed encoding of a relational property by self-composed code and using another FRAMA-C plugin, STADY [17]. STADY⁵ is a testing-based counterexample generator. In particular, STADY

⁴ <https://stackoverflow.com>.

⁵ See <https://github.com/gpetiot/Frama-C-StaDy>.

Benchmark	Proof (WP)			Counterex. gen. (STADY)		
	P1	P2	P3	P1	P2	P3
ArrayInt-false.c	✓	✓	✗	–	–	✓
ArrayInt-true.c	✓	✓	✓	–	–	–
CatBPos-false.c	✗	✗	✗	✓	✓	✓
Chromosome-false.c	✓	✗	✗	–	✂	✓
Chromosome-true.c	✓	✓	✓	–	–	–
CollItem-false.c	✗	✗	✗	✓	✓	✓
CollItem-true.c	✓	✓	✓	–	–	–
Contact-false.c	✓	✗	✗	–	✓	✓
Container-false-v1.c	✗	✓	✓	✓	–	–
Container-false-v2.c	✗	✗	✗	✓	✓	✓
Container-true.c	✓	✓	✓	–	–	–
DataPoint-false.c	✗	✗	✗	✓	✓	✓
FormItem-false.c	✓	✓	✗	–	–	✓
FormItem-true.c	✓	✓	✓	–	–	–
IsoSprite-false-v1.c	✗	✗	✗	✓	✓	✓
IsoSprite-false-v2.c	✗	✗	✓	✓	✓	–
Match-false.c	✗	✓	✗	✓	–	✓
Match-true.c	✓	✓	✓	–	–	–
NameComparator-false.c	✗	✓	✓	✓	–	–
NameComparator-true.c	✓	✓	✓	–	–	–
Node-false.c	✓	✓	✗	–	–	✓
Node-true.c	✓	✓	✓	–	–	–
NzbFile-false.c	✗	✓	✓	✓	–	–
NzbFile-true.c	✓	✓	✓	–	–	–
PokerHand-false.c	✓	✗	✗	–	✂	✂
PokerHand-true.c	✓	✓	✓	–	–	–
Solution-false.c	✓	✓	✗	–	–	✓
Solution-true.c	✓	✓	✓	–	–	–
TextPosition-false.c	✓	✗	✗	–	✓	✓
TextPosition-true.c	✓	✓	✓	–	–	–
Time-false.c	✗	✓	✓	✓	–	–
Time-true.c	✓	✓	✓	–	–	–
Word-false.c	✗	✗	✓	✓	✓	–
Word-true.c	✓	✓	✓	–	–	–

Fig. 10. Comparator properties analysed with WP and STADY after RPP translation

tries to find an input vector that will falsify an ACSL annotation for which WP could not decide whether it holds, thereby showing that the code is not conforming to the specification.

We apply STADY to try to find a test input such that the **assert** clause at the end of the *wrapper* function is false. The results are shown in the STADY columns of Fig. 10. Obviously, STADY does not try to find counterexamples for properties that are proved valid by WP. For properties that are not proved valid, ✓ indicates that a counterexample is found (within a timeout of 30 s), while ✂ indicated the only case where a counterexample is not generated before a 30-s timeout. A longer timeout (60 min) did not improve the situation in that

case. Symbol \otimes denotes two cases where the code translation uses features that are currently not yet supported by STADY. As shown in the table, thanks to the RPP translation, STADY was able to find counterexamples for almost all unproven properties. Notice that some examples required minor modifications so that STADY can be used. To be able to use testing, we had of course to add bodies for unimplemented functions. Other modifications consisted in reducing the input space to a representative smaller domain (by limiting the size of an input array) for some examples to facilitate counterexample generation [17].

6.2 Runtime Assertion Checking

The code transformation technique of RPP also enables runtime verification of relational properties through the E-ACSL plugin [10,20]. More precisely, the E-ACSL plugin translates ACSL annotations into C code that will check them at runtime and abort execution if one of the annotations fails. We tested the E-ACSL plugin on the test inputs generated by STADY in order to check that each generated counterexample does indeed violate the relational property. As expected, the obtained results validate those of the previous section. Since counterexample generation with STADY [17] basically includes a runtime assertion checking step for each test datum considered during the test generation process, we do not present the results of this step in separate columns.

7 Conclusion and Future Work

We have presented a major extension to an existing verification technique for relational properties, implemented in the FRAMA-C plugin RPP. The extension adds support for functions with side effects (access to global variables and pointer dereferences) and recursive functions. RPP relies on FRAMA-C/WP for automatic or interactive proof of the relational properties and offers the ability to use them as hypothesis in other proofs. Moreover, beyond WP, RPP also allows users to take advantage of E-ACSL and STADY plugins to verify relational properties at runtime and to produce a test input exhibiting the issue when a function does not respect the specified relational property. We have also shown that our implementation can handle a wide variety of properties and code: we consider a large class of relational properties with several, possibly nested, function calls.

However, there are still some limitations, inherent to our use of sequential self-composition. First, in the case of relational properties linking functions with large bodies or a large number of functions, the size of the generated wrapper function may explode, leading to POs that cannot be handled by automated theorem provers or even generated by weakest precondition calculus. A first solution for this problem is to use the modularity of the approach to reduce the size of the function and prove sub-properties. However, it is not always possible to modify an existing implementation. Alternative methods, based on a generalization of the technique proposed in [9] for verifying `\from` clauses, and that do not rely on the generation of a wrapper function seem thus desirable. The notation of

relational properties in the presence of side effects can be seen somewhat heavy to use. To make this notation more succinct, some shorthands for most common usages will be useful. The possibility to use runtime verification and testing is an important benefit in situations where the proof does not conclude. Furthermore, treatment of loops needs to be improved. In particular, it is not possible yet to specify “relational invariants” that would allow relating the behavior of a loop in two different contexts, while this is often necessary to complete the proof of a relational property. Solutions based on program products [2] look promising. Finally, as already mentioned, we need to extend our technique to handle potential aliases across the executions involved in a relational property.

Acknowledgement. The authors thank the FRAMA-C and PATHCRAWLER teams for providing the tools and support. Special thanks to François Bobot, Loïc Correnson, and Nicky Williams for many fruitful discussions, suggestions and advice. Many thanks to the anonymous referees for their helpful comments.

References

1. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proceedings of the 38th SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017), pp. 362–375. ACM (2017)
2. Barthe, G., Crespo, J.M., Kunz, C.: Product programs and relational program logics. *J. Log. Algebr. Methods Program.* **85**(5), 847–859 (2016)
3. Barthe, G., D’Argenio, P.R., Rezk, T.: Secure information flow by self-composition. *J. Math. Struct. Comput. Sci.* **21**(6), 1207–1252 (2011)
4. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z.: WP Plugin Manual v1.0 (2017). <http://frama-c.com/download/frama-c-wp-manual.pdf>
5. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. <http://frama-c.com/acsl.html>
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004), pp. 14–25 (2004)
7. Bishop, P.G., Bloomfield, R.E., Cyra, L.: Combining testing and proof to gain high assurance in software: a case study. In: 24th International Symposium on Software Reliability Engineering (ISSRE 2013), pp. 248–257. IEEE (2013)
8. Blatter, L., Kosmatov, N., Le Gall, P., Prevosto, V.: RPP: automatic proof of relational properties by self-composition. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 391–397. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_22
9. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V.: Functional dependencies of C functions via weakest pre-conditions. *Int. J. Softw. Tools Technol. Transf. (STTT)* **13**(5), 405–417 (2011)
10. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Proceedings of the ACM Symposium on Applied Computing (SAC 2013), pp. 1230–1235. ACM (2013)
11. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of the American Mathematical Society Symposia on Applied Mathematics, vol. 19 (1967)

12. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969)
13. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR - combining static verification and dynamic analysis. *J. Autom. Reason.* **60**(3), 337–363 (2018)
14. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. *Formal Asp. Comput.* **27**(3), 573–609 (2015). <http://frama-c.com>
15. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: Cohen, E., Rybalchenko, A. (eds.) *VSTTE 2013*. LNCS, vol. 8164, pp. 170–190. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54108-7_9
16. Petiot, G., Botella, B., Julliand, J., Kosmatov, N., Signoles, J.: Instrumentation of annotated C programs for test generation. In: *14th International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, pp. 105–114. IEEE (2014)
17. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your proof fails? Testing helps to find the reason. In: Aichernig, B.K.K., Furia, C.A.A. (eds.) *TAP 2016*. LNCS, vol. 9762, pp. 130–150. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41135-4_8
18. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language (2012). <http://frama-c.com/download/e-acsl/e-acsl.pdf>
19. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pp. 57–69. ACM (2016)
20. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: *Proceedings of the International Symposium on Memory Management (ISMM 2017)*, pp. 47–58. ACM (2017)



Under-Approximation Generation Driven by Relevance Predicates and Variants

J. Julliand^(✉), O. Kouchnarenko^(✉), P.-A. Masson^(✉), and G. Voiron^(✉)

FEMTO-ST, UMR 6174 CNRS, Univ. Bourgogne Franche-Comté,
16, route de Gray, 25030 Besançon Cedex, France
{jjulliand,okouchna,pamasson,gvoiron}@femto-st.fr

Abstract. In test generation, when computing a reachable concrete under-approximation of an event system's predicate abstraction, we aim at covering each reachable abstract transition with at least one reachable concrete instance. As this is in general undecidable, an algorithm must finitely instantiate the abstract transitions for it to terminate. The approach defended in this paper is to first concretely explore the abstract graph, while concretizing the abstract transitions met at most once. However, some abstract transitions would require that loops were taken previously for them to become reached. To this end, in a second phase, a test engineer guides the exploration by describing a relevance predicate able to travel such loops. We give hints on how to design and express a relevance predicate, and provide a method for automatically extracting a variant out of it. A relevance guided concretization algorithm is given, whose termination is ensured by using this variant. Experimental results are provided that show the interest of the approach.

Keywords: Predicate abstraction · Under-approximation generation
Loop variant · Relevance predicate

1 Introduction

In model-based testing [1, 2], the user wants to derive a test suite from a model, that achieves a given coverage (e.g. all states, all transitions, etc.) of it. Sometimes the infinite or very large size of the explicit state space of the model makes its coverage impossible, and an abstraction of the model can be used instead: the possibly infinitely many explicit states are grouped into finitely many abstract super-states. In predicate abstraction [3], explicit states are mapped onto abstract ones by means of a set of predicates that characterizes each abstract state. These predicates can for example automatically derive from a formalized test intention [4]. An abstract transition links two abstract states when it has at least one explicit instantiation. Such transitions are called *may* transitions [5], meaning that they may be instantiated.

The general framework of our work is to generate tests from predicate abstractions of event systems, that are a special kind of action systems. Contrarily to programs, event systems have no explicit control flow that could be

preserved in an abstraction for guaranteeing abstract paths to be explicitly instantiable as reachable and connected sequences. To this end, [6] introduces an algorithm that widens a frontier of reached states by systematically trying to prolong the existing concrete sequences with instantiating some yet unexplored abstract transition. The approach is called concrete exploration (CXP). It aims at covering each abstract transition, but only once for avoiding the concrete state space blow-up. Experiments in [6] have shown that, despite covering most of the abstract transitions, this approach fails at covering some of them whose enabling would require that previous transitions were taken repeatedly in loops.

In this paper we propose that selected loops are allowed to be traversed by designing an adequate *relevance predicate*. It is domain specific, and relies on the knowledge owned by a test engineer of the model that (s)he has written. We revisit the relevance function of Grieskamp et al. [7], that achieves a similar goal in the context of deriving a Finite State Machine from an Abstract State Machine. Our solution is to observe the coverage achieved by CXP, in order to drive the loop executions towards a *test goal*, i.e. reaching one or more concrete states in which the non-covered abstract transitions become enabled. Our relevance predicate expresses a condition over two consecutive concrete states, telling for the target concrete state if it is relevant or not to continue the exploration from it. It has to make the exploration go through cycles. To achieve termination of this process, we propose to deduce –from a relevance predicate exhibited by the test engineer– a variant that strictly decreases until it reaches a minimal value.

Summarized, our contributions are to: (1) propose a method for designing a relevance predicate, as well as a simple language for its expression, (2) automatically deduce a loop variant from a relevance predicate expressed in this language, (3) exhibit an algorithm that implements the approach by completing, with a relevance predicate as input, an existing under-approximation, (4) experimentally assess the method. The formal background required for reading the paper is given in Sect. 2. We illustrate our approach in Sect. 3 through the example of a simple coffee vending machine. Computing of a concrete under-approximation, designing a relevance predicate and deducing a loop variant from it are explained in Sect. 4. The algorithm that implements the method is given in Sect. 5. The experimental results of applying the method to five case studies are in Sect. 6. In Sect. 7 we position our approach w.r.t. related work, and we conclude the paper in Sect. 8.

2 Background

In this paper, systems are specified by event systems (ES) described in the B syntax [8, 9]¹. Notice however that our proposals and results are general enough since event system semantics is given by labelled transition systems (LTS).

¹ Our experimental models are written in B, but could alternatively be translated into a syntax with guarded commands [10], such as Abstract State Machines [11, 12].

This section first provides the syntax and the semantics of B event systems. Then we present a predicate abstraction and formalize it for event systems by means of May Transition Systems (MTS). Finally, we recall the notion of variant, usually used to prove the termination of iterative programs and systems. It will serve as the support to terminate the exploration from a relevance predicate.

2.1 Model Syntax and Semantics

Let us first introduce B event systems in Definition 1. They are composed of events specified by means of guarded actions [10]. Once the system is in a state satisfying the guard of an event, the latter is spontaneously fired.

Definition 1 (Event System). *Let EvName be a set of event names. A B event system is a tuple $\langle X, I, \text{Init}, \text{Ev} \rangle$, where X is a set of state variables, I is a state invariant, Init is an initialization action such that I holds in any initial state, and Ev is a set of event definitions, each of the form $e \stackrel{\text{def}}{=} a$ where $e \in \text{EvName}$ is the name of the event and a the action it performs. Note that every application of an event must preserve I .*

Definition 2 ((Concrete) State of an Event System). *A (concrete) state of an event system $\langle X, I, \text{Init}, \text{Ev} \rangle$ is a proposition preserving I defined as a conjunction of valuations of all state variables in X .*

The events are defined by composing the following five primitive actions: *skip*, an action with no effect, $x := E$ an action assigning the value of the arithmetic expression E to the state variable x , $P \Rightarrow a$, a guarded action requiring the event system to be in a state satisfying the predicate P before the action a can be applied, $a_1 \square a_2$, a bounded non-deterministic choice between the two actions a_1 and a_2 and finally $@z.a$ an action applying the action a which depends on the bound variable z whose value is chosen non-deterministically. The guard (noted grd) defines the firing condition of an action. They are defined on the primitive actions by: $\text{grd}(\text{skip}) \stackrel{\text{def}}{=} \text{true}$ (the *skip* action can always be applied), $\text{grd}(x := E) \stackrel{\text{def}}{=} \text{true}$ (the single assignment action can always be applied), $\text{grd}(P \Rightarrow a) \stackrel{\text{def}}{=} P \wedge \text{grd}(a)$ (as defined before, the guarded action only applies a if the system is in a state satisfying P , and the guard of a ($\text{grd}(a)$) must also be satisfied for a to be applied), $\text{grd}(a_1 \square a_2) \stackrel{\text{def}}{=} \text{grd}(a_1) \vee \text{grd}(a_2)$ (one of the actions a_1 or a_2 whose guard is satisfied is applied), $\text{grd}(@z.a) \stackrel{\text{def}}{=} \exists(z).\text{grd}(a)$ (there must exist some bound variable z satisfying the guard of a (which depends on z) for a to be applied).

See Fig. 1 in Sect. 3 for an example of a B event system. Following [13], we define the semantics of event systems by means of a labelled transition system (LTS) whose concrete states are defined in Definition 2. Let $e \stackrel{\text{def}}{=} a$ be an event. It has a *weakest precondition* [14] w.r.t. a set Q' of target states, denoted $wp(a, Q')$. It is the largest set of states from which applying a always leads to a state in Q' . An event also defines a relation between the values of the state variables

before (X) and after (X') the application of the event. It is expressed by the before-after predicate of the event $e \stackrel{\text{def}}{=} a$, denoted $\text{prd}_X(a)$.

Let us now formally define wp and prd_X following [8]. We associate the sets of states Q and Q' with predicates: a set of states Q defines a predicate Q that holds in any state of Q , but does not hold in any state not in Q .

We define the wp w.r.t. the five primitive actions by: $\text{wp}(\text{skip}, Q') \stackrel{\text{def}}{=} Q'$, $\text{wp}(x := E, Q') \stackrel{\text{def}}{=} Q'[E/x]$ that is the usual substitution of x by E , $\text{wp}(P \Rightarrow a, Q') \stackrel{\text{def}}{=} P \Rightarrow \text{wp}(a, Q')$, $\text{wp}(a_1 \parallel a_2, Q') \stackrel{\text{def}}{=} \text{wp}(a_1, Q') \vee \text{wp}(a_2, Q')$, $\text{wp}(@z.a, Q') \stackrel{\text{def}}{=} \forall z. \text{wp}(a, Q')$, where z is not a free variable in Q' .

Then prd_X is defined w.r.t. wp by $\text{prd}_X(a) \stackrel{\text{def}}{=} \neg \text{wp}(a, x'_1 \neq x_1 \vee \dots \vee x'_n \neq x_n)$. It is a predicate over the state variables $X = \{x_1, \dots, x_n\}$ in the source state before a , and the target state variables $X' = \{x'_1, \dots, x'_n\}$ after a .

2.2 Predicate Abstraction

Predicate abstraction [3] is a special instance of the framework of abstract interpretation [15] that maps the potentially infinite state space C of an LTS onto the finite state space A of an abstract transition system *via* a set $\mathcal{P} \stackrel{\text{def}}{=} \{p_1, p_2, \dots, p_n\}$ of n predicates over the state variables. The set of abstract states A contains 2^n states. Each state is a tuple $q \stackrel{\text{def}}{=} (q_1, q_2, \dots, q_n)$ with q_i being equal either to p_i or to $\neg p_i$, and q is also considered as the predicate $\bigwedge_{i=1}^n q_i$. We define a total abstraction function $\alpha : C \rightarrow A$ such that $\alpha(c)$ is an abstract state q where c satisfies q_i for all $i \in 1 \dots n$. By a misuse of language, we say that c is in q , or that c is a concrete state of q .

Let us now define abstract *may* transitions. Consider two abstract states q and q' , and an event e . There exists a *may* transition $q \xrightarrow{e} q'$, if and only if there exists at least one concrete transition $c \xrightarrow{e} c'$ such that $\alpha(c) = q$ and $\alpha(c') = q'$. The *may* transition is *reachable* if and only if there is at least one such concrete transition $c \xrightarrow{e} c'$ whose source state c is reachable from a concrete initial state.

We check predicate satisfiability thanks to SMT solvers. For a predicate P , we define the solver invocation $\text{SAT}_c(P)$ as returning either a model of P , or **unsat** if P is unsatisfiable, or **unknown** if the solver failed to determine the satisfiability of P . We also define $\text{SAT}(P)$ as the predicate that is true iff $\text{SAT}_c(P)$ returns a model. Let $e \stackrel{\text{def}}{=} a$ be an event definition, $q \xrightarrow{e} q'$ is a *may* transition iff $\text{SAT}(\neg \text{wp}(a, \neg q') \wedge q)$. We compute a concrete witness $c \xrightarrow{e} c'$ by using the before-after predicate: $(c, c') := \text{SAT}_c(\text{prd}_X(a) \wedge q'[X'/X] \wedge q)$ where $q'[X'/X]$ is the q' predicate in which each state variable x_i is substituted by x'_i .

2.3 May Transition Systems

Definition 3 introduces *may* transition systems (MTS) having abstract states, and abstract *may* transitions. Definition 4 associates an abstraction defined by an MTS with an ES. The reader will be provided with an example in Sect. 3.

Definition 3 (May Transition System). Let EvName be a finite set of event names, and $\mathcal{P} \stackrel{\text{def}}{=} \{p_1, p_2, \dots, p_n\}$ be a set of predicates. Let A be a set of 2^n abstract states defined from \mathcal{P} . A tuple $\langle Q, Q_0, \Delta \rangle$ is an MTS if it satisfies the following conditions: $Q (\subseteq A)$ is a finite set of states, $Q_0 (\subseteq Q)$ is a set of abstract initial states, and $\Delta (\subseteq Q \times \text{EvName} \times Q)$ is a may transition relation with labels in EvName .

Definition 4 (MTS from an ES and abstraction predicates). Let $ES \stackrel{\text{def}}{=} \langle X, I, \text{Init}, \text{Ev} \rangle$ be an ES, and $\mathcal{P} \stackrel{\text{def}}{=} \{p_1, p_2, \dots, p_n\}$ be a set of n predicates over X defining a set of 2^n abstract states. A tuple $\langle Q, Q_0, \Delta \rangle$ is an MTS from ES and \mathcal{P} if it satisfies the following conditions:

- $Q \stackrel{\text{def}}{=} \{q \in A \mid \exists (q', e). (q \xrightarrow{e} q' \in \Delta \vee q' \xrightarrow{e} q \in \Delta)\}$,
- $Q_0 \stackrel{\text{def}}{=} \{q \mid q \in A \wedge (\text{SAT}(\text{prd}_X(\text{Init}) \wedge q[X'/X]))[X/X']\}$,
- $\Delta \stackrel{\text{def}}{=} \{q \xrightarrow{e} q' \mid q \in A \wedge q' \in A \wedge e \stackrel{\text{def}}{=} a \in \text{Ev} \wedge \text{SAT}(\neg \text{wp}(a, \neg q') \wedge q)\}$.

Reachable MTS. The reachable MTS from ES and \mathcal{P} is the MTS that contains all the reachable *may* transitions, and only those ones. The notion of reachable MTS is the same as that of *true* FSM in the context of abstract state machines [7]. As such, and as proved in [7], computing the reachable MTS from an ES and a set \mathcal{P} of abstraction predicates is in general an undecidable problem.

2.4 Variant of Iterative Systems

The notion of variant is usually used to prove the termination of iterative programs and systems. In this paper variants are associated with relevance predicates, which can be seen as a kind of test goal.

For example in the deductive verification tools Frama-C [16] and KeY [17], in order to prove the termination of program loops, the engineer must provide for each loop a variant annotation that defines a natural integer arithmetic expression. This expression must be non-negative before each iteration of the loop, and must strictly decrease at each iteration. For example, in a binary search algorithm in the array interval $L \dots R$, the variant is the expression $R - L$ that defines the length of the search interval. It has to strictly decrease at each execution of the body of the following loop: **while** $L < R$ **do** $M := (L + R + 1)/2$; **if** $T[M] \leq X$ **then** $L := M$ **else** $R := M - 1$ **fi od**. So, the algorithm terminates when the interval is such that $L = R$.

The contributions of this paper provide means, for a test engineer, to cover by tests the transitions whose enabling require that a loop of events have previously been executed. We mainly propose that the test engineer provides a test goal described by means of a relevance predicate, from which we deduce a variant such that any event satisfying the relevance predicate strictly decreases this variant.

3 Running Example

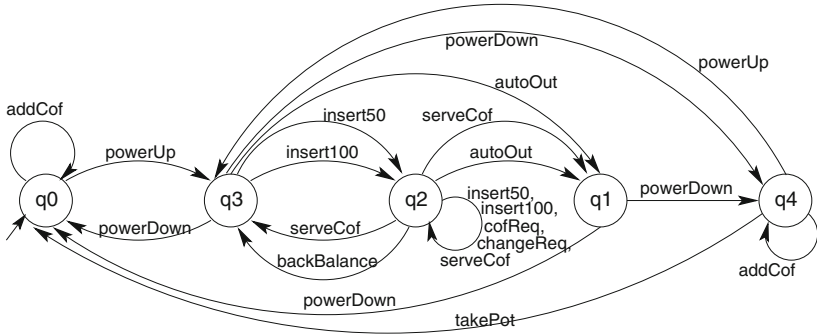
Our illustrative example is a simplified coffee vending machine (see the ES in Fig. 1). It has a *Balance*, which can be augmented by putting coins of value either 50 or 100 (events `insert50` and `insert100` in Fig. 1). *Balance* may not exceed an arbitrary fixed constant named *MAX_Bal*. There are arbitrary constants for the maximal number of coffees stored in the machine (*MAX_Cof*), and the maximal value (*MAX_Pot+50*) of the *Pot* (the money kept by the machine). Notice that *Balance* and *Pot* are multiples of 50 (specified in the invariant). The machine has a *Status* which indicates if it is switched on (1) or off (0), or out of order (2). When switched on, the machine can serve coffees, after a request by the user (event `cofReq` that corresponds to pressing the “request coffee” button), at the price of 50 each (event `serveCof`): this price is retrieved from the *Balance* and sent to the *Pot*. The number of available coffees is modelled by the *CofLeft* variable. The user can ask for its change (event `changeReq` corresponds to pressing the “give change” button). The events `changeReq` and `cofReq` are mutually exclusive. The user can then get its unused balance back (event `backBalance`). When switched off, the machine can be refilled with coffee (event `addCof`), and its *Pot* retrieved (event `takePot`). The events `powerUp` and `powerDown` are for switching the machine respectively on or off. Finally, a special event (`autoOut`) sets the machine out of order: it models the unexpected occurrence of a failure while the machine is in use. It also occurs when either there is no more coffee, or the *Pot* is full (see `serveCof`). Figure 2 represents the MTS of the ES of Fig. 1 for the three following predicates: $p_0 \stackrel{\text{def}}{=} \text{Status} = 0 \wedge \text{Pot} \geq \text{MAX_Pot} - 50$, $p_1 \stackrel{\text{def}}{=} \text{Status} = 1$ and $p_2 \stackrel{\text{def}}{=} (\text{Status} = 1 \wedge \text{AskChange} = 0 \wedge \text{AskCof} = 0 \wedge \text{Balance} = 0) \vee \text{Status} = 2$ that are respectively the guards of the events `takePot`, `autoOut` and `powerDown`.

The test generation method presented in [6] generates a concrete LTS that is an under-approximation of the semantics of the specification in Fig. 1. The method concretizes all the *may* transitions but some instances are not connected to the initial state of the under-approximation, due to the choice of traversing each *may* transition only once. Sometimes previous transitions should have been taken in loop for reaching a connected concrete state in which the targeted transition is enabled. This is for example the case with the transition $q_2 \xrightarrow{\text{serveCof}} q_1$. It serves the machine’s last coffee in stock. Its enabling requires to previously execute a loop which serves all coffees until emptying the stock. The idea presented in this paper for covering such transitions is to trigger a second step, for completing a posteriori the LTS computed at the first step by the algorithm of [6]. This second step is allowed to loop through some cycles of the abstract graph, by generating new concrete states from the existing ones as long as they are *relevant*. For guaranteeing this looping to terminate, we propose a state to be relevant as long as it decreases a variant of the loop.

Relevance Predicate Example: In the coffee machine, transition $q_2 \xrightarrow{\text{serveCof}} q_1$ can only be triggered once the coffee stock is empty ($\text{CofLeft} = 0$). This requires having previously looped between states q_3 and q_2 through the events `insert50`,

X	$\stackrel{\text{def}}{=} \{Balance, Pot, Status, CofLeft, AskCof, AskChange\}$
I	$\stackrel{\text{def}}{=} Pot \in 0..MAX_Pot + 50 \wedge Balance \in 0..MAX_Bal \wedge$ $CofLeft \in 0..MAX_Cof \wedge Pot \bmod 50 = 0 \wedge Balance \bmod 50 = 0 \wedge$ $Status \in 0..2 \wedge AskCof \in 0..1 \wedge AskChange \in 0..1 \wedge$ $AskChange = 1 \Rightarrow (Balance > 0 \wedge AskCof = 0) \wedge$ $AskCof = 1 \Rightarrow (Balance \geq 50 \wedge AskChange = 0) \wedge$ $Balance = 0 \Rightarrow (AskCof = 0 \wedge AskChange = 0)$
Init	$\stackrel{\text{def}}{=} Balance := 0 \parallel Status := 0 \parallel Pot := 0 \parallel$ $CofLeft := 10 \parallel AskCof := 0 \parallel AskChange := 0$
insert50	$\stackrel{\text{def}}{=} Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge$ $Balance + 50 \leq MAX_Bal \Rightarrow Balance := Balance + 50$
insert100	$\stackrel{\text{def}}{=} Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge$ $Balance + 100 \leq MAX_Bal \Rightarrow Balance := Balance + 100$
powerUp	$\stackrel{\text{def}}{=} Status = 0 \wedge CofLeft > 0 \wedge Pot \leq MAX_Pot \Rightarrow$ $Status := 1 \parallel Balance := 0 \parallel AskCof := 0 \parallel AskChange := 0$
powerDown	$\stackrel{\text{def}}{=} (Status = 1 \wedge AskChange = 0 \wedge AskCof = 0 \wedge Balance = 0) \vee$ $Status = 2 \Rightarrow Status := 0$
autoOut	$\stackrel{\text{def}}{=} Status = 1 \Rightarrow Status := 2$
takePot	$\stackrel{\text{def}}{=} Status = 0 \wedge Pot \geq MAX_Pot - 50 \Rightarrow Pot := 0$
cofReq	$\stackrel{\text{def}}{=} Status = 1 \wedge Balance \geq 50 \wedge AskCof = 0 \wedge$ $AskChange = 0 \Rightarrow AskCof := 1$
changeReq	$\stackrel{\text{def}}{=} Status = 1 \wedge Balance > 0 \wedge AskCof = 0 \wedge$ $AskChange = 0 \Rightarrow AskChange := 1$
addCof	$\stackrel{\text{def}}{=} \exists x. (x \in 1..MAX_Cof \wedge CofLeft + x \leq MAX_Cof$ $\wedge Status = 0 \Rightarrow CofLeft := CofLeft + x)$
serveCof	$\stackrel{\text{def}}{=} Status = 1 \wedge Balance \geq 50 \wedge AskCof = 1 \wedge CofLeft > 0 \wedge$ $Pot \leq MAX_Pot \Rightarrow$ $AskCof := 0 \parallel Balance := Balance - 50$ $\parallel CofLeft := CofLeft - 1 \parallel Pot := Pot + 50$ $\parallel (Pot \geq MAX_Pot \vee CofLeft = 1 \Rightarrow Status := 2$ $\parallel Pot + 50 \leq MAX_Pot \wedge CofLeft \neq 1 \Rightarrow skip)$ $\parallel (Balance > 50 \Rightarrow AskChange := 1 \parallel Balance = 50 \Rightarrow skip)$
backBalance	$\stackrel{\text{def}}{=} Status = 1 \wedge Balance > 0 \wedge AskChange = 1 \Rightarrow$ $Balance := 0 \parallel AskChange := 0$

Fig. 1. ES specification of a coffee machine

Fig. 2. MTS of the coffee machine w.r.t. predicates p_0 , p_1 and p_2 .

insert100, cofReq and serveCof. The progress conditions along that loop are that either the variable $Balance$ increases, or the variable $CofLeft$ decreases, or the variable $AskCof$ passes from zero to one. In terms of the before and after values of the variables, this is expressed as the following *relevance predicate* (RP): $Balance' > Balance \vee CofLeft' < CofLeft \vee (AskCof = 0 \wedge AskCof' = 1)$.

4 Test Generation Based on Relevance Predicates

We first define the concept of Approximated Transition System (ATS) which brings together an MTS and one of its under-approximations. Section 4.1 gives an overview of the process in two phases of under-approximation that we propose for computing an ATS. Then Sect. 4.2 explains how to design an RP on which the second phase (detailed in Sect. 5) is based. Finally Sect. 4.3 gives the relationship between an RP and a variant that guarantees the termination of the method.

We call Approximated Transition System (ATS, see Definition 5) the reunion of an abstraction with one of its under-approximations that is a concrete part of the LTS, which is the semantics of the event system from which the MTS is deduced.

Definition 5 (Approximated Transition System). *Let $\langle Q, Q_0, \Delta \rangle$ be an MTS. A tuple $\langle Q, Q_0, \Delta, C, C_0, \Delta^c, \alpha \rangle$ is an ATS whose $\langle C, C_0, \Delta^c, \alpha \rangle$ is a concretization of the MTS where C, C_0 are sets of respectively concrete states and concrete initial states, $\Delta^c (\subseteq C \times \text{EvName} \times C)$ is a concrete labelled transition relation, and α is a total abstraction function from C to Q .*

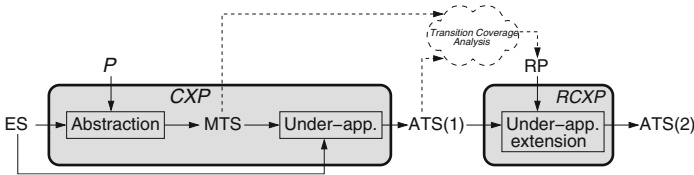


Fig. 3. ATS computation process

4.1 Process Overview

We sketch our process in Fig. 3. We propose to compute an ATS in two steps. We get a first version (ATS (1) in Fig. 3) by an approach [6] called CXP for *concrete exploration*, that traverses and concretizes each abstract transition only once. Then, thanks to a relevance predicate RP provided by the test engineer, selected loops of abstract transitions are additionally traversed and concretized by RCXP, in order to connect to new concrete transitions. As a result, ATS (1) is extended to ATS (2).

The CXP approach is fully described in [6]. The two operations **Abstraction** and **Under-app.** are summarized as follows.

1. **Abstraction.** The test engineer designs a set of abstraction predicates P related to the behaviour of the system ES (s)he wishes to observe. It is proposed in [4] that these predicates are extracted from a *test purpose*, which is a test intention formalized by a pattern *a la* Dwyer et al. [18]. Using algorithm in [6] provides the test engineer with an MTS that over-approximates the reachable MTS.

2. **Under-app.** The under-approximation is computed by concretizing on the fly each *may* transition once, as it is discovered. The principle is to compute an instance that prolongs, whenever possible, some existing sequence connected to a concrete initial state. For CXP’s efficiency, each abstract transition is concretized only once and thus cannot be applied repeatedly. The ATS obtained is called ATS (1) in Fig. 3.

In general, not all the instances of abstract transitions built by ATS (1) are reached, even though they are possibly reachable. It is always the case in particular, when repeating some transitions in a cycle would have been necessary for enabling another transition. In case ATS (1) fails at building a connected instance of a *may* transition, the transition is concretized anyway but as a “hanging” instance, i.e. disconnected from the previously reached part of the under-approximation.

The second step, called RCXP for *relevant concrete exploration*, requires human interaction. The test engineer analyses, for each abstract transition of the MTS unreached in ATS (1), if (s)he thinks it could have been reached. If so (s)he identifies which transitions taken in loop it would require for reaching it. For that (s)he can observe in the MTS which cycles lead to enabling the target transitions. As this looping may not terminate, (s)he has to provide an RP telling whether or not it is relevant to pursue in the loop. The operation **Under-app. extension** consists of adding to ATS (1) the concrete transitions obtained by this RP guided exploration. It results in ATS (2), in which the transitions originally targeted by RP are possibly reached. An algorithmic implementation of RCXP is given in Sect. 5. Let us for now illustrate the process and RP design through the coffee machine example.

4.2 Design of a Relevance Predicate and Illustration of the Method

For illustrating the application of the method, we consider the coffee machine example, and a requirement stating that it must not break down after being powered off, so that collecting the pot remains possible. Using the temporal logic patterns of Dwyer et al. [18], this can be expressed as: *Never autoOut Between powerDown and takePot*. As proposed in [4], the tester can use the guards of the events invoked in this test purpose as abstraction predicates for computing the MTS. Here, this gives the predicates p_0 , p_1 and p_2 defined in Sect. 3, from which the abstraction of Fig. 2 has been computed. The tester executes CXP [6] with these predicates as input, and observes the resulting MTS coverage. In the case of the coffee machine, the two transitions $q_2 \xrightarrow{\text{serveCof}} q_1$ and $q_1 \xrightarrow{\text{powerDown}} q_4$ and the state q_4 are not covered. Having designed the model, the tester is able to understand that the transition $q_2 \xrightarrow{\text{serveCof}} q_1$ serves the last coffee in stock, so that its coverage would have required that previously all the coffees were served. By looking at the MTS, it is easy to see that covering this transition would require looping between states q_3 and q_2 . (S)He identifies as illustrated in Sect. 3 the set of events to loop through in order to reach his (her)

goal. In our case, the goal is serving the last coffee and the set of events to loop through is `insert50`, `insert100`, `cofReq` and `serveCof`. After this step (s)he has to express by means of a before-after predicate how it is relevant that the variables assigned in these events evolve. The RP is then the disjunction of these before-after predicates. For the coffee machine example, this gives the RP described in Sect. 3, paragraph *Relevance Predicate Example*. The variables have to decrease a variant for the looping to terminate. Let us now explain in Sect. 4.3 how to deduce this variant from the RP.

4.3 Variant Deduced from a Relevance Predicate

Given an RP, this section shows how to derive a variant that guarantees the termination of the computation of relevant concrete states. We assume that the test engineer uses a simple language defined as follows to express the RP, where x , x' denote the state variable x respectively before and after an event application:

$$\begin{aligned} rel_p &::= rp_1 \vee \dots \vee rp_n \\ rp &::= ap \mid cp \\ ap &::= x' < x \mid x' > x \mid x = v \wedge x' = v' \\ cp &::= b_1 \Rightarrow ap_1 \wedge \dots \wedge b_m \Rightarrow ap_m \end{aligned}$$

An RP is a disjunction of before-after predicates, each of which being either an atomic predicate ap or a conditional predicate cp . We consider these predicates to only use the following three variable types: intervals of integer $MIN_x \dots MAX_x$, booleans, and finite enumerated sets of labels. We assume that the atomic predicates ap over a state variable x expresses that an integer variable either strictly decreases ($x' < x$) or increases ($x' > x$), or the value of an enumerated variable (including the boolean type) passes from v to v' . We assume that $v \neq v'$. We also consider that in the conditional predicate pattern cp , each b_i is a boolean condition on the source state. For any concrete state c , we finally assume that there exists one and only one i such that the predicate b_i is satisfied in the concrete state c , denoted as: $c \models b_i$.

Let c , c' denote respectively the state c before and after an event execution. The first following three rules associate an initial variant, denoted $V_{init}(rp, c)$, with the concrete state c depending on the RP rp . The next five rules associate the next value of the variant, denoted $V(rp, c')$, with an RP rp in a target state c' reached from a source state c . Notice that in Rules 4, 5, 6 and 8, $V(rp, c) \stackrel{\text{def}}{=} V_{init}(rp, c)$ when c is an initial state:

1. $V_{init}(ap, c) \stackrel{\text{def}}{=} Card(Type(x))$,
2. $V_{init}(b_1 \Rightarrow ap_1 \wedge \dots \wedge b_m \Rightarrow ap_m, c) \stackrel{\text{def}}{=} \begin{aligned} &\text{if } c \models b_1 \text{ then } V_{init}(ap_1, c) \\ &\text{else if } \dots \text{ else if } c \models b_m \text{ then } V_{init}(ap_m, c), \end{aligned}$
3. $V_{init}(rp_1 \vee \dots \vee rp_n, c) \stackrel{\text{def}}{=} V_{init}(rp_1, c) + \dots + V_{init}(rp_n, c)$,

4. $V(x' < x, c') \stackrel{\text{def}}{=} V(x' < x, c) - (x - x')$,
5. $V(x' > x, c') \stackrel{\text{def}}{=} V(x' > x, c) - (x' - x)$,
6. $V(x = v \wedge x' = v', c') \stackrel{\text{def}}{=} V(x = v \wedge x' = v', c) - 1$,
7. $V(b_1 \Rightarrow ap_1 \wedge \dots \wedge b_m \Rightarrow ap_m, c') \stackrel{\text{def}}{=} \begin{array}{l} \text{if } c' \models b_1 \text{ then } V(ap_1, c') \\ \text{else if } \dots \text{ else if } c' \models b_m \text{ then } V(ap_m, c'). \end{array}$
8. $V(rp_1 \vee \dots \vee rp_n, c') \stackrel{\text{def}}{=} \sum_{\{i \mid i \in 1..n \wedge (c, c') \models \neg rp_i\}} V(rp_i, c) + \sum_{\{i \mid i \in 1..n \wedge (c, c') \models rp_i\}} V(rp_i, c')$.

Rule 1 allows applying as many operations as the size of the enumerated sets or of the integer intervals ($MAX_x - MIN_x + 1$). Rule 2 applies the previous rule according to the condition that holds in the initial state. Rule 3 defines the initial variant value as the sum of the variant values of each of the disjunction members. Rules 4 and 5 define that the next variant value decreases of the difference between the two successive values of x . Rule 6 defines that the next variant value for a modification of an enumerated variable decreases of one. Rule 7 defines that the next variant value for a conditional predicate decreases as much as the atomic predicate that is satisfied in the state c' . Last, with Rule 8, the variant in the target state of a relevant predicate is unchanged for the disjunction member that are not satisfied, and varies according to the rules that apply for the ones that are satisfied.

Property 1. If an RP is satisfied, then the associated variant decreases.

Proof. For a transition $c \xrightarrow{e} c'$ in an ATS and for an RP rp , the variant decreases if $V(rp, c') < V(rp, c)$. We prove that for the three predicate cases: ap , cp and rel_p . For an atomic predicate, the variant decreases respectively of $|x - x'|$ and one respectively according to Rules 4, 5 and 6. For a conditional predicate, the variant decreases as much as the atomic predicate that is true according to Rule 7. For a disjunctive predicate rel_p the variant decreases, according to Rule 8. Indeed, (1) the variant is not modified for the disjunction members that are not satisfied in the consecutive states c, c' , and (2) the variant decreases for the disjunction members that are satisfied in the states c, c' because they are either ap or cp , for which the decrease has already been shown.

5 RCXP Algorithm

In this section we present the main contribution of this paper. The RCXP (for relevant concrete exploration) algorithm implements the second step of the process presented in Sect. 4.1.

5.1 Under-Approximation Extension Using Relevance Predicates

To extend the ATS computed in the first place by CXP, we propose an algorithm called RCXP which aims at covering the non-covered transitions. It is driven by

an RP designed as explained in Sect. 4.2. RCXP is designed from the concepts of relevant state and goal state. A goal state is a state in which an non-covered abstract transition is triggerable. Informally a state is relevant when it gets closer to a goal state. Formally we say that a target state c' by a transition t is relevant w.r.t. the source state c of t when (c, c') satisfies an RP (see Sect. 4.3).

Algorithm RCXP. Concretization Algorithm using Relevance Predicate

Inputs : $\langle Q, Q_0, \Delta, C, C_0, \alpha, \Delta^c \rangle$: an ATS; A : the set of all abstract states
relevance_pred_X: a relevance predicate
Outputs : $\langle Q, Q_0, \Delta, C, C_0, \alpha, \Delta^c \rangle$: the ATS enriched
Variables : *RCS* (resp. *PRCS*): the set of relevant concrete states to process
 (resp. processed)

```

1  RCS := {c | c ∈ C ∧ Vinit(relevance_predX, c) ≥ 0}; PRCS := ∅;
2  /* the reachable concrete states computed by CXP */
3  while RCS ≠ ∅ do
4    choose c ∈ RCS;
5    RCS := RCS − {c}; PRCS := PRCS ∪ {c};
6    q := α(c);
7    foreach q' ∈ A do
8      foreach e def a ∈ Ev do
9        if q  $\xrightarrow{e}$  q' ∈ Δ then
10       (c, c') := SATc(c ∧ prdX(a) ∧ q'[X'/X] ∧ relevance_predX);
11       if (c, c') ∉ {unknown, unsat} then
12         if V(relevance_predX, c') ≥ 0 ∧ c' ∉ PRCS then
13           | RCS := RCS ∪ {c'};
14         end
15         α(c') := q'; C := C ∪ {c'}; Δc := Δc ∪ {c  $\xrightarrow{e}$  c'};
16       else
17         /* a goal state is reached, we try to apply the transition from it */
18         (c, c') := SATc(c ∧ prdX(a) ∧ q'[X'/X]);
19         if (c, c') ∉ {unknown, unsat} then
20           | α(c') := q'; C := C ∪ {c'}; Δc := Δc ∪ {c  $\xrightarrow{e}$  c'};
21         end
22       end
23     end
24   end
25 end
26 end

```

Algorithm RCXP launches its execution from each state c built by CXP that is evaluated as relevant, assuming that the variant expression $V_{init}(rp, c)$ is non-negative (line 1). The algorithm tries to reach a new relevant target concrete state (line 10) for each target abstract state (line 7) and for each event (line 8) such that the corresponding transition is *may* (line 9). If such a state c' is found (lines 12–15), it is added (line 15) to the under-approximation. Additionally in case c' is new and has a non-negative variant value (line 12), it is added to the set of relevant states to be processed (line 13). When no more relevant state is found (*else* statement in line 16), a goal state has been reached. The algorithm tries to finally apply the event e from it (lines 18–21) because it might correspond to a non-covered transition. The algorithm's result is the input ATS enriched with new concrete states and transitions.

5.2 Soundness, Complexity and Termination

This section discusses the soundness, complexity and gives the termination proof for the RCXP algorithm.

Soundness. RCXP computes an under-approximation of the reachable MTS. Indeed, as our method only keeps transition instances that are connected to an initial concrete state, all the *may* transitions that we cover are reachable, and thus are part of the reachable MTS.

Complexity. Let us denote by C_{in} the set C of concrete states of the input ATS. For each state $c \in C_{in}$, RCXP computes a concrete instance of each *may* transition whose source state is $\alpha(c)$ (there are at most $|Ev| \times |A|$ of them). From every state reached from these concrete transitions (at most $|C_{in}| \times |Ev| \times |A|$ states), RCXP launches a search for relevant successors. The number of computed successors is bounded by the maximum number of steps allowed by the variant, which equals $\max_{c \in C_{in}} (V_{init}(rp, c))$. Thus, our algorithm runs in $O(|C_{in}| \times |Ev| \times |A| \times \max_{c \in C_{in}} (V_{init}(rp, c)))$. Notice that $|C_{in}|$, $|Ev|$ and $|A|$ depend on the size of the abstract graph and that in practice the number of “relevant events” is likely to be lower than $|Ev|$. This means that for an abstract graph of reasonable size, the complexity is dominated by the number of steps of the variant.

Termination. Our algorithm computes new concrete states only from concrete states for which the variant on the one hand is non-negative, and on the other hand strictly decreases (see Property 1). Thus our algorithm terminates. In addition, as the number of relevant states may explode, RCXP has been implemented with a timeout option modifiable by the tester.

6 Experiments

The tool used to generate the results presented in this section, as well as the complete set of examples, along with their corresponding sets of abstraction and relevance predicates, can be downloaded, compiled and used by following the instructions at <https://github.com/stratosphr/stratestx/wiki>.

6.1 Experimental Results

We have experimented with five different case studies: a multiple battery-powered electrical system (EL [19]), the coffee machine CM presented in this paper, two explorations of an automatic subway line (L14, as yet unpublished), an elevator (ELV, as yet unpublished) and a subpart of the GSM 11.11 standard (GSM [20]). The subway modelled by L14 in our experimentation has three stations and three trains that circulate in ring around them. The test goal is to observe half a revolution of a train around the ring. Two different relevance

predicates have been experimented with: in L14-1 the half-lap can be that of any train, whereas it is for a fixed train in L14-2. The GSM system considered corresponds to the exploration and reading of files on a SIM (*Subscriber Identity Module*) card with different access rights. Some files can only be read when a correct PIN (*Personal Identification Number*) is entered by the user. After three unsuccessful attempts with the wrong PIN, the card is locked and can only be unlocked if the user enters the correct PUK (*PIN Unlock Key*). After ten unsuccessful attempts with the wrong PUK, the protected files can never be read again.

The results are given in Table 1. Columns #Ev, #AP, #AS_{rchbl} and #AT_{rchbl} give respectively per model the numbers of: events, abstraction predicates, reachable abstract states and reachable abstract transitions in the MTS. Then the results per model are spread over three lines for comparing: the CXP approach (1st line), the RCXP approach (2nd line) and a full exploration of the reachable concrete space (FULL, on 3rd line). Notice that we have chosen the problem sizes in this table for making the full exploration possible.

Table 1 gives the number of: abstract states and transitions reached (#AS_{rchd} and #AT_{rchd}), target abstract states and transitions of RCXP (#AS^{rel} and #AT^{rel}) and concrete states and transitions either built (#CS and #CT) or reached from an initial state (#CS_{rchd} and #CT_{rchd}). The other columns indicate the percentage of abstract states and transitions reached (%AS = $\frac{\#AS_{rchd}}{\#AS}$ and %AT = $\frac{\#AT_{rchd}}{\#AT}$) and target abstract states and transitions of RCXP reached (%AS^{rel} = $\frac{\#AS_{rchd}^{rel}}{\#AS_{rchd}^{rel}}$ and %AT^{rel} = $\frac{\#AT_{rchd}^{rel}}{\#AT_{rchd}^{rel}}$ where #AS^{rel}_{rchd} and #AT^{rel}_{rchd} are respectively the number of target abstract states and transitions of RCXP reached). The Time column gives the computation times in *hours*, *minutes* and *seconds*.

6.2 Results Analysis

Table 1 shows that RCXP succeeds at reaching all the RP targeted transitions in all of the five case studies (see that all the percentages equal 100 in the %AT^{rel}

Table 1. ATS computation results

Sys.	#Ev	#AP	#AS _{rchbl}	#AT _{rchbl}	Alg.	#AS _{rchd}	%AS	#AS ^{rel}	%AS ^{rel}	#AT _{rchd}	%AT	#AT ^{rel}	%AT ^{rel}	#CS	#CS _{rchd}	#CT	#CT _{rchd}	Time
EL	4	2	4	11	CXP	2	50	-	-	6	54.55	-	-	28	6	17	6	00:00:01
					RCXP	4	100	2	100	11	100	2	100	58	42	53	45	00:00:05
					FULL	4	100	-	-	11	100	-	-	896	896	9856	9856	00:02:10
CM	11	3	5	21	CXP	4	80	-	-	12	57.14	-	-	46	11	33	12	00:00:01
					RCXP	5	100	1	100	19	90.48	1	100	149	117	179	158	00:00:04
					FULL	5	100	-	-	21	100	-	-	1742	1742	4503	4503	00:00:49
L14-1	15	3	4	49	CXP	2	50	-	-	4	8.16	-	-	99	5	57	4	00:00:04
					RCXP	4	100	2	100	49	100	45	100	2897	2840	7533	7498	00:50:08
					FULL	4	100	-	-	49	100	-	-	2982	2982	7950	7950	00:14:39
L14-2	15	3	4	49	CXP	2	50	-	-	4	8.16	-	-	103	5	57	4	00:00:05
					RCXP	4	100	2	100	11	22.45	5	100	136	40	93	41	00:00:16
					FULL	4	100	-	-	49	100	-	-	2982	2982	7950	7950	00:13:47
ELV	11	3	4	27	CXP	3	75	-	-	9	33.33	-	-	49	10	36	9	00:00:02
					RCXP	4	100	1	100	23	85.19	3	100	255	240	401	390	00:01:38
					FULL	4	100	-	-	27	100	-	-	1656	1656	6556	6556	00:05:27
GSM	5	2	4	25	CXP	1	25	-	-	5	20	-	-	45	5	30	5	00:00:01
					RCXP	4	100	3	100	16	64	8	100	155	115	208	183	00:00:16
					FULL	4	100	-	-	25	100	-	-	> 100000	> 100000	> 1000000	> 1000000	> 48:00:00

column). Moreover RCXP even succeeds in two out of the five case studies at reaching all the reachable abstract transitions (see the three percentages that equal 100 in the %AT column). RCXP generates far less concrete transitions than FULL. The most spectacular example is EL where RCXP only builds 53 concrete transitions among the 9836 built by FULL. The ratio between RCXP and FULL depends on RP. In EL, only one action is allowed by RP so that the number of concrete transitions (#CT) explored by RCXP is very small w.r.t that explored by FULL. By contrast in L14-1, all actions are allowed by RP, which makes #CT for RCXP and FULL very close. In L14-2, as RP restricts the actions allowed to that of a single chosen train, RCXP reduces #CT in large proportion: 93 out of the 7950 of FULL. But only 22.45% coverage of the abstract transitions is achieved, due to the other trains' actions being unexplored. The ELV case is similar, with an RP allowing only the actions on the inside lift buttons. RCXP builds 401 concrete transitions out of the 6556 of FULL, but covers 85.19% of the abstract transitions. Note that the ELV case has necessitated several RP design attempts before covering 100% of the targeted transitions. As many smartcard like systems, the GSM allows everything to happen but returns error status words in case of unauthorized events occurring (this is called defensive programming). Therefore, its events are in practice very weakly guarded. For this reason, and because the GSM system has a lot of state variables, the FULL exploration is not feasible in reasonable time due to the huge state space (more than 100,000 states and more than a million transitions). However, the height targeted transitions (leading to states where the SIM card was definitely locked) were all successfully covered in less than 20 seconds by RCXP by only instanciating 155 states and 208 transitions. The design of the relevance predicate was also easy since it only had to decrease the number of attempts remaining for the PIN and the PUK. This shows that the method can be applied to systems of industrial size and that designing relevance predicates for such systems is not necessarily harder.

RCXP succeeds at reaching as many targeted abstract transitions as FULL with, except for L14-1, a much smaller number of concrete transitions generated, which results in smaller RCXP generation times w.r.t. FULL. Table 1 shows that in four cases (EL, CM, L14-2, GSM), the time taken by RCXP is much closer to CXP than it is to FULL. This is less spectacular with ELV, but RCXP remains faster than FULL by roughly 70%. L14-1 is the exception where RCXP lasts four times as long as FULL. This is an extreme case because in that experiment any action on any train is considered as relevant if it helps moving a train in a privileged direction. Here computing the relevant states amounts to enumerate about half of them, which is more costly with RCXP than a full enumeration with FULL, due to the RP evaluation at each step. The examples EL, CM, L14-2 and GSM show that the smallest the subset of events allowed to occur by RP is, the more the RCXP exploration is efficient.

We have measured the concrete graph's growth w.r.t. to the problem size of our case studies. For the two of them (EL and CM) for which the FULL exploration time took less than two hours despite its growth, we have drawn

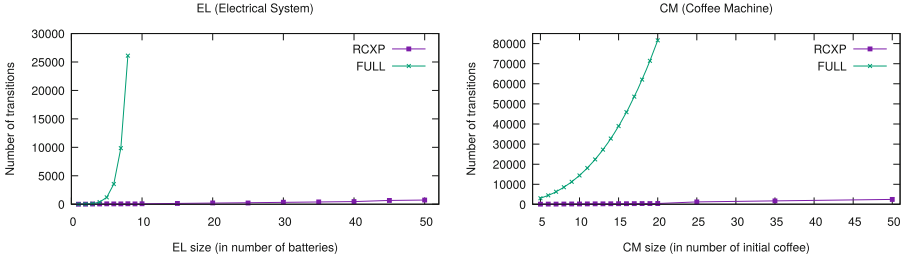


Fig. 4. #CT growth for FULL and RCXP against the system’s size

curves in Fig. 4 to compare RCXP and FULL’s respective growth. As expected the curves show that FULL grows exponentially. These curves show by contrast a linear growth of RCXP. In EL the RP only allows battery fail actions, thus the sequences explored by RCXP grow linearly with the number of batteries. In CM the RP aims at serving all the coffees, thus RCXP’s linear growth with the number of coffees. For the other cases, we have observed that L14-2 RCXP’s exploration doesn’t grow with the number of trains as only one of them is observed, but grows linearly with the number of stations, for the reason that the observed train has to move as many times as there are stations. The ELV case is similar with the number of lift moves growing linearly with the number of storeys to serve. Finally the L14-1 case showed not a linear but exponential growth of RCXP. Indeed all the trains are observed in this experiment. Thus adding one train leads to explore all of its actions, as well as their interleavings with the actions of all the other trains.

As a general conclusion of these experiments we observe that RCXP provides a means for covering the reachable part of the abstraction, and that it behaves efficiently provided that the growth of the ATS resulting of CXP is controlled, and that the number of events involved in the relevance predicate is small.

7 Related Work

The closest methods to ours are those proposed in [7], where a relevance function is introduced, in [21] that extends [7] and in [22] that implements the process of [7] in Spec Explorer (SE). These methods are for generating tests. They generate a Finite State Machine (FSM) that is an under-approximated concretization of an Abstract State Machine (ASM) *may* predicate abstraction. Ideally, the methods seek for building the *true* FSM of the ASM, which contains only reachable links, but all of them. SE [22] proposes five techniques to implement the defined relevance function and to prune the search space: state grouping, directed search, parameter selection, state filtering and action restriction that are closely related to our approach, though with many differences. Our method begins by computing an abstraction in which the abstract states group the concrete states defined from a set of state predicates automatically extracted from a test purpose. In SE, the tester must give a state-based grouping expression. Then our

method computes a concrete under approximation by directed search (CXP). As in SE, the tester selects for that the values of many parameters, e.g. the initial number of coffees in the CM. In SE, the directed search is applied after all the pruning parameters have been given. For us, the covering of each abstract state and transition only once by CXP leads to a very strong state filtering. To relax this filtering, the tester designs a relevance predicate by observing which abstract states and transitions are not covered, in order to fix a new coverage goal. Then (s)he executes a new directed search (RCXP), that filters the states that satisfy the relevance predicate. The tester does not provide an execution’s maximum length as in SE, but termination is ensured thanks to a variant automatically computed from the relevance predicate. Last, our method allows action filtering by defining the RP, whereas in SE the tester strengthens some actions’ guards.

In [23,24], the set of abstraction predicates is iteratively refined in order to compute a bisimulation of the model’s semantics when it exists. Except by arbitrary limiting the number of refinement step (as suggested in [23]), none of these two methods is guaranteed to terminate, because the refinement step may would be repeated infinitely if no bisimulation quotient exists for the system. SYNERGY [25] and DASH [26] combine under-approximation and over-approximation for checking safety properties on programs. As we aim at proposing an efficient method for building a *reachable* under-approximation that covers all the abstract states and transitions w.r.t. a specification and a set of predicates, our algorithm does not refine the approximation but refines the under-approximation thanks to a relevance predicate associated to a variant, which guarantees the refinement process to terminate.

Some other work under-approximate an abstraction for generating tests. The tools Agatha [27], DART [28], CUTE [29], EXE [30] and PEX [31] also compute abstractions from models or programs, but only by means of symbolic executions [32]. This data abstraction approach computes an execution graph. Its set of abstract states is possibly infinite whereas it is finite with our method.

Another approach [33] for computing an under-approximation of a predicate abstraction is to characterize the abstract transitions not only as *may* ones, but also as *must+* and *must-*. Indeed, abstract sequences in the shape of $must-^* \cdot may \cdot must+^*$ can necessarily be instantiated as connected concrete sequences. An attempt to prolong an existing under-approximation thanks to these additional modalities is experimentally tested in [34].

8 Conclusion and Further Work

We have proposed a method for computing (or rather completing) a concrete under-approximation of a *may* abstraction. Building a new concrete transition is conditioned by the fact that it prolongs an existing reachable concrete sequence while satisfying a user defined relevance predicate. To ensure termination this predicate has to decrease a variant, and we propose a method for automatically extracting a variant out of the relevance predicate. We have experimented with

five case studies, for which we have achieved 100% coverage of the abstract transitions targeted by the relevance predicate, but with far less transitions than with the full exploration (except for one case as discussed in Sect. 6.2).

This work shows that the efficiency and success of RCXP depends on how the system is abstracted and the relevance predicate are chosen. Our results suggest that targeting a few transitions at a time with RCXP is preferable even if repeating the process is necessary. This corresponds to performing several test campaigns with different test objectives. We intend to experiment with various ways of abstracting and writing relevance predicates so as to investigate these methodological aspects. Also a more expressive relevance predicate language and a more semantic estimation of the variant's initial value are to be proposed.

References

1. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005). <https://doi.org/10.1007/b137241>
2. Utting, M., Legeard, B.: Practical Model-Based Testing. Morgan Kaufmann, Burlington (2006)
3. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
4. Bride, H., Julliand, J., Masson, P.A.: Tri-modal under-approximation for test generation. *Sci. Comput. Program.* **132**(P2), 190–208 (2016)
5. Godefroid, P., Jagadeesan, R.: On the expressiveness of 3-valued models. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 206–222. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36384-X_18
6. Julliand, J., Kouchnarenko, O., Masson, P.-A., Voiron, G.: Approximating event system abstractions by covering their states and transitions. In: Petrenko, A.K., Voronkov, A. (eds.) PSI 2017. LNCS, vol. 10742, pp. 211–226. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74313-4_16
7. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA, pp. 112–122 (2002)
8. Abrial, J.R.: The B Book. Cambridge University Press, Cambridge (1996)
9. Abrial, J.R.: Modeling in Event-B: System and Software Design. Cambridge University Press, Cambridge (2010)
10. Dijkstra, E.: Guarded commands, nondeterminacy, and formal derivation of programs. *Commun. ACM* **18**(8), 453–457 (1975)
11. Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.): ASM 2000. LNCS, vol. 1912. Springer, Heidelberg (2000). <https://doi.org/10.1007/3-540-44518-8>
12. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Log.* **1**(1), 77–111 (2000)
13. Bert, D., Cave, F.: Construction of finite labelled transition systems from B abstract systems. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 235–254. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40911-4_14

14. Dijkstra, E.: A Discipline of Programming. Prentice-Hall, Upper Saddle River (1976)
15. Cousot, P., Cousot, R.: Abstract interpretation frameworks. *J. Log. Comput.* **2**(4), 511–547 (1992)
16. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. *Formal Asp. Comput.* **27**(3), 573–609 (2015)
17. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): *Deductive Software Verification - The KeY Book - From Theory to Practice*. LNCS, vol. 10001. Springer, Heidelberg (2016). <https://doi.org/10.1007/978-3-319-49812-6>
18. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: 21st International Conference on Software Engineering, ICSE 1999, Los Angeles, California, USA, pp. 411–420. ACM (1999)
19. Bué, P.-C., Julliand, J., Masson, P.-A.: Association of under-approximation techniques for generating tests from models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 51–68. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21768-5_5
20. Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Softw. Pract. Exp.* **34**(10), 915–948 (2004)
21. Veanes, M., Yavorsky, R.: Combined algorithm for approximating a finite state abstraction of a large system. In: ICSE 2003/Scenarios Workshop, pp. 86–91 (2003)
22. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with spec explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) *Formal Methods and Testing*. LNCS, vol. 4949, pp. 39–76. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_2
23. Namjoshi, K.S., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 435–449. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_33
24. Păsăreanu, C.S., Pelánek, R., Visser, W.: Predicate abstraction with under-approximation refinement. *LMCS* **3**(1:5), 1–22 (2007)
25. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYN-ERGY: a new algorithm for property checking. In: SIGSOFT FSE, pp. 117–127 (2006)
26. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S., Thakur, A.V.: Proofs from tests. *IEEE Trans. Softw. Eng.* **36**(4), 495–508 (2010)
27. Rapin, N., Gaston, C., Lapitre, A., Gallois, J.P.: Behavioral unfolding of formal specifications based on communicating extended automata. In: ATVA (2003)
28. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI, pp. 213–223 (2005)
29. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/SIGSOFT FSE, pp. 263–272 (2005)
30. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: ACM CCS, pp. 322–335 (2006)
31. Tillmann, N., de Halleux, J.: Pex—white box test generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
32. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. *STTT* **11**(4), 339–353 (2009)

33. Ball, T.: A theory of predicate-complete test coverage and generation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 1–22. Springer, Heidelberg (2005). https://doi.org/10.1007/11561163_1
34. Julliand, J., Kouchnarenko, O., Masson, P.A., Voiron, G.: Two under-approximation techniques for 3-modal abstraction coverage of event systems: joint effort? In: TASE 2017, Nice, France, September 2017, to appear in IEEE



Using Dependence Graphs to Assist Verification and Testing of Information-Flow Properties

Mihai Herda^{1(✉)}, Shmuel Tyszberowicz^{2(✉)}, and Bernhard Beckert^{1(✉)}

¹ Karlsruhe Institute of Technology, Karlsruhe, Germany
{herda,beckert}@kit.edu

² The Academic College Tel Aviv Yaffo, Tel Aviv, Israel
tyshbe@tau.ac.il

Abstract. Information-flow control (IFC) techniques assist in avoiding information leakage of sensitive data to an observable output. Unfortunately, the various IFC approaches are either imprecise, thus producing many false positive alerts, or they do not scale. Using system dependence graphs (SDGs) to model the syntactic dependencies between different program parts is a highly scalable approach that enables to check whether the observable output depends on the sensitive input. While this approach is sound, security violations that it reports can be false alarms. We present a technique to overcome these problems by combining two existing approaches in a novel way. We show how each security violation reported by an SDG-based approach can be used to create a simplified program that can then be handled with a second approach to prove or disprove the reported violation. As the second approach we use deductive verification and test case generation. We show that our approach is sound, and demonstrate its benefits by means of examples. We discuss the challenges of implementing the approach using JOANA and KeY.

Keywords: Information flow control · Noninterference
System dependence graph · Deductive verification · Testing

1 Introduction

Software developers focus primarily on correctness with respect to functional requirements. However, when writing programs, non-functional requirements should also be considered first-class citizens. One non-functional requirement is related to confidentiality and requires the avoidance of illegal *information flow*; i.e., to prevent situations where *high* (confidential) input is leaking to *low* (public) output. This property is known as *noninterference* [13]. Intuitively it requires that the high input does not interfere with the low output. Thus, by observing the program's output, one cannot distinguish between different high inputs; i.e., if a program is executed twice with different high inputs but identical low inputs, an attacker will observe identical behaviors (an attacker can observe

low information but not high information). For example, if a credit card number is high input, unauthorized viewers (e.g., people working in the company’s warehouse) should not be able to observe this information, directly or indirectly.

We introduce the *low-equivalence* relation (\sim_L) to characterize program states that are indistinguishable for an attacker. A program state s is an assignment of values to variables. We assume that the input of a program is included in the initial state and that the output of a program is included in the final state. Two states, s and s' , are low-equivalent iff all low variables in s and in s' have the same value. In this work we consider only sequential programs, and we use the property called *noninterference* as it was first introduced by [13].

Definition 1 (Noninterference). *A program P is noninterferent iff for all initial states s_1, s_2 , either (1) $s_1 \sim_L s_2 \Rightarrow s'_1 \sim_L s'_2$, where s'_1, s'_2 are final program states after executing P in the initial states s_1 and s_2 , respectively, or (2) the program does neither terminate for s_1 nor for s_2 .*

This means that two program executions starting in two low-equivalent states must terminate at two low-equivalent states, or not terminate at all. This guarantees that low outputs are not influenced by high inputs. In the rest of the paper we will refer to noninterference with regard to a given high input and a given low output simply as noninterference.

Various approaches and tools for checking noninterference exist. Some have a high degree of automation, yet produce many false alarms as they over-approximate the information flow. Others are more precise, but require more effort and user interaction. We describe some of the main approaches. Approaches that are based on *System Dependency Graphs* (SDGs) syntactically compute the dependencies between the program statements and check whether the low output depends on the high input (see, e.g., [14]). Whereas they scale very well, such approaches over-approximate the actual dependencies in the program which results in false alerts. Approaches that are based on *type systems*, for instance [25], enforce secure information flow by assigning a security type (e.g., high or low) to the program variables and then checking whether the expressions in the program conform to the type system. *Logic-based* approaches (e.g. [7]), have a higher precision, i.e., they produce less false alarms, as they also consider the semantics of the program statements. However, they have a lower scalability. In those approaches the proof obligation is to show that the terminating states of two program executions are low-equivalent, assuming that the two initial states are low-equivalent. False alarms only occur when the system fails to find a proof in the allotted time even though the proof obligation is valid. Proving noninterference using this approach requires a quadratic number of program execution paths to be checked compared to proving a functional property. Approaches for *test cases generation* are also affected by this: a quadratic number of test cases is necessary to achieve the same coverage as for a functional property.

Our Contribution. In this paper we describe an approach to information-flow analysis that combines an SDG-based technique with deductive verification and test case generation. Note, however, that our approach enables other techniques

(e.g., static analyses) to be used together with the SDG-based technique. The highly scalable SDG-based approach is used to generate a simplified program for each possible security violation. These simplified programs are information-flow equivalent (with regard to the analyzed security violation) to the original program. This means that every illegal flow in the simplified program has a corresponding illegal flow in the original program, and vice versa. The necessary effort for the second approach, i.e., deductive verification and test case generation, is decreased by our approach by: (i) excluding pairs of high inputs and low outputs and the possible security violation they represent; (ii) excluding execution paths in the program; and (iii) excluding program statements.

We have implemented our approach using the JOANA tool [14] for the SDG-based analysis and the KeY system [2] for theorem proving and test case generation. Applying our approach on some example programs has shown that it increases the scalability of the deductive verification and the test case generation techniques, allowing them to focus just on those program parts that may contribute to the illegal flow of information.

We organize the paper as follows. In Sect. 2 we present a running example that will be used to demonstrate our ideas. Section 3 describes how we generate a simplified program. In Sect. 4 we show how the simplified program can help with verification, and in Sect. 5 we explain how the simplified program helps with information-flow test case generation. In Sect. 6 we discuss theoretical and technical details of our approach. In Sect. 7 we present work related to that done by us, and Sect. 8 concludes.

2 Running Example

```

int secure(int high, int low){
    if(low == 5){
        low = identity2(low, high);}
    else{
        if(low == 2){
            low = identity1(low, high);}
        else{
            low = identity2(low, high);}}
    return low;
}

int identity1(int low, int high){
    low = low + high;
    low = low - high;
    return low;
}

int identity2(int low, int high){
    return low;
}

```

Listing 1. Running example

The method `secure` in Listing 1 has a secret input `high`, a non-secret input `low`, and a public output—the method’s result. The method is noninterferent, as the result value does not depend on the value of `high`, and this can be proved using deductive verification. The proof requires to consider nine symbolic execution paths: we need to analyze two program runs as required by Definition 1, and for each of those we need to consider three cases, namely that the input `low` (a) is 5, (b) is 2, or (c) has any other value. SDG-based techniques for checking noninterference will report a security violation, as the called method `identity1` contains a *syntactic* dependency between its return value (that gets afterwards

assigned to `low`) and the parameter `high`. This dependency, however, only affects the path in which the initial value of `low` is 2. Hence, the other two execution paths can be guaranteed to be secure by the SDG-based method. In the following section, we will explain how we can simplify the running example program and we show the advantages of this simplification in proving the noninterference of the program using a logic-based approach.

3 Generation of the Simplified Program

To check the noninterference property for a given program, we combine the following two approaches: (i) *SDG-based information-flow analysis*, and (ii) *deductive theorem proving* or *test-case generation* or both. The SDG-based analysis is purely syntactic, highly scalable, and sound. However, some of the reported noninterference violations may be false positives. But even if there are false warnings, this analysis allows us to exclude some execution paths and statements that are guaranteed not to have any effect on the noninterference property. The second approach then only needs to deal with those parts of the program that can potentially lead to an illegal flow of information according to phase (i).

3.1 SDG-Based Information-Flow Analysis

In the first step we use an SDG-based static analysis technique for determining whether there exists a potential dependence between high input and low output. The usefulness of Program Dependence Graphs (PDGs) [12] and System Dependence Graphs (SDGs) [19] in the context of information-flow security has been first noticed in the nineties [29]. Decades of research in this area have resulted in JOANA, a tool that statically analyzes Java programs of up to 100K LOC [14]. We explain how SDG-based information-flow analysis works using the JOANA tool. Our approach can be realized using other SDG-based analysis tools as well. We define SDGs for deterministic inter-procedural programs that consist of variable assignments, branching, loops, and function calls. For those programs we also assume that a control-flow graph (CFG) is available.

The noninterference property is specified by annotating program parts corresponding to high inputs and low outputs. JOANA builds an SDG for the program. An SDG is a directed graph consisting of interconnected PDGs, where a PDG represents a single procedure of a program as a directed graph. A detailed discussion on SDGs can be found, e.g., in [19]. Nodes in the SDG represent program statements, conditions, or input parameters, and edges represent dependencies between the nodes; i.e., an edge between nodes exists if the execution of one node may depend on the outcome of the other node. The CFG nodes of a program are also present in the SDG. There are roughly two types of edges in an SDG: data dependence edges, representing possible direct dependencies and control dependencies which represent possible indirect dependencies (other dependencies for supporting object orientation and multi-threading are defined in [17]). Whether an edge exists between two nodes in the SDG is determined

syntactically by analyzing the control flow graph of the analyzed program. An overview of formal definitions of the two types of dependencies can be found in [31]. We present the two standard ones here. To determine whether a node is data-dependent on another node, we define for each CFG node n two sets, Def_n and Use_n , which contain the program variables that are defined in the node and that are used in the node, respectively. Node n' is data dependent on n if there is a variable v that is used in n' and defined in n , and there is a path from n to n' in the CFG such that v is not defined on any node between n and n' on that path. The standard definition of a control dependency between two nodes states that node n' is control-dependent on node n if the choice of the outgoing edge from n in the CFG determines whether node n' is reached. Note that it is undecidable whether a CFG path is realizable during the execution of the program, i.e., some paths in the CFG represent executions that cannot actually take place, thus the CFG is an overapproximation of the program behavior. Since the SDG edges are defined using CFG paths, they too are an overapproximation of the dependencies in the program. In the rest of the paper, we refer to the program execution described by a CFG path as *execution path*. If an SDG path is a sub-path of an execution path, we say that that execution path corresponds to the SDG path. An SDG path has one or more corresponding execution paths.

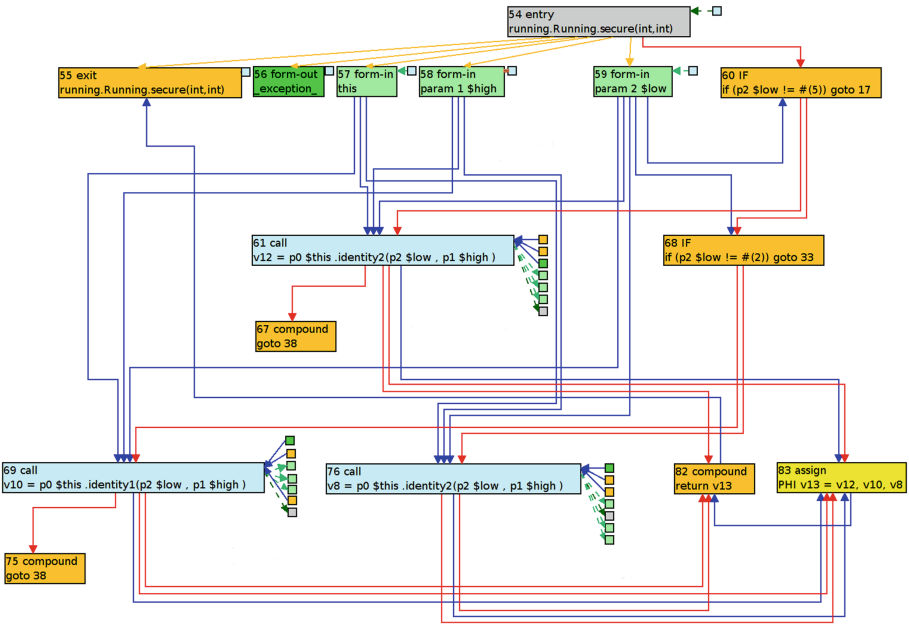


Fig. 1. The SDG of the running example

Figure 1 presents the SDG generated by JOANA for the `secure` method in our running example. The dependencies inside the three calls of the

methods `identity1` and `identity2` are hidden, and only the method call nodes are shown. We use this simplification also when describing paths in the SDG. Node 58 represents the parameter `high` of method `secure`, and node 55 the method exit node. The two nodes are annotated as `high` and `low`, respectively.

SDG-based information-flow analysis approaches detect illegal information flows through graph analysis, using a special form of conditional reachability analysis—*slicing* and *chopping*—at the SDG level. A security violation is reported when there is an SDG path from a high node to a low node. JOANA is sound [31], i.e., any illegal information flow in the program can occur only in the execution paths corresponding to an SDG path from a high node to a low node. Thus, if no such SDG path is found, the program is noninterferent. However, when an SDG path from a high to a low node exists, the program may still be noninterferent. To distinguish between real and false security alarms, we use a second technique (verification or test case generation).

For the running example JOANA reports a security violation that contains an SDG path from the parameter `high` to the return value of the method `secure`. JOANA successfully determines that there is no dependency between the parameter `high` and the return value of the two calls to the method `identity2`. Since the edges in the SDG represent purely syntactical dependencies, JOANA cannot show the same for the call of `identity1`. Thus, the SDG path $58 \rightarrow 69 \rightarrow 82 \rightarrow 55$ is reported as a possible violation (the numbers correspond to the node ids in Fig. 1). For the two execution paths where `identity2` is called the absence of any illegal information flow is shown. In the second, more precise, approach the analysis of these two execution paths can be skipped.

3.2 Generating the Simplified Program

SDG-based information-flow analysis approaches use slicing and chopping techniques. Program slicing [32] removes statements from a program in order to reduce its size and complexity while retaining some specified aspects of its behavior. Slices are defined with respect to a *slicing criterion*, i.e., a statement or a variable in the program. Forward slicing is used to compute the program statements which are influenced by the criterion statement, whereas backward slicing finds the programs statements that influence that criterion statement.

Definition 2 (Forward, Backward Slice). *Given an SDG and a criterion statement represented by a node n in the SDG, we define the forward slice $S_{fw}(n)$ and the backward slice $S_{bw}(n)$ as the following sets:*

- $S_{fw}(n) = \{n_s \mid n_s \text{ is reachable from } n \text{ in the SDG}\}$
- $S_{bw}(n) = \{n_s \mid n \text{ is reachable from } n_s \text{ in the SDG}\}$

A node that is not contained in the backwards slice of some node n cannot influence n , and a node not contained in the forward slice of n cannot be influenced by n . The slice nodes also determine a (sliced) program that is constructed from the original program by removing those statements which are not in the slice. One property of the backward slice is that, for every input, the variables in the criterion statement have the same values as in the original program.

Definition 3 (Chop). *Given two nodes n_h and n_l in an SDG, we define the chop of these nodes as the set $C(n_h, n_l) = S_{fw}(n_h) \cap S_{bw}(n_l)$.*

In the case of information-flow analysis, program chopping provides information on the program statements that are included in a path from a high program part to a low program part. A security violation reported by the SDG-based approach has a corresponding chop for a high input and low output pair represented by nodes in the SDG. This chop represents an overapproximation of the nodes that are influenced by the high input and that influence the low output.

The chop that was created by JOANA for the running example contains only one path. We know that only the statements of the nodes in this path can be relevant for the potential illegal information flow. Using a chop and the analyzed program we can generate a simplified program, which we call *chop-based program*. The chop-based program of the running example is shown in Listing 2.

Definition 4 (Chop-based program). *For a given program P and a chop $C(n_h, n_l)$, the chop-based program P_C is constructed by removing all statements from P that have no corresponding node in $C(n_h, n_l)$.*

Theorem 1. *Given a program P and a chop $C(n_h, n_l)$, if the noninterference property with respect to the high input corresponding to n_h and the low output corresponding to n_l holds for the chop based program P_C , then it holds for the original program P as well.*

Proof. An illegal information flow may occur only on the execution paths determined by the SDG paths from a high source to a low sink, and all these SDG paths are present in the chop reported as a security violation. Thus, it suffices to show that the noninterference property holds for the program executions determined by the chop. Since the chop-based program contains by construction all statements corresponding to the nodes in the chop, the execution paths determined by the chop in the original program are a subset of the execution paths of the chop-based program. Thus, if the noninterference property holds for the chop-based program, it must also hold for every execution path described by the chop, and hence also for the original program. \square

```
int secure(int high, int low) {
    low = identity1(low, high);
    return low;
}
```

Listing 2. Chop-based program for the running example

```
int secure(int high, int low) {
    if (low == 5) {
        disruptExecution();
        low = identity2(low, high);
    }
    else {
        if (low == 2) {
            low = identity1(low, high);
        }
        else {
            disruptExecution();
            low = identity2(low, high);
        }
    }
    return low;
}
```

Listing 3. Simplified program for the running example

The chop-based program is greatly simpler than the original and this fact can help to considerably reduce the effort for deductive verification. However, some statements that affect path conditions (i.e., conditions on the inputs that need to hold for an execution path to be taken) under which an illegal information flow can occur in the original program may be lost in the chopping process. Therefore, the chop-based program may allow executions that are not possible in the original program. For this reason, an illegal information flow in the chop-based program does not necessarily have a corresponding illegal flow in the original program, since the path conditions that need to hold for the illegal information flow may be unsatisfiable in the original program. Suppose, e.g., that the method `identity1` in the running example is secure only if the condition `low == 2` holds when the method is called. This condition holds in the original program, but not in the chop-based program. In this case the original program would be noninterferent, but the chop-based program would contain an illegal information flow. Thus, whereas the verification effort can be significantly reduced for some programs, the missing path conditions may cause the noninterference property to become impossible to verify for other programs. Furthermore, the chop-based program is of little use for test-case generation, since the test data for this program may take another path when used to test the original program. For example, the input parameter `low` in the original program must be 2 for the `identity1` method to be called, but this is not true for the chop-based program. Thus, the coverage achieved when testing the simplified program does not translate to a coverage in the original program. Moreover, it may be that an information-flow leak detected when testing the chop-based program is not a leak in the original program. E.g., for the method `identity1` in the running example to be called to catch the “potential” information flow from the parameter `high` to its return value, the value of `low` has to be 2 in order for the leak to be observable in the original program. Since the path condition is no longer present in the chop-based program, the test-case generation approach can generate two inputs that lead to a potential leak in the chop-based program but not in the original program.

To overcome the problems of the chop-based programs, we introduce a new kind of programs—*simplified programs*. A simplified program is based on the backward slice of the low output, and has some execution paths excluded. We decide which paths to exclude by analyzing both the SDG of the entire program and the chop, to determine whether a branching node (e.g., a node representing an if-statement) has to be true or false for an illegal information flow to occur.

Definition 5 (Analysis of branching nodes). *Let $n_b \in S_{bw}$ be a conditional branching node in the backward slice of some node n_l corresponding to the low output. Let N_{true} be the set of successor nodes following the true branch of n_b in the CFG, and let N_{false} be the successor nodes following the false branch in the CFG. We define n_b to be a condition that must be true if the analyzed chop $C(n_h, n_l)$ contains nodes from N_{true} and no nodes from N_{false} . Conversely, n_b is defined as must be false condition if the chop contains nodes from N_{false} and no nodes from N_{true} .*

Theorem 2. *Given a high input and a low output corresponding, respectively, to the SDG nodes n_h and n_l , and a branching node n_b that must be true (resp. false), any execution path of the original program along the false (true) branch of n_b will not lead to an illegal information flow from n_h to n_l .*

Proof. This property results from the soundness of the SDG-based approach: an illegal information flow from n_h to n_l can occur only along an execution path determined by chop $C(n_h, n_l)$. However, because the node n_b must be true (false), we know that no successor of n_b is in the chop, thus the chop does not include any execution path along the false (true) branch of n_b . \square

This allows the exclusion the execution paths that do not lead to an illegal information flow according to Theorem 2 from the analysis in the second step. We exclude these paths by adding a special statement that disrupts the symbolic execution at the beginning of a false branch for a branching statement that *must be true* and at the beginning of a true branch for a branching statement that *must be false*. When the program is symbolically executed for verification and reaches a disruptive statement, the proof closes automatically for that branch. Test case generation also immediately halts for that path once the symbolic execution reaches a disruptive statement. We can now define the simplified program.

Definition 6 (Simplified program). *Let n_h and n_l be two nodes in the SDG of a program P (corresponding to a high input and a low output). We construct the simplified program P_S from P by:*

1. *Removing all nodes that are not in the backward slice $S_{bw}(n_l)$.*
2. *Analyzing the remaining conditional nodes and adding disruptive statements on their true resp. false branches that cannot lead to an illegal information flow according to Theorem 2.*

Theorem 3. *Given a high input and a low output in a program P with corresponding nodes n_h and n_l in the SDG for P , the simplified program constructed according to Definition 6 is noninterferent with respect to n_h and n_l if and only if the original program is noninterferent with respect to n_h and n_l .*

Proof. If the simplified program is noninterferent, then none of its execution paths leads to an illegal information flow. The simplified program contains all execution paths determined by $C(n_h, n_l)$, therefore none of the chop paths leads to an illegal information flow. Due to the soundness of the SDG-based approach, an illegal information flow can occur only along an execution path determined by the chop. Therefore also the original program must be noninterferent.

If the original program is noninterferent, then the backward slice $S_{bw}(n_l)$ of the low output is noninterferent as well, since for every input the low output of the backward slice is identical to that of the original program. As shown in Theorem 2, adding disruptive statements excludes only execution paths that are guaranteed not to lead to an illegal information flow in the original program. Hence the simplified program must be noninterferent as well. \square

Theorem 4. *Given a high input and a low output in a program P with corresponding nodes n_h and n_l in the SDG of P , two concrete high inputs h_1 and h_2 for n_h lead to two different low outputs, l_1 and l_2 , in n_l in the simplified program P_S if and only if h_1 and h_2 lead to the same two different low outputs l_1 and l_2 in the original program P .*

Proof. The simplified program P_S is a backward slice with respect to the low output, in which some paths that are guaranteed not to lead to an illegal information flow are excluded. The two high inputs leading to two different low outputs in P_S cannot have taken one of the excluded paths, otherwise these paths would not have been excluded. Since the remaining, not excluded, execution paths of P_S are those of the backward slice with respect to the low output, the inputs that take those execution paths will lead to the same low outputs in the original program. If the two high inputs lead to two different low outputs in the original program, the simplified program will lead to the same two different low outputs, because of the slice property and because the execution paths could not have been excluded since they do lead to an illegal information flow. \square

The chop reported by JOANA for the running example contains nodes 58, 69, 82, and 55 (Fig. 1). In the backward slice of the low output (i.e., of node 55), there are two branching nodes, 60 and 68, corresponding to the two if-statements in the example program. Analyzing the two branching nodes, we automatically determine that for an illegal information flow to be possible the first if-statement has to take the *false* branch and the second if-statement has to take the *true* branch. The program in Listing 3 is the simplified program of the running example. While in general the backward slice of the return statement can be much smaller than the original program, in our example it contains the entire program. Nevertheless, our approach determines that the paths leading to the call of `identity2` cannot lead to an illegal information flow, and it adds two statements `disruptExecution()` which stop symbolic execution when verifying the running example or generating test cases—as we will see in the next sections.

4 Verification of the Simplified Program

We now show how using the simplified program can assist in reducing the verification effort compared to the effort needed to verify the original program. Since our implementation uses KeY we begin by explaining how KeY works.

The KeY system is a deductive program verification tool for Java, which is based on JavaDL, a first-order dynamic logic for Java. Properties of the program are specified as method contracts and auxiliary specifications such as loop invariants using an extension of the Java Modeling Language (JML) [24]. KeY transforms the specification and the program code into *proof obligations* formalized in JavaDL and performs a proof using a *sequent calculus*. For functional properties the proof obligation has the form $\Longrightarrow \text{Pre} \rightarrow \langle P \rangle \text{Post}$, meaning that the program P that is executed in a prestate in which the precondition `Pre` holds terminates in a poststate where the postcondition `Post` will hold. JavaDL

provides an *update* operator that can be used to “store” changes to the program state and simplify them if possible.

During proof construction, the program is symbolically executed, statement by statement, using the appropriate rules for Java programs in the KeY calculus. Rules are applied automatically according to the heuristic provided by KeY. The state changes affected by each individual statement are (symbolically) expressed and added to the update. Updates provide a compact representation of the symbolic state, and are changed according to the program state changes. When the entire program has been symbolically executed, the changes recorded in the update are simplified and applied to the postcondition. After this step, KeY attempts to show the unsatisfiability of the remaining sequents, which contain only-first order formulas. It is important to note that a split in the program, due to a branching statement, causes also a split in the proof. Thus the resulting proof tree contains at least one branch for each execution path in the program.

The logic JavaDL, the sequent calculus, and the JML specification language have been extended such that KeY supports the verification of information-flow properties for sequential Java programs [7]. The proof obligation that needs to be resolved expresses that two program runs that start in low equivalent states will also terminate in low equivalent states (similarly to Definition 1 of noninterference). Using KeY to verify noninterference for the running example requires 771 rule applications. After symbolic program execution is finished, nine proof branches remain to be closed—one for each combination of paths in the two program runs, as explained in Sect. 2. For the chop-based program in Listing 2, verification needs 298 rule applications, and only one proof goal remains to be closed after symbolic execution (as there is only one possible path combination). However, as already explained, noninterference of the chop-based program is a sufficient but not a necessary condition for noninterference of the original program. To get a necessary condition ensuring that noninterference is preserved when the program is simplified, path conditions need to be preserved, which is the case in the simplified program according to Definition 6. The verification of the simplified program in Listing 3 requires 511 rule applications. In this case, the symbolic execution halts when one of the two program runs reaches a path that has already been deemed secure by JOANA, and the corresponding proof branch is closed. Thus, of the nine proof goals remaining after symbolic execution, eight are trivially closed—sometime even before their symbolic execution is finished.

The running example showcases how the SDG-based approach can assist verification by excluding statements and execution paths from the program. The exclusion of execution paths is especially useful when dealing with information-flow properties such as noninterference. If the original program has n execution paths, the verification process must prove that the noninterference property holds for n^2 paths. Adding a single disruptive statement, the number of paths that need to be verified in the simplified program drops to a number between $(n - 1)^2$ and $n^2/4$ (depending on whether the affected condition is at the top level or not). Thus, the number of execution paths that need to be analyzed with the theorem prover can drop to a quarter of those required for the original program.

The statements removed from the original program can also lead to a dramatic decrease of the verification effort. Consider the example in Listing 4. The method `secure` contains a call to the method `sort` that has no influence on any potential information flow from the parameter `high` to the return value of `secure`. The verification of the method `secure` would normally be done using a verified method contract for `sort`. Using the method contract in the noninterference proof of `secure` increases the size of the proof. Moreover, the method `sort` itself needs to be specified and verified before it can be soundly used in the proof of `secure`, thus increasing the user’s workload. For the example shown in Listing 4 a trivial contract for the method `sort` is sufficient. However, the user has to look into the code, notice that the call of the `sort` method has nothing to do with a potential information flow, and then to specify and verify it. Our approach can automatically detect such statements and soundly remove them.

```
int [] a;
int secure(int high, int low) {
    low = high * 0;
    sort(a);
    return low;
}
```

Listing 4. Example containing a complex method call

```
int [] a;
int secure(int high, int low) {
    low = high * 0;
    a = new int [5];
    for(int i=0; i<a.length; i++){
        a[i] = low;}
    return low;
}
```

Listing 5. Multiple low outputs example

The individual analysis of simplified programs corresponding to high input and low output pairs can also benefit verification. Consider the program in Listing 5. We regard the parameters `high` and `low` to be the high resp. low inputs, and both the return value of the `secure` method and the potentially thrown exception of this method are regarded as low outputs. In this case, two simplified programs are generated—one for the potential flow from `high` to the return value and one for the potential flow from `high` to the exception. For the first pair, the array generation with `new` and the loop initializing the array are removed, thus making the absence of this flow trivial to verify. For the second pair, no statements are removed; however, the verification engineer can verify the absence of a thrown exception by specifying the normal termination of the method as a functional property and doing a functional proof.

Most tools support the classical noninterference property, in which the low-equivalence relation is equality. KeY additionally allows the noninterference property to be defined by requiring object structures in the two low-equivalent program states to be (only) isomorphic. This is useful for showing noninterference for methods that create new objects, as two independent program runs will generate different references but isomorphic structures. KeY also allows an expression to be declassified, i.e., the attacker is allowed to know the value of the declassified expression, but no more than that. Both these extensions of the noninterference property are relaxations, and thus a program that fulfills the noninterference property as defined in Definition 1 fulfills the extended properties as

well. Hence, we can use our approach to generate the chop-based or the simplified program even when attempting to prove the extended noninterference property.

5 Testing the Simplified Program

When the verification of the noninterference property fails, testing can be used to either find an illegal information flow or to gain confidence in the program by running a high coverage test suite. For both goals, manually writing test cases that achieve a good path coverage is a difficult task. For noninterference this problem is further escalated because the test must run the program twice to check for an illegal flow. The simplified program can be used to reduce the effort required for generating noninterference test cases.

Symbolic execution-based test generation is a well established technique for generating test suites with a high path coverage. However, this technique suffers from the *path explosion problem*, i.e., the number of possible execution paths of a program is very large, resulting in a reduced scalability. For the noninterference property, the number of paths is quadratic when compared to usual functional properties. This is where our technique comes into play. By testing the simplified program rather than the original program, the number of execution paths that need to be considered can decrease, thus improving the scalability.

We have implemented a prototypical test-case generator (TCG) for information flow properties by extending the automatic TCG of KeY (see [2, Chap. 12]). The TCG attempts to generate two low equivalent inputs for each pair of path conditions from the two program runs. The TCG starts by loading the same proof obligation for noninterference as would be done when verifying the property. We no longer require auxiliary specification, but unwind the loops and inline method calls for a bounded number of times (specified by the user). The program is symbolically executed twice and we obtain a proof tree that has a proof goal for each pair of execution paths. A model for a formula represented by such a proof goal satisfies both path conditions in addition to the low-equivalence of the two inputs. An SMT solver (Z3) is used to find a model for these formulas. The user now has two options. With the first option, only test cases with input pairs that lead to not low-equivalent output pairs are generated. These tests showcase a counterexample. With the second option, the postcondition is ignored and the TCG attempts to generate a test case for each (satisfiable) pair of path conditions. These tests can be used to validate the results of verification. An noninterference test consists of two executions of the method under test, with two low-equivalent inputs, that result in two outputs. A test oracle checks whether the two outputs are low equivalent, determining whether the test is successful.

The running example contains three execution paths and, thus, the TCG will attempt to generate 3^2 input pairs. However, only three pairs are low equivalent as the branch that is taken is determined by the low input. Thus, the TCG generates a test suite with three noninterference test cases. When running the TCG on the simplified program, only one test case will be generated for the case

in which both `low` inputs are 2. This is possible due to the inserted statements in the simplified program that disrupt symbolic execution. For the simplified program, the TCG attempts (and succeeds) to generate a noninterference test case only once, compared to the nine attempts for the original program. Hence, in this case eight test-case generation attempts (calls to the SMT solver) that cannot lead to a noninterference property violation are soundly skipped. Testing the simplified program can reduce the number of generated test cases also by removing program statements that are not relevant for computing the low output. When testing the program shown in Listing 4, the method `sort` would be inlined during the symbolic execution phase, thus requiring a large number of execution paths to be tested. By removing the call to `sort`, the simplified program contains only one execution path. Removing statements also leads to reduction in the computational resources needed to run the information-flow tests.

An additional benefit of using our two-step approach is the fact that we treat each high input and low output pair separately. If an illegal information flow is found during testing, the user can easily identify the high input and low output of the leak by noticing for which simplified program the test fails.

It is difficult to define an appropriate coverage criterion for noninterference test suites. For functional properties full path coverage is ideal, however, this is not achievable in the general case because some execution paths may have unsatisfiable path conditions, i.e., no input will result in such an execution path being taken. When extending the path-coverage criterion to information-flow testing, requiring a test case for every pair of execution paths in the two program runs, the low-equivalence requirement for the two inputs poses an additional problem because many pairs of execution paths are incompatible. This results in a low path coverage that is not an indication for a badly designed test suite. In the running example, only for three out of the nine path pairs a noninterference test case can be generated; resulting in a path coverage of only 33%. Execution paths excluded from the original program have path conditions that depend on the high input. Those paths are thus determined only by the low input and are likely to form incompatible pairs with other paths. By excluding them, the achieved path coverage becomes a better indicator for the quality of testing.

6 Discussion

The novelty of our approach is that we soundly bridge the gap between two kinds of approaches: the scalable over-approximating IFC approach and the more precise but less scalable one. This is achieved by automatically transforming the output of the SDG-based approach into an input of a more precise approach, thus simplifying the analyzed program. Furthermore, the simplified program that we generate is not a mere slice of the original program; by taking advantage of the noninterference property, we also exclude entire branches from the analysis with the precise approach. We have shown that a single branch exclusion can lead the more precise approach to handle only one quarter of the execution paths that

would otherwise need to be handled. This is a crucial advantage, as existing precise IFC approaches suffer from an exacerbated path explosion problem.

We defined two types of simplified programs: the chop-based program and the simplified program; both can be useful for verification, but sometimes the chop-based program makes verification impossible. To help the user chose between the two versions, we devise a criterion based on the following theorem:

Theorem 5. *Given the SDG of a program and a chop representing a reported security violation, then if every node that represents a conditional branching statement in the SDG is also present in the chop, the chop-based program is noninterferent if the original program is noninterferent.*

Proof. All branching conditions depend on the high input, otherwise the corresponding branching nodes would not be in the chop. The consequence for such a program is that all execution paths must lead to the *same* low output, otherwise the low output would be conditionally dependent on a high output. Removing statements that do not depend on the high input would change the value of this output but the noninterference property remains unaffected. \square

For such programs the chop-based program is better suited to assist verification, since no path can be excluded when generating the simplified program as the conditional branching nodes have at least one true and one false successor in the chop. The simplified program in this case is the backward slice based on the low output of the original program and it may contain more statements than the chop-based program. The inputs that lead to an illegal information flow in the chop-based program will as well lead to one in the original program, and thus the chop-based program can even be used for testing. We check whether the chop fulfills the condition described in Theorem 5, and respectively use the chop-based or the simplified program.

The implementation of our approach uses JOANA and KeY, two tools that do not work on the same programming language. While JOANA works on Java bytecode that was transformed into a single static assignment (SSA) form, KeY works on Java source code. For the soundness of our implementation (but not for the soundness of our approach), we must assume that compiling a Java program into bytecode does not change the information-flow properties of the program. Moreover, this also raises the issue of mapping bytecode statements into source code statements. We are able to determine the source code line (which may contain more than one source code statement) from which a bytecode statement originates. However, a source code statement can be compiled into more than one bytecode statement and, due to the SSA form, some source code statements may not even have a corresponding bytecode statement. Using these tools it is thus impossible to generate the chop-based and the simplified program as defined in Sect. 3. Instead, we generate an overapproximation of the chop-based and the simplified program by removing a line in the source code only if: (i) the SDG contains a node corresponding to a bytecode statement originating from that line, and (ii) no such node is in the chop (for the chop-based program) or in the backward slice of the low output (for the simplified program). To avoid multiple

source code statements on the same line, we preprocess the source code program and transform it into a program with only one statement per line.

SDG-based slicing as done in JOANA can result in a program that is not executable or that may incorrectly handle jump statements, such as `goto`, `break`, or `continue`. Our approach is nevertheless feasible, as various solutions that enhance SDG-based slicing and enable the generation of executable program slices from SDGs have been proposed; see, e.g., [1, 5, 18]. We obtain executable slices by not removing lines containing certain types of statements such as constructors or static initializers and by supporting only programs without jump statements.

While JOANA supports full Java (minus reflection), KeY handles sequential Java programs only. KeY also requires that the source code or method contracts of library methods to be available. The implementation of our approach using these two tools thus supports the same Java subset that KeY does.

The scalability of our approach is bounded by the scalability of the two tools it uses. JOANA is able to handle programs of up to 100k LOC, whereas KeY can handle programs of up to 1000 LOC. The generation of the backward slice and the chop that are necessary for constructing the simplified program are anyway the operations that JOANA performs in its analysis, and the analysis of the branching nodes is a graph reachability problem similar to how slicing is done in the SDG-based approach. The main bottleneck of our approach is therefore the analysis using KeY rather than the generation of the simplified program. Thus, the most favorable case for using our approach is when an original program that is too large for KeY is simplified and reduced to a size that KeY can handle. This is achievable either by removing program parts as done for the program in Listing 4 or by excluding execution paths as done for the running example.

7 Related Work

A lot of research has been done on IFC, dating back to the works described in [8, 9] and later in [13]. A survey on approaches for IFC is found in [28]. In what follows, we describe some approaches that are similar to ours.

The *Hybrid Approach* [22] also aims to combine automatic dependency-graph analysis and a theorem proving. The user first attempts to show noninterference using JOANA. If user suspects the reported violation to be a false alarm, he must identify the cause of the alarm and extend the program such that the low output is overwritten with a value that does not depend on the high input. The extended program is rechecked by JOANA, and KeY is used to show that the extended program computes the same low output as the original. This approach improves the precision provided by JOANA. However, the communication between tools is done manually; there is no assistance in finding the causes of the false alarms. The results provided by JOANA are not used to simplify the program. In fact, by extending it, the program verified by KeY becomes even more complex.

The *Combined Approach* (CA) [6] uses theorem proving to show that SDG edges corresponding to method calls do not represent a real dependency in the

program. If this is shown for an edge on every path from a high input to a low sink, then the respective violation is disproved. Using our approach individual statements that have no effect on a potential illegal information flow can be removed while the CA works on a method level of granularity. The CA and the approach shown in this paper are orthogonal. In fact, the approach described here can be used to simplify the proof obligations generated by the CA. The CA is well suited for programs where the part that cannot be handled by the SDG-based approach is concentrated in a called method. If the syntactic dependencies between the high input and low output are spread throughout the program, then the whole program needs to be verified and no simplification is done to it. In such cases our approach, however, can still profit from the SDG-based analysis.

Another combination of SDG-based approaches and theorem provers is by checking the satisfiability of the path conditions for the execution paths determined by the reported security violation [17,30]. If a path condition is unsatisfiable, then the respective execution path cannot lead to an illegal information flow. A program input that satisfies the path condition serves as a “witness” and the user can analyze the program execution with that single input and check whether an illegal information flow occurs. This is a difficult task, especially for indirect dependencies. The noninterference tests that we generate have two inputs and show the illegal information flow more clearly.

A sound information-flow testing mechanism based on standard testing techniques and on a combination of dynamic and static analyses was proposed in [23]. Once a path coverage property has been achieved, a conclusion regarding noninterference can be established. A tool that uses symbolic execution in combination with a form of self-composition for noninterference testing of C programs is presented in [26]. A logic-based approach to detect and generate exploits for information flow properties and present them as a JUnit test is described in [10]. Information flow test case generation was also done in [15,20].

The combination of static and dynamic analysis has been employed in other areas as well. Testing can be used to better understand the reasons of a failed proof attempt; see, e.g. [27]. An incomplete proof can also be used to generate a simplified software monitor [3], thus reducing the execution overhead of the monitored software. Programming languages with dynamic features, e.g. JavaScript, are difficult to handle for purely static analyses. This led to the development of hybrid static-dynamic approaches for these languages as, e.g., in [4,16]. Dynamic techniques have been used to generate program invariant candidates, which can then be used to aid static techniques [11,21].

8 Conclusion

We have explored an approach for proving or testing the noninterference property of a program, that uses SDG-based analysis to remove irrelevant program parts and to exclude execution paths that do not lead to an illegal information flow. For each pair of high input and low output we generate a simplified program. We have shown that the simplified program is information-flow equivalent to

the original program. Thus, a noninterference proof for the simplified program is also one for the original program. The same holds for the counterexample. The examples in the paper show how the simplified program assists in the verification and testing of the noninterference property.

We have discussed implementation details of our approach using JOANA as an SDG-based analysis tool and KeY as both a theorem prover and a test case generator. Because the two tools work respectively on Java bytecode and Java source code, our prototypical implementation generates an overapproximation of the simplified program. This is only an engineering challenge, our approach can be implemented using tools that work on the same programming language.

References

1. Agrawal, H.: On slicing programs with jump statements. In: ACM SIGPLAN Notices, vol. 29, pp. 302–312. ACM (1994)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book: From Theory to Practice. LNCS, vol. 10001. Springer, Cham (2016). <https://doi.org/10.1007/978-3-319-49812-6>
3. Ahrendt, W., Chimento, J.M., Pace, G.J., Schneider, G.: Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods Syst. Des.* **51**(1), 200–265 (2017)
4. Artho, C., Biere, A.: Combined static and dynamic analysis. *Electron. Notes Theor. Comput. Sci.* **131**, 3–14 (2005)
5. Ball, T., Horwitz, S.: Slicing programs with arbitrary control-flow. In: Fritzson, P.A. (ed.) AADeBUG 1993. LNCS, vol. 749, pp. 206–222. Springer, Heidelberg (1993). <https://doi.org/10.1007/BFb0019410>
6. Beckert, B., Bischof, S., Herda, M., Kirsten, M., Kleine Büning, M.: Combining graph-based and deduction-based information-flow analysis. In: Workshop on Hot Issues in Security Principles and Trust (HotSpot), pp. 6–25 (2017)
7. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Information flow in object-oriented software. In: Gupta, G., Peña, R. (eds.) LOPSTR 2013. LNCS, vol. 8901, pp. 19–37. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14125-1_2
8. Denning, D.E.: A lattice model of secure information flow. *Commun. ACM* **19**(5), 236–243 (1976)
9. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. *Commun. ACM* **20**(7), 504–513 (1977)
10. Do, Q.H., Kamburjan, E., Wasser, N.: Towards fully automatic logic-based information flow analysis: an electronic-voting case study. In: Piessens, F., Viganò, L. (eds.) POST 2016. LNCS, vol. 9635, pp. 97–115. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49635-0_6
11. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* **69**(1), 35–45 (2007)
12. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* **9**(3), 319–349 (1987)
13. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Symposium on Security and Privacy (SP), pp. 11–20 (1982)

14. Graf, J., Hecker, M., Mohr, M.: Using JOANA for information flow control in Java programs - a practical guide. In: Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, Aachen, 26 Februar–1 März 2013, pp. 123–138 (2013). <http://subs.emis.de/LNI/Proceedings/Proceedings215/article6906.html>
15. Gruska, D.P.: Information flow testing. *Fundamenta Informaticae* **128**(1–2), 81–95 (2013)
16. Hackett, B., Guo, S.Y.: Fast and precise hybrid type inference for JavaScript. *SIGPLAN Not.* **47**(6), 239–250 (2012)
17. Hammer, C., Krinke, J., Snelting, G.: Information flow control for Java based on path conditions in dependence graphs. In: Symposium on Secure Software Engineering, pp. 87–96 (2006)
18. Harman, M., Lakhota, A., Binkley, D.: Theory and algorithms for slicing unstructured programs. *Inf. Softw. Technol.* **48**(7), 549–565 (2006)
19. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.* **12**(1), 26–60 (1990)
20. Hritcu, C., Lampropoulos, L., Spector-Zabusky, A., de Amorim, A.A., Dénès, M., Hughes, J., Pierce, B.C., Vytiniotis, D.: Testing noninterference, quickly. *J. Funct. Program.* **26** (2016). <https://doi.org/10.1017/S0956796816000058>
21. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR. *J. Autom. Reason.* **60**(3), 337–363 (2018)
22. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., Mohr, M.: A hybrid approach for proving noninterference of Java programs. In: Fournet, C., Hicks, M.W., Viganò, L. (eds.) Computer Security Foundations Symposium (CSF), pp. 305–319. IEEE Computer Society (2015)
23. Le Guernic, G.: Information flow testing. In: Cervesato, Iliano (ed.) ASIAN 2007. LNCS, vol. 4846, pp. 33–47. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76929-3_4
24. Leavens, G.T., Kiniry, J.R., Poll, E.: A JML tutorial: modular specification and verification of functional behavior for Java. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, p. 37. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_6
25. Lortz, S., Mantel, H., Starostin, A., Bähr, T., Schneider, D., Weber, A.: Cassandra: towards a certifying app store for Android. In: ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), pp. 93–104. ACM (2014)
26. Milushev, D., Beck, W., Clarke, D.: Noninterference via symbolic execution. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE -2012. LNCS, vol. 7273, pp. 152–168. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30793-5_10
27. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliard, J.: Your proof fails? Testing helps to find the reason. In: Aichernig, B.K., Furia, C.A. (eds.) TAP 2016. LNCS, vol. 9762, pp. 130–150. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41135-4_8
28. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE J. Sel. Areas Commun.* **21**(1), 5–19 (2006)
29. Snelting, G.: Combining slicing and constraint solving for validation of measurement software. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 332–348. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61739-6_51
30. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.* **15**(4), 410–457 (2006)

31. Wasserrab, D., Lohner, D.: Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In: Aderhold, M., Autexier, S., Mantel, H. (eds.) Verification Workshop (VERIFY). EPiC Series in Computing, vol. 3, pp. 141–155 (2010)
32. Weiser, M.: Program slicing. In: International Conference on Software Engineering (ICSE), pp. 439–449. IEEE Press (1981)



Tactic Program-Based Testing and Bounded Verification in Isabelle/HOL

Chantal Keller^(✉)

LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay, UMR8623,
91405 Orsay, France
`Chantal.Keller@lri.fr`

Abstract. Program-based test-generation methods (also called “white-box” tests) are conventionally described in terms of a control flow graph and the generation of path conditions along the paths in this graph. In this paper, we present an alternative formalization based on state-exception monads that allows for direct derivations of path conditions from program presentations in them; the approach lends itself both for program-based testing procedures—designed to meet classical coverage criteria—and bounded verification. Our formalization is implemented in the Isabelle/HOL interactive theorem prover, where symbolic execution can be processed through tactics implementing test-generation strategies for various coverage criteria. The resulting environment is a major step towards testing support for the development of invariants and post-conditions in C verification environments similar to Isabelle/AutoCorres.

Keywords: White-box testing · Bounded verification
Symbolic execution · Coverage criteria · Interactive theorem proving

1 Introduction

In this paper, we present a range of program-based (“white-box”) test generation methods inside an interactive theorem prover. Conventional implementations [3, 7, 8] convert the abstract-syntax tree of the source program into a control flow graph (CFG for short) defining a set of paths, giving rise to various path-coverage criteria. In contrast, we base our work on a shallow embedding of programs in the state-exception monad. This presentation can be seen as a minimalistic imperative core language tuned to program verification. As a side-effect, our test-generation procedure meeting different coverage criteria is implemented by a semantically neutral annotation process combined with tactical decomposition based on derived rules; it is therefore a verified tool by construction.

The contributions are the following. First, we propose to perform a symbolic execution of programs using the semantic rules of a state-exception monad. Compared to conventional presentations, it provides a lightweight environment for white-box test generation (around 1500 LOC, including proofs). Second, we embed the process into the Isabelle/HOL proof assistant, to offer:

- a formal verification of symbolic execution, *via* the correctness of the state-exception monad rules;
- reasonably efficient automatic engines for various coverage criteria, *via* Isabelle’s support for term manipulation;
- bounded model checking, *via* the possibility to formally prove the validity of the abstract test cases.

Our paper proceeds as follows. After recalling the CFG-based approach for white-box testing on a running example, we detail our novel approach based on monads. We demonstrate the resulting symbolic execution rules by example, and explain their use both for bounded verification and for testing by injecting different forms of test-hypothesis. We conclude by tactics—implemented in the Isabelle/HOL interactive theorem prover—achieving various coverage criteria by construction, and illustrate the approach on a few examples.

All the material can be found at <https://www.lri.fr/~keller/TAP18>. For readability reasons, the definitions in the paper are written in Higher Order Logic (as defined by Church’s Simple Type Theory [5]), expressed in a ML-like language with pre-defined symbols such as implication (\implies), set comprehension ($\{- \mid -\}$), . . . , and a type constructor **theorem** that can be applied only to valid expressions (validity being proved interactively by Isabelle/HOL tactics). The notation

assumes H: P

shows Q

means **theorem** $P \implies Q$ (giving the name H to hypothesis P).

2 The Classical Approach to White-Box Testing

In order to contrast our approach to “the classical one”, we will briefly present the latter using a running example: an algorithm for computing the integer square root of an integer. We use a vanilla imperative language in order to represent our example program:

```

1  int squareroot(int a):
2  -- pre : 0 ≤ a
3  -- post: result2 ≤ a ∧ a < (result+1)2
4  { int tm = 1; int sqsum = 1; int i = 0;
5    while sqsum ≤ a {
6      i := i + 1;
7      tm := tm + 2;
8      sqsum := tm + sqsum;
9    };
10  return(i)
11 }
```

The **return** command assigns its argument to the implicitly declared return value **result** that is in the variable scope of the post-condition.

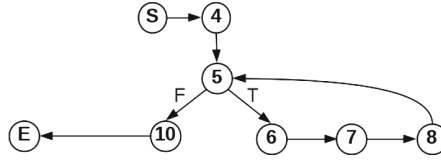


Fig. 1. The CFG for the integer squareroot program

The algorithm computes the sum of odd numbers and exploits the well-known fact:

$$\sum_{x=0}^{i-1} (2x + 1) = 1 + 3 + 5 + \dots + (2i - 1) = i^2$$

Thus, the impairs are accumulated in the variable `sqsum` to a series of squares $(i+1)^2$. If `sqsum` becomes larger than the input `a`, `i` must be its integer square-root.

In this article, we focus on program-based test methods, that make use of both the specification and the program itself to automatically build concrete tests in order to check if the program meets the specification on those tests, preferably given various coverage criteria.

The classical approach transforms our example program into a control-flow graph (CFG) as shown in Fig. 1. Except for the start-node `S` and the end-node `E`, the nodes are labeled with corresponding program line numbers; these nodes represent the state-set that is reachable after a program point. These state-sets can be characterized by formulas associated to nodes; for example: $4 \mapsto \{\sigma \mid \sigma.tm = 1\}$, $5 \mapsto \{\sigma \mid \sigma.tm = 1 \wedge \sigma.sqsum = 1\}$, etc; the notation $\sigma.tm$ standing for “the value of `tm` in state σ ”.

The node labeled 7 is a *decision node* where the left outgoing arc represents the computations where the evaluation of the condition yielded false (F), while the right outgoing arc represents the evaluation to true (T) leading to one more traversal of the loop.

On the basis of a CFG, the notion of an execution path can be established: the path `[S,4,5,10,E]` is the path that does not traverse the loop, `[S,4,5,6,7,8,5,10,E]` the path that traverses the loop exactly once, etc. The formula ϕ_π characterizing the set of states that will lead to an execution along a path π is called a *path condition*; for the case $\pi = [S, 4, 5, 10, E]$, for example, ϕ_π is $\sigma.a = 0$. Path conditions can be constructed automatically by a symbolic execution process (a variant thereof will be presented later in the paper); however, a path condition can be unsatisfiable reflecting the fact that it does not necessarily represent a computation that is actually possible. This may happen due to conflicting conditions in decision nodes, for example. Whenever ϕ_π is unsatisfiable, π is called *infeasible*.

While the set of execution paths is infinite whenever the program contains unbounded loops, CFG’s and the resulting notion of paths lend themselves naturally to *coverage criteria* which are fairly easy to understand and which found

their way into industrial applications and technical standards for software quality such as ISO 25119 [13]. For example a coverage criterion could constrain the set of all (if possible feasible) paths of a CFG to a path set covering all decision nodes (**allcond**), a path set covering all transitions (**alltrans**), or the set of paths that traverse all loops at most k times (**allpath_k**). In particular, variants based on **allpath₃** are used in many industrial development processes since they have empirically shown a reasonably good compromise between cost and error detection capacity.

3 Instead of CFG's: Symbolic Execution on Monads

3.1 Basic Definitions

We base this work on the Monad theory distributed with the HOL-TestGen testing framework [4]. This presentation is geared towards testing (see Sect. 8 for a comparison to other monad theories).

We define the monad-type as a transition function from a state of type σ to a successor state and some output of type $'\sigma$:

type $('\sigma, ' \sigma)$ $\text{MON}_{SE} = ' \sigma \rightarrow (' \sigma * ' \sigma)$ option

The composition operator on monads *bind* and the neutral element *unit* are standard:

let $\text{bind}_{SE} : (' \sigma, ' \sigma)$ $\text{MON}_{SE} \rightarrow (' \sigma \rightarrow (' \sigma, ' \sigma)$ $\text{MON}_{SE}) \rightarrow (' \sigma, ' \sigma)$ MON_{SE}
 $= \text{fun } f \text{ g} \rightarrow (\lambda \sigma. \text{ case } f \text{ } \sigma \text{ of None} \rightarrow \text{None} \mid \text{Some } (\text{out}, \sigma') \rightarrow \text{g out } \sigma')$

let $\text{unit}_{SE} : ' \sigma \rightarrow (' \sigma, ' \sigma)$ $\text{MON}_{SE} = \text{fun } e \rightarrow (\lambda \sigma. \text{Some}(e, \sigma))$

We will use alternative notations for the bind_{SE} combinator: the notation $\times \leftarrow f$; g stands for $\text{bind}_{SE} f$ (**fun** $\times \rightarrow \text{g}$) and f ; g for $_ \leftarrow f$; g . It is straightforward to prove the fundamental unit and associativity monad laws.

3.2 The Enriched Monad Infrastructure

As standard, in addition to these two basic blocks, it is convenient to declare common constructions used to define and manipulate monadic programs.

It is possible to add combinators for exception raising and handling and other usual programming constructs. We focus only on the conditional and the slightly more tricky case of loop definitions:

let $\text{if}_{SE} : (' \sigma \rightarrow \text{bool}) \rightarrow (' \sigma, ' \sigma)$ $\text{MON}_{SE} \rightarrow (' \sigma, ' \sigma)$ $\text{MON}_{SE} \rightarrow (' \sigma, ' \sigma)$ MON_{SE}
 $= \text{fun } c \text{ E } F \rightarrow (\lambda \sigma. \text{ if } c \text{ } \sigma \text{ then } E \text{ } \sigma \text{ else } F \text{ } \sigma)$

let $\text{while}_{SE} : (' \sigma \rightarrow \text{bool}) \rightarrow (\text{unit}, ' \sigma)$ $\text{MON}_{SE} \rightarrow (\text{unit}, ' \sigma)$ MON_{SE}
 $= \text{fun } c \text{ B} \rightarrow (\text{lfp } (\Gamma c \text{ B}))$

In the definition of while_{SE} , lfp is a least-fixpoint operator, and $\Gamma b \text{ cd}$ is a state relation defined by:

```

let  $\Gamma$  b cd : 'σ * 'σ
= fun cw → {(s,t) | if b s then (s,t) ∈ (cd O cw) else s = t}

```

where $_ \circ _$ is the relation composition (remind that $\{_ \mid _\}$ is set comprehension). Informally, it means that the initial and final states of a loop are equal if the condition is false, and related by one more application of the loop body otherwise.

The proof of the unfold theorem:

theorem while_SE_unfold:

```

(whileSE b do c od) = (ifSE b then c; -whileSE b do c od else return() fi)

```

is relatively complicated, but folklore (it is described in Winskell's book [18]; a first formal treatment in Isabelle we know of is [12]). Since our objective is to perform symbolic execution, the while operator can be decorated with a natural integer that limits the number of loops unrolling:

theorem while_n_unfold:

```

(while[Suc n] b do c od) = (ifSE b then c; -while[n] b do c od else return() fi)

```

where while[n] is defined as while_{SE} by ignoring the decoration n.

Note furthermore that we will embed **assume**_{SE} and **assert**_{SE} to model pre- and post-conditions, respectively. The construction **assume**_{SE} puts a program in a initial state that satisfies some predicate P (if such a state exists), and **assert**_{SE} checks if the final state of a program satisfies some predicate (by returning the program that succeeds if and only if its state satisfies P).

```

let assumeSE : ('σ → bool) → (unit, 'σ) MONSE
= fun P → (λσ. if ∃σ. P σ then Some((), SOME σ. P σ) else None)

```

```

let assertSE : ('σ → bool) → (bool, 'σ) MONSE
= fun P → (λσ. if P σ then Some(True,σ) else None)

```

Here, the construction SOME σ. P σ returns a state σ that satisfies P (note that it is guarded by the fact that P is satisfiable).

3.3 Symbolic Execution Rules for the Monad

Instead of the syntax-based concept “execution path”, we define the semantic concept of a *valid test-sequence* as a valid monad execution of a particular format: it consists of a series of monad computations $m_1 \dots m_n$ applied to inputs $i_1 \dots i_n$ and a post-condition P wrapped in a return depending on observed output. Validity is formally defined as follows:

```

let validSE : 'σ → (bool, 'σ) MONSE → bool"
= fun σ m → (m σ ≠ None) and fst(the (m σ))"

```

where the operator **the** is defined such that **the** (Some x) returns x (again, it is guarded by the fact that $m \sigma \neq \text{None}$). We will write $\sigma \models_m$ for **valid**_{SE} σ m. Since each individual computation m_i may fail, the concept of a valid test-sequence corresponds to a feasible path in a non-deterministic automaton, that

leads to a state in which the observed output satisfies P . Using the notation introduced in Sect. 3.1, we will write an entire sequence as follows:

$$\sigma \models_{\mathbf{o}_1} \leftarrow \mathbf{m}_1 \ i_1; \dots; \mathbf{o}_n \leftarrow \mathbf{m}_n \ i_n; \text{return}_{SE} (P \ \mathbf{o}_1 \dots \mathbf{o}_n)$$

Note that since the \mathbf{m}_i can be conditionals or a while loop, a sequence represents the stack of executions yet to be executed, and the \mathbf{o}_j the intermediate results stored on way (if any).

The notion of a valid test-sequence has two facets. On the one hand, it is executable, i.e., a *program*, if and only if $\mathbf{m}_1, \dots, \mathbf{m}_n, P$ are. Thus, a code generator can map a valid test-sequence statement to code. In particular, in Isabelle/HOL, depending on the configuration, the code generator can map the calls to the \mathbf{m}_i to Isabelle/HOL-defined operations or to external code, i.e., some code to be tested. On the other hand, and this is a major strength of this monadic approach, valid test-sequences can be treated by a standard and simple family of symbolic executions rules, characterized by the following schema (for all monadic operations \mathbf{m} of a system, which can be seen as its step-functions):

$$\sigma \models \text{return}_{SE} P = P \quad (1)$$

$$\mathbf{m} \ i \ \sigma = \text{None} \implies (\sigma \models \mathbf{s} \leftarrow \mathbf{m} \ i; \mathbf{m}' \ \mathbf{s}) = \text{False} \quad (2)$$

$$\mathbf{m} \ i \ \sigma = \text{Some}(\mathbf{b}, \sigma') \implies (\sigma \models \mathbf{s} \leftarrow \mathbf{m} \ i; \mathbf{m}' \ \mathbf{s}) = (\sigma' \models \mathbf{m}' \ \mathbf{b}) \quad (3)$$

$$(\sigma \models (\mathbf{if}_{SE} \mathbf{b} \ \text{then} \ \mathbf{c} \ \text{else} \ \mathbf{d} \ \text{fi}); -\mathbf{m}) = (\mathbf{b} \ \sigma \wedge \sigma \models \mathbf{c}; -\mathbf{m}) \vee (\neg \mathbf{b} \ \sigma \wedge \sigma \models \mathbf{d}; -\mathbf{m}) \quad (4)$$

$$\begin{aligned} (\sigma \models \text{while}[\text{Suc } n] \ \mathbf{b} \ \text{do} \ \mathbf{c} \ \text{od}; -\mathbf{m}) = \\ (\sigma \models (\mathbf{if}_{SE} \mathbf{b} \ \text{then} \ \mathbf{c}; -\text{while}[n] \ \mathbf{b} \ \text{do} \ \mathbf{c} \ \text{od} \ \text{else} \ \text{return}_{SE}(\text{fi})); -\mathbf{m}) \end{aligned} \quad (5)$$

$$(\sigma \models \text{assume}_{SE} P; -\mathbf{m}) = (\forall \sigma' \in \{\sigma' \mid P \ \sigma'\}. (\sigma' \models \mathbf{m})) \quad (6)$$

$$(\sigma \models \text{assert}_{SE} P; -\mathbf{m}) = (P \ \sigma \wedge (\sigma \models \mathbf{m})) \quad (7)$$

This kind of rules is usually specialized for concrete operations \mathbf{m} ; if they contain pre-conditions $C_{\mathbf{m}}$ (constraints on i and state), or conditions, this calculus will just accumulate them and construct a constraint system to be treated by a solver (see next section for an example).

A technical improvement specific to Isabelle is to use its meta-logic (which is based on an intuitionistic fragment of Higher Order Logic) instead of Isabelle/HOL connectives. It gives better performance compared to rewriting the rules presented above. For example, the conditional rule is expressed as case-splitting as follows, similar to a disjunction elimination rule:

$$\frac{\sigma \models \mathbf{if}_{SE} \mathbf{b} \ \text{then} \ \mathbf{c} \ \text{else} \ \mathbf{d} \ \text{fi}; -\mathbf{m} \quad \begin{array}{c} [\sigma \models \mathbf{c}; -\mathbf{m}, \mathbf{b} \ \sigma] \\ \vdots \\ Q \end{array} \quad \begin{array}{c} [\sigma \models \mathbf{d}; -\mathbf{m}, \neg \mathbf{b} \ \sigma] \\ \vdots \\ Q \end{array}}{Q} \quad (8)$$

4 Representing Programs and Symbolic Execution

We are ready to undertake the final steps to actually represent imperative programs as a symbolic evaluation problem (that will be used in bounded verification and testing scenarios later).

We will introduce a notation for the assignment, which is modeled to never fail in our core language:

let `assign` : $(\sigma \rightarrow \sigma) \rightarrow (\text{unit}, \sigma) \text{ MON}_{SE} = \mathbf{fun} \ f \ \sigma \rightarrow \text{Some} \ (\(), f \ \sigma)$

for which we derive the desired destruction rule:

$$\sigma \models \text{assign } f \ ;- \ m = f \ \sigma \models m$$

With respect to the representation of state, we follow the idea of Isabelle/SIMPL [14] to reuse records where the record fields represent the program variables. For our running example, this means that we define the state as:

```
type state = {tm   : int ,
              i    : int ,
              sqsum : int ,
              a    : int }
```

Note that the variable `a` could also be modeled as a parameter not represented in the state since it is not modified. As standard, from this record, one can generate the accessor functions `a`, `sqsum`, `i` and `tm`; update operations like $\sigma \langle \text{tm} := E \rangle$; and a memory-theory with rules like $\text{tm}(\sigma \langle \text{tm} := E \rangle) = E$ and $\text{sqsum}(\sigma \langle \text{tm} := E \rangle) = \text{sqsum } \sigma$. (In Isabelle, they are all generated automatically.) As standard, we extend the notation for updates to chains of updates such as

$$\sigma \langle \text{tm} := 1, \text{sqsum} := 1, i := 0 \rangle$$

where the rightmost “wins” when applied to the same record field. Note that the types of record fields can be arbitrary HOL types; here, we profit largely from our compact shallow embedding representation. Moreover, other memory-models could be used as well.

The right-hand side of assignments, assertions, and conditions in \mathbf{if}_{SE} and \mathbf{while}_{SE} are represented as state-transition functions or as state-to-bool predicates. However, we will use notations such as $\langle \text{sqsum} \leq a \rangle$ for $\mathbf{fun} \ \sigma \rightarrow (\text{sqsum } \sigma) \leq (a \ \sigma)$, i.e. $\langle _ \rangle$ represents a parser that applies any record field name to a bound variable (for the state) that is λ -abstracted at the topmost level. Similarly, $\langle i := i + 1 \rangle$ represents $\lambda \sigma. \ \sigma \langle i := (i \ \sigma + 1) \rangle$.

To semi-interactively perform symbolic execution of our programs under test, we state the pre- and post-conditions as Isabelle theorems (which will allow us to apply manually or systematically the correctness rules of the state-exception monad, as explained in the remaining of the article). Thus, we can represent our squareroot example program in the following format:

assumes `annotated_program`:
 $\sigma_0 \models \text{assume}_{SE} \ \langle 0 \leq a \rangle \ ;-$

```

⟨tm := 1⟩ ; -
⟨sqsum := 1⟩ ; -
⟨i := 0⟩ ; -
(whileSE ⟨sqsum ≤ a⟩ do
  ⟨i := i + 1⟩ ; -
  ⟨tm := tm + 2⟩ ; -
  ⟨sqsum := tm + sqsum⟩
od) ; -
assertSE(λσ. σ = σR)

```

shows $\sigma_R \models \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle$

Note that σ_R is a free variable in this goal denoting the “result state” after the execution of our squareroot program; it is the purpose of the entire assumption to construct this state (symbolically). However, in the conclusion, we require that the post-condition is to hold in this state which expresses our verification notion.

We will run a little simulation of our rule set in order to show how everything fits together; we will automate the entire process in subsequent sections, targeting different objectives.

Assume that we want to explore the program up to all paths of the depth 3. Then we rewrite `whileSE` by `while[Suc(Suc(Suc 0))]` and apply subsequently the destruction of `assumeSE` (Rule 6) and repeatedly the destruction of `assign` (Rule 7). This transforms our goal into:

```

∀σ. 0 ≤ a σ ⇒
  σ (⟨tm := 1, sqsum := 1, i := 0⟩) ⊨
    (while[Suc(Suc(Suc 0))] ⟨sqsum ≤ a⟩ do
      ⟨i := i + 1⟩ ; -
      ⟨tm := tm + 2⟩ ; -
      ⟨sqsum := tm + sqsum⟩
    od) ; -
  assertSE(λσ. σ = σR)

```

Further repetitive applications of destruction of `while[_]` and `ifSE` (Rules 4 and 5) as well as `assign` (Rule 7) leave us basically with a proof state of the following form:

1. $\forall \sigma. 9 \leq a \sigma \Rightarrow$
 $\sigma (\langle i := 3, tm := 7, sqsum := 16 \rangle) \models$
 $(\text{while}[0] \langle sqsum \leq a \rangle \text{ do}$
 $\quad \langle i := i+1 \rangle ; -$
 $\quad \langle tm := tm+2 \rangle ; -$
 $\quad \langle sqsum := tm + sqsum \rangle \text{ od}) ; -$
 $\text{assert}_{SE} (\lambda\sigma. \sigma = \sigma_R) \Rightarrow$
 $\sigma_R \models \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle$
2. $\forall \sigma. 4 \leq a \sigma \Rightarrow \neg 9 \leq a \sigma \Rightarrow$
 $\sigma_R = \sigma (\langle i:=2, tm:=5, sqsum:=9 \rangle) \Rightarrow$
 $\sigma (\langle i:=2, tm:=5, sqsum:=9 \rangle)$
 $\models \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle$
3. $\forall \sigma. 1 \leq a \sigma \Rightarrow \neg 4 \leq a \sigma \Rightarrow$

$$\begin{aligned}
& \sigma_R = \sigma(i := 1, tm := 3, sqsum := 4) \implies \\
& \sigma(i:=1, tm:=3, sqsum:=4) \\
& \models \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle \\
4. \forall \sigma. 0 \leq a \ \sigma \implies \neg 1 \leq a \ \sigma \implies \\
& \sigma_R = \sigma(tm:=1, sqsum:=1, i:=0) \implies \\
& \sigma(tm:=1, sqsum:=1, i:=0) \\
& \models \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle
\end{aligned}$$

This proof state contains now by construction:

- in the last sub-goal, the path condition for never entering the loop (essentially $a \ \sigma = 0$),
- in the third sub-goal, the path condition for entering the loop exactly once ($1 \leq a \ \sigma < 4$),
- in the second sub-goal, the path condition for entering the loop twice ($4 \leq a \ \sigma < 9$),
- in the first sub-goal, the path condition for traversing the loop more than twice ($9 \leq a \ \sigma$).

In the first sub-goal, the remaining `while [0]` represents the class of all possible remaining executions; therefore, for this class no elimination of the σ_R can be achieved via application of the one-point-rule.

Note that the decoration on `while []` allows one to unroll nested loops at different depths.

In the remaining of the paper, sub-goals containing assumptions of the form: $\sigma \models (\text{while [0]} \dots \text{do} \dots ; - \dots)$ will be called *incomplete*, and the others will be called *complete*. Obviously, the latter represent execution paths through the program where the resulting equation $\sigma_R = E(\sigma)$ bounds σ_R to the result of the symbolic execution.

5 Verification vs. Testing

At this stage, we have obtained the result of symbolic executions up to a certain depth. Once again, the embedding into the Isabelle/HOL proof assistant offers a lightweight framework to perform bounded verification or testing, or a combination of the two.

To this end, the ideas of the Isabelle/HOL-TestGen system [4] can be reused. HOL-TestGen exploits the concept that two types of hypotheses are used to express the differences between a proof of a property P and its test [9]:

- the *uniformity hypothesis* assumes that if a test passes for one instance of a partition of the input-output relation of a program specification P , then P will hold for the whole partition, and
- the *regularity hypothesis* assumes that if a test passes with sufficiently “deep” or “complex” input data for P , then P will always hold.

HOL-TestGen generates from test specifications of the format:

$$\text{pre } x \rightarrow \text{post } x \text{ (PUT } x)$$

for the (black-box) program under test PUT, which is logically just an uninterpreted constant, a partitioning of the input-output relation, and *explicit* test hypotheses which were added to test-property on the fly as a consequence of the targeted test-criterion. In a last step, HOL-TestGen generates test-drivers—basically test-oracles from post-conditions—that are linked to the actual code of PUT. Note again that it is not necessary to construct an invariant in a white-box testing approach, the precondition for filtering illegal input and the post-condition for generating oracles suffices.

In this section, we explain how these ideas apply in our setting. For the engineering part consisting in transforming goals into test hypotheses and test cases, we refer the reader to the description of the HOL-TestGen system [4].

5.1 Strategy: Bounded Verification (Also Called Bounded Model-Checking)

If we adopt this overall concept to white-box testing, we need only to find an equivalent to the regularity hypothesis. The strategy for bounded verification consists of:

- attempting to prove the *complete goals* automatically. For the case 2) in squarerooot, for example, this boils down to proving:

$$\begin{aligned} 4 \leq a \ \sigma \implies \neg 9 \leq a \ \sigma \implies \\ \sigma (|i:=2, tm:=5, sqsum:=9) \models \\ \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle \end{aligned}$$

which falls into a fragment decided by many automated provers. (In Isabelle, this goal is automatically discharged by the `auto` command.)

- admitting the *incomplete goals*. They would require an invariant for their proofs. Rather than attempting to prove them, we turn them into an explicit test hypothesis of the form:

$$\begin{aligned} \text{THYP}(\forall \sigma. 9 \leq a \ \sigma \rightarrow \\ \sigma (|i := 3, tm := 7, sqsum := 16) \models \\ (\text{while}_{SE} \langle sqsum \leq a \rangle \text{ do} \\ \quad \langle i := i+1 \rangle ; - \\ \quad \langle tm := tm+2 \rangle ; - \\ \quad \langle sqsum := tm + sqsum \rangle \text{ od}) ; - \\ \text{assert}_{SE} (\langle i^2 \leq a \wedge a < (i+1)^2 \rangle)) \end{aligned}$$

where `THYP(x)≡x` just serves as a semantically neutral marker to control the tactic process. This form of explicit test-hypothesis states “beyond our analysis depth, we assume that the program is correct” and represents a regularity hypothesis adapted to program-based testing.

Adding the explicit regularity hypothesis to the assumptions of the original goal (thus weakening it logically) allows for a formal proof of the modified goal making explicit under which assumptions our program (model) satisfies the specification.

5.2 Strategy: White-Box Testing

Testing differs from bounded verification in basically two ways: first, we use additionally the uniformity hypothesis, stating that for each partition of the input-output relation (i.e. the path conditions), we assume that the program is correct provided that we found one instance in this partition where it behaves correctly (in ISO 25119 [13], this assumption is used for what is called “equivalence class testing”). Second, the concrete instance of a partition, called concrete test-case and usually constructed by a constraint solver, can be converted into test driver code that is run against the real program, not just a model of it. This turns testing into a validation method that covers also hardware, the underlying operating system, the compiler, etc, of the program under test. Therefore, evaluators in formal evaluation schemes like Common Criteria insist on tests *validating* a (code) model against “the real thing”; verifications based on immanent arguments over models are acceptable as complements, but not as a complete replacement of tests.

In our running example, the uniformity test-hypothesis will be, for example:

$$\begin{aligned} \text{THYP}((\exists \sigma. 4 \leq a \ \sigma \wedge \neg 9 \leq a \ \sigma \wedge \\ \sigma \models \text{squareroot} \ ; \text{-} \ \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle) \rightarrow \\ (\forall \sigma. 4 \leq a \ \sigma \implies \neg 9 \leq a \ \sigma \rightarrow \\ \sigma \models \text{squareroot} \ ; \text{-} \ \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle)) \end{aligned}$$

where `squareroot` is an abbreviation for our program code.

A constraint solver might find the solution $a \ \sigma = 7$ for the path-condition above, thus permitting us to construct automatically from the uniformity hypothesis the concrete test:

$$\sigma (a := 7) \models \text{squareroot} \ ; \text{-} \ \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle$$

6 Support for Coverage Criteria

In the end of Sect. 4, we detailed how a manual application of the rules of the state-exception monad (via Isabelle basic tactics) perform step-by-step symbolic execution. In this section, we explain how to build new tactics that automate the process, with support for various coverage criteria.

These tactics are based on the following observations. Repeated applications of the rule for symbolic execution of `ifSE` (Rule 4) guarantee *branch coverage*, since each branch of each control structure is covered. In our small imperative language, this is equivalent to *decision coverage*, since we do not handle function calls yet [15]. Similarly, repeated applications of the rule for `whileSE` loops (Rule 5) immediately followed by the rule for `ifSE` guarantee loop coverage up to a certain depth.

The Isabelle tactical language, called Eisbach [11], allows us to design various tactics that apply and combine the symbolic execution rules, together with simplifications of the goal:

- the tactic `branch_and_loop_coverage` simply relies on the two coverage criteria described above;
- the tactic `mcdc_and_loop_coverage` covers more: it also performs *Modified Condition/Decision Coverage* (MC/DC for short), meaning that each condition (i.e. Boolean sub-expression) appearing in a decision affects the decision outcome independently;
- conversely, the tactic `loop_coverage_positive_branch` covers less: it performs loop coverage but, for branches, always chooses the first branch (other choices, such as random, could as well be implemented). This may be useful if one wants to explore loop unrolling without a combinatorial explosion (see e.g. Sect. 7.1).

For instance, applied to the annotated program of Sect. 4, the tactic

```
apply (branch_and_loop_coverage "Suc (Suc (Suc 0))")
```

directly leads to the proof state presented in the end of the section.

The Eisbach tactical language defines new tactics by combining existing ones using the syntax of regular expression. For instance, the first two tactics are programmed from a single parametric method defined as follows:

```
method loop_coverage for n::nat methods simp_mid simp_end =
  (bound_while n)?, loop_coverage_steps simp_mid, simp_end?
```

The syntax means the following. `loop_coverage` is the name of the tactic, parameterized by a natural number `n` and two other tactics named `simp_mid` and `simp_end`. This tactic sequentially performs:

1. `(bound_while n)?`: every occurrence (if there is any, represented by the `?`) of `whileSE` is replaced by `while[n]`, justified by the definition of `while[n]`.
2. `loop_coverage_steps simp_mid`: the equations of Sect. 4 are repeatedly applied: the auxiliary tactic `loop_coverage_steps` is a simple loop that, as much as possible, applies those equations, and simplifies intermediate results by the (abstract) tactic `simp_mid`. Note that, at each step, at most one equation can be applied, depending on the first instruction remaining in the program.
3. `simp_end?`: if possible, the result is simplified by the (abstract) tactic `simp_end`.

The choice of the simplifications leads to the first two variants.

- In the case of branch coverage, only basic simplifications are performed:

```
method branch_and_loop_coverage for n::nat =
  loop_coverage n memory_simp simp_all
```

where `memory_simp` is the memory-theory presented in Sect. 4 and `simp_all` is the standard Isabelle simplifier.

- As for MC/DC, simplifications should also split conjunctive and disjunctive hypotheses according to their elimination rules. To this end, we can use the Isabelle `auto` tactic¹:

¹ A specific tactic that only calls the simplifier and applies elimination rules of connectives would work as well and be less powerful.

method `mcdc_and_loop_coverage` **for** `n::nat = loop_coverage n auto auto`

The third tactic `loop_coverage_positive_branch` is similar but the rule for `ifSE` is applied only after loop unrollings. Other branches are symbolically executed using a weaker rule:

$$(\sigma \models (\text{if}_{SE} b \text{ then } c \text{ else } d \text{ fi}); -m) = (b \sigma \wedge \sigma \models c; -m) \vee (\text{opaque}(\neg b \sigma \wedge \sigma \models d; -m))$$

where `opaque` is a tag that forbids further application of rules.

These prototype tactics are already reasonably efficient: it takes less than 30 s to unroll the loop 100 times with MC/DC on our running example, searching counter-examples up to 10000.

7 Examples

This section illustrates the flexibility and expressivity of the monadic approach (Sect. 7.1) and the tactics we just presented (Subsects. 7.1 and 7.2). These examples and more have been implemented in Isabelle/HOL and are fully automated using the tactics; they can be found in the online material.

7.1 Maximum of an Array

Symbolic execution of programs manipulating usual data types can be faithfully performed using standard functional representation. We give here the example of a function computing the maximum of an array: the array can be represented as a function whose domain is non-negative integers together with a length. The state thus contains

```
type state = {arr   : nat → int,   (* The array is represented as a function ... *),
              l     : nat,       (* ... and a length *)
              i     : nat,       (* The loop index *)
              res   : int        (* The result *)
            }
```

and the program is the usual one:

assumes `annotated_program`:

```
σ0 ⊨ assumeSE ⟨1 ≤ l⟩ ; -
  ⟨res := arr 0⟩ ; -
  ⟨i := 1⟩ ; -
  (whileSE ⟨i < l⟩ do
    (ifSE ⟨res < arr i⟩ then ⟨res := arr i⟩ else skipSE fi) ; -
    ⟨i := i + 1⟩
  od) ; -
  assertSE(λσ. σ = σR)
```

shows `σR ⊨ assertSE ⟨(∀ k < l. res ≥ arr k) ∧ (∃ k < l. res = arr k)⟩`

The post-condition is also standard, stating that the result is greater or equal than all the elements of the array, and equal to one of them, under the pre-condition that the length is at least 1. Such a property is well-suited for testing or bounded checking.

On this example, full bounded symbolic execution presented in this article (and performed by the `branch_and_loop_coverage` and `mcdc_and_loop_coverage` tactics) unrolls the loop a given amount of time and explores both branches inside the loop, leading to an exponential blow-up where the maximum could be anywhere in the array. Under the regularity hypothesis though, it may be sufficient to test only one or a few possible cases for the maximum at each length, which is obtained by the tactic `loop_coverage_positive_branch` (or any variant that executes only one branch at each step). We refer the reader to the online material for an executable comparison.

In any case, this example actually generates abstract test cases that precisely determinate the position of the maximum of the array.

Other standard data-structures can be modeled such as hash tables, or lists (using Isabelle/HOL lists).

7.2 Median of Three Integers

To illustrate MC/DC vs. branch coverage, we take the example of a program computing the median of three integers:

assumes `annotated_program`:

$$\begin{aligned} \sigma_0 \models & (\text{if}_{SE} \langle (b \leq a \wedge a \leq c) \vee (c \leq a \wedge a \leq b) \rangle \text{ then } \langle \text{res} := a \rangle \text{ else} \\ & (\text{if}_{SE} \langle (a \leq b \wedge b \leq c) \vee (c \leq b \wedge b \leq a) \rangle \text{ then } \langle \text{res} := b \rangle \text{ else} \\ & \langle \text{res} := c \rangle \text{ fi}) \text{ fi} ; - \\ & \text{assert}_{SE} (\lambda \sigma. \sigma = \sigma_R) \end{aligned}$$

shows $\sigma_R \models \text{assert}_{SE} \langle (\text{res} = a \vee \text{res} = b \vee \text{res} = c) \wedge (\text{res} > a \longrightarrow \text{res} \leq b \wedge \text{res} \leq c) \rangle$

The post-condition only consider one case but can be completed by adding the second conjunct for every permutations of `a`, `b` and `c`.

The `branch_and_loop_coverage` tactic generates only three abstract test cases whose premises are:

- $(b \leq a \wedge a \leq c) \vee (c \leq a \wedge a \leq b)$
- $(a \leq b \wedge b \leq c) \vee (c \leq b \wedge b \leq a)$
- the negation of the first two

whereas, as required, the `mcdc_and_loop_coverage` tactic generates six abstract test cases corresponding to the possible modified conditions, of the shape $(b \leq a \wedge a \leq c)$ for every permutation of `a`, `b` and `c`.

8 Related Work

Using monads is a standard technique for representing stateful computations. Leaving aside existing implementations in HOL4 and Coq, the

Isabelle/HOL library alone defines a standardized monad-syntax for both a non-deterministic (Kleisli-like) and a deterministic monad. Other libraries include Isabelle/Simpl [14], Isabelle/ORCA [2] and Isabelle/AutoCorres [10]. These infra-structures are geared towards program-proofs or program refinement proofs, include `bool's` to capture termination, and, in the case of `Imperative.HOL`, a very specific heap-memory model used in the AutoCorres Tool for verifying C Programs. Besides having improved syntax support, the used monad here is geared to pure partial program semantics and optimized forms of partial evaluation therein. In particular, it is agnostic to a particular memory model. The presented loop-unfold theorem is not available in neither of mentioned monad theories.

Generating tests by counterexample generators is an active research area. There are basically two approaches. One is to take an input formula, try to construct a family of finite models, usually by bit-blasting into SAT problems, and to construct a counter-example on this basis [1, 16]. The other one is to interpret the input formula as a filter, i.e. to compile it to program for a Boolean function, and stimulate it by random values until a hit is found. This concept going back to [6] is known as QuickCheck and leads to a wealth of implementations for various languages meanwhile. Both approaches suffer from their generality when it comes to the generation of counterexamples for *programs* with pre-conditions. Moreover, they cannot compete with white-box testers with respect to the depth of program exploration as well as the coverage of given criteria imposed by standards such as [13].

As currently most developed white-box testers we mention Pex [7] and Path-Crawler [17]. They present direct algorithmic implementations working on CFG's and scale well for realistic sizes of programs. In contrast, our approach is based on a shallow representation of a semantics and derived rules. It enjoys the following two advantages:

- the code is very small (around a 1500 LOC, including proofs of correctness) but first experiments show that it is reasonably efficient (Sect. 6);
- our implementation is based on derived rules and can therefore guarantee correctness.

Regarding expressivity, the programming language handled by our approach is Turing complete; we leave for future work to handle convenient paradigms such as local states (e.g. by replacing the memory model by a stack), function calls and recursion.

The aforementioned state-of-the-art testers have been combined with techniques for borderline analysis, regular expression constraint-solvers and test-execution environments supporting virtual system calls. The approach presented in the paper is being integrated in the HOL-TestGen² [4] framework, to make use of its concrete test generator and code extraction. In addition, it explicitly constructs the test hypotheses.

² See <https://www.brucker.ch/projects/hol-testgen> for more details, in particular the TestSequence theory.

9 Conclusion

We have shown an approach to model white-box testing of block-structured imperative programs. We used a shallow, monad-style embedding of the language. We believe this allows for a particularly concise and elegant formalization of the symbolic execution process, which is traditionally described on a control-flow graph: the trick is done by just eight rules with little deductive cost (first-order matching). We have shown that the process can be easily wrapped up in a tactic process.

The approach lends itself to precisely study the borderlines between deductive verification, bounded verification and testing in a uniform setting. By re-using HOL-TestGen's concept of explicit test-hypothesis, the approach allows us to establish a precise link between test and proof.

It was not our objective to develop in this paper a full-blown tool (for that, we would have to integrate it into, say, Isabelle/SIMPL which necessitates to cope with much more features and machinery). Still, the shown experiments indicate that our approach does scale fairly well. Therefore, we believe that our technique has the potential for a tool that effectively tests pre- and post-conditions as well as invariants for realistic program verification attempts.

Acknowledgments. The author would like to thank Burkhart Wolff for setting up the foundations of this work. She also thanks the anonymous reviewers for valuable and detailed comments on how to improve the article.

References

1. Blanchette, J.C., Nipkow, T.: Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_11
2. Bockenek, J.A.: An extension of Isabelle/UTP with simpl-like control flow. Ph.D. thesis, Virginia Polytechnic Institute and State University (2017)
3. Botella, B., Delahaye, M., Ha, S.H.T., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: experience with pathcrawler. In: Proceedings of the 4th International Workshop on Automation of Software Test, AST 2009, Vancouver, BC, Canada, 18–19 May 2009, pp. 70–78 (2009)
4. Brucker, A.D., Wolff, B.: On theorem prover-based testing. *Formal Asp. Comput. (FAOC)* **25**(5), 683–721 (2013)
5. Church, A.: A set of postulates for the foundation of logic (1). *Ann. Math.* (1932)
6. Claessen, K., Hughes, J.: Testing monadic code with quickcheck. *SIGPLAN Not.* **37**(12), 47–59 (2002)
7. de Halleux, J., Tillmann, N.: Parameterized unit testing with pex. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 171–181. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_12
8. Filieri, A., Pasareanu, C.S., Visser, W.: Reliability analysis in symbolic pathfinder: a brief summary. In: Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25–28 Februar 2014, Kiel, Deutschland, pp. 39–40 (2014)

9. Gaudel, M.-C.: Testing can be formal, too. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (eds.) CAAP 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59293-8_188
10. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don’t sweat the small stuff: formal verification of C code without the pain. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, UK, June 2014, pp. 429–439. ACM (2014)
11. Matichuk, D., Wenzel, M., Murray, T.: The Eisbach user manual. Isabelle Commun. (2015)
12. Nipkow, T.: Winskel is (almost) right: towards a mechanized semantics textbook. *Formal Aspects Comput.* **10**, 171–186 (1998)
13. Working Group (WG26) of the ISO/IEC JTC1/SC7 Software and Software Engineering Committee. ISO/IEC/IEEE 29119 Software Testing: The International Standard for Software Testing (2007–2014)
14. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
15. FCAS Team et al.: What is a “decision” in application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC). Technical report position paper (2002)
16. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
17. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 281–292. Springer, Heidelberg (2005). https://doi.org/10.1007/11408901_21
18. Winskel, G.: *The Formal Semantics of Programming Languages*. MIT Press, Cambridge (1993)



Verification Coverage for Combining Test and Proof

Viet Hoang Le^{1(✉)}, Loïc Correnson², Julien Signoles², and Virginie Wiels³

¹ CEA MiPy, Labège, France

viethoang.le@cea.fr

² CEA LIST, Software Reliability and Security Laboratory, Gif-sur-Yvette, France

{loic.correnson,julien.signoles}@cea.fr

³ ONERA, Palaiseau, France

Virginie.Wiels@onera.fr

Abstract. The V&V practices of safety-critical industries (e.g. avionics) are currently based on either unit testing or unit proof to verify that a function satisfies its low-level requirements in order to be compliant with the highest certification levels [26] (e.g. DO-178C level A for avionics software). In this context, the verification engineer must assess sufficient coverage of both code (structural coverage) and specification (functional coverage). However, there is no shared method for test and proof to measure structural coverage. In practice, this prevents the verification engineer from combining test and automatic proof to verify low-level requirements of a common piece of code in order to mitigate the verification cost. This paper fills this gap between test and proof by introducing a new notion of verification coverage based on mutation coverage. It subsumes functional coverage and structural coverage for both unit testing and unit proof. Consequently, it allows the verification engineer to mix test tools and automatic provers in the verification process for the sake of reducing verification cost, in the sense that the more automation is used during the verification, the less resource is spent to verify the program.

Keywords: Coverage criteria · Combining test and proof

1 Introduction

In software development of critical systems, the code verification step is crucial since it prevents unexpected behaviors to arise during program execution. In particular, the verification engineers must ensure that the program satisfies its specifications. For this purpose, testing is the most commonly used technique to reach the expected level of confidence. It consists in running the tested program on some input data and comparing the expected results according to the given *oracles derived from the program specifications* [28, 32]. Program proof is another suitable verification technique [9]. It consists in statically verifying the program with respect to its specifications for all possible executions by means of logical reasoning [17–19].

Whatever the underlying technique, one must ensure that enough verification has been performed. This part of the verification process is usually performed by measuring *coverage*, e.g. test coverage. For this purpose, the DO-178 standard for avionic software [29] introduces two processes: functional analysis and structural analysis. The former guarantees that all the program specifications are verified, while the latter guarantees that every path and every piece of code in the program is reached and contributes to producing the expected results.

Functional analysis is independent from the verification technique. The verification of a function achieves full functional coverage as soon as one succeeds to *test* or *prove* all the specifications [12, 13], thus all program specifications are verified. Structural analysis depends on the underlying verification technique. For testing, it relies on various structural coverage criteria (statement coverage, branch coverage, MC/DC coverage, *etc.*) [2] to ensure that executing a test suite covers each path and/or piece of code in the right way. For program proof, structural analysis may be performed by fulfilling different objectives. For instance, in DO-333 [30], one must ensure four objectives [10, 26]: assumption coverage (each proof assumption is checked); completeness (the specifications specify outputs for every input condition and, conversely, input conditions for every output), data-flow (all the dependencies between inputs and outputs are found) and extraneous code (every piece of code depends on at least one specification). By guaranteeing their objectives, we ensure that neither path nor piece of code contributes to producing a result unexpected in any specification existed.

This current workflow has a major limitation: while test coverage on the one hand and proof coverage on the other hand are well-known concepts, it is not possible to use test coverage for proof and conversely. Test coverage can only be used when testing the entire program with the oracle derived from program specification, while proof coverage is only defined when all the program specifications are proved. Consequently, for a particular piece of code, it is not possible to test some specifications while proving the others, because there is no way to define the coverage of the combined verification.

Nowadays, during a proof campaign, the engineer relies on automated theorem provers in the hope to prove all program specifications and to ensure all objectives defined in DO-333 [9]. Usually most proof obligations are automatically discharged, but sometimes a few of them might not. In such a case, on account of the above-mentioned limitation, the engineer may either manually prove them and verify coverage through DO-333, or discard the proof campaign and rely on testing as defined in DO-178. In both cases, the verification process is much more expensive because it requires a lot of additional manual work.

This paper presents a new notion of verification coverage which aims at reducing the verification cost by keeping the existing proofs and adding only the necessary tests to complete the verification process. It subsumes functional coverage and structural coverage for both test and proof. It also relies on a *new notion of witness* that formalizes a verification activity and allows the verification engineer to sum up which verification technique has been used to validate that a particular piece of code contributes to enforcing some specifications. Furthermore, *we*

introduce a methodology and a companion algorithm that allow the verification engineer to check whether a verification campaign is complete with respect to this coverage.

The remainder of our paper is organized as follows. First, Sect. 2 discusses related work. Then, Sect. 3 introduces a running example. Section 4 presents the general idea of our work, which is then formalized in Sect. 5. Section 6 explains how to automatize as much as possible a verification campaign with respect to our new verification coverage. Finally, Sect. 7 exemplifies our process on the running example.

2 Related Work and Discussion

As previously mentioned, coverage is a major obstacle for combining test and proof. Typically, it prevents us from complete a partial proof campaign by means of testing. Several existing works already study different kinds of combinations of testing and formal verification techniques [3, 8, 21, 33], but they do not deal with the coverage issue. According to Bishop et al. [8], these combinations can be divided into four levels described below. We aim at defining a notion of coverage for the last two ones:

- Level 1 (Separately): test and proof are applied separately to verify different parts of the system;
- Level 2 (Assistance): proof supports test or conversely;
- Level 3 (Friendship): proof contributes to the automated generation of tests and their results are combined;
- Level 4 (Unification): test and proof are fully combined.

Our notion of coverage is based on existing notions of *label coverage* and *mutation coverage*. Label coverage [5, 6] relies on *labels*. Labels are logical formulae attached at program points. They can encode most structural coverage criteria. Originally, they were used to automatically generate test suites that satisfy a given structural criterion. Then, in [4], their usage has been extended in order to detect unfeasible labels when combining program proof and abstract interpretation. However, in these works, formal methods were only used for supporting testing (level 2 above). In particular, structural coverage was still limited to testing and cannot be applied to program proof. More recently, labels have been extended to hyperlabels [23, 24] to enlarge the variety of criteria that can be represented. We believe that our technique can be extended to hyperlabels, but we leave this to future work.

Our work also relies on mutants (in the sense of [16, 25]) to check our coverage metrics. A mutant m of a program p is a program obtained by slightly modifying p . Mutation testing consists in verifying that the outputs for p and m differ. In that case, one says that the mutant m is killed. Mutation coverage is defined by the number of killed mutants. Our work extends the usage of mutants to program proof. Many different mutation schemes exist [1, 27] for various programming languages. Our work relies on statement deletion to create mutants inspired by

Delamaro et al. [15]. But we also use other mutation operators like expression modification when it is more beneficial than statement deletion. Our proposed methodology is indeed independent from the underlying mutation schemes, but they may lead to different results: the choice of the best mutation scheme for a particular use case is let to the end-user. Mutation has also been explored to propose a notion of coverage for model checking [11]. The model is mutated and the model part is considered covered if the mutant survives. Their notion of coverage does not apply to testing. However, it inspired our more general notion of verification coverage.

3 Running Example

This section introduces a running example that illustrates current issues when combining test and proof in order to verify a C function in the context of a DO-178 certification.

Figure 1 presents a scheme of a function `transform`. The complete C code is omitted for the sake of brevity. This function is typical of reactive embedded software: it computes an output signal from an input by a linear regression depicted in Fig. 2. Here, all the values are bounded by an upper bound s_{\max} and a lower bound s_{\min} . In C program, these bounds are global variables. The linear regression also depends on parameters x_1, x_2, y_1 and y_2 that must satisfy the following consistency constraints:

$$s_{\min} \leq x_1 < x_2 \leq s_{\max} \quad \text{and} \quad s_{\min} \leq y_1 \leq s_{\max} \quad \text{and} \quad s_{\min} \leq y_2 \leq s_{\max}.$$

Signals are implemented by a structure named `Signal`. Each `Signal` contains a floating-point value in interval $[s_{\min}, s_{\max}]$, and an error flag indicating whether the constraints are satisfied. Parameters x_i and y_i ($i = 1, 2$) are passed to the C function via another structure named `Block`. Furthermore, the function returns 0 when the constraints are satisfied and an error code otherwise.

```

1 int transform (Block *p, Signal *input, Signal *output){
2 // 1. verify block validity
3 // 2. modify the value of the output signal w.r.t. the input signal
4 // 3. set the error flag
5 }
```

Fig. 1. Scheme of function `transform`.

The verification objectives for this function are the informal requirements defined in the spirit of DO-178. They may be formalized and split in four groups of specification:

- ERR** 7 specifications defining the error code in case of invalid parameters.
- OK** 1 specification formalizing the result when the constraints are satisfied.
- VALUE** 3 specifications defining the expected value of the output signal.
- VALID** 1 specification controlling the error flag of the output signal.

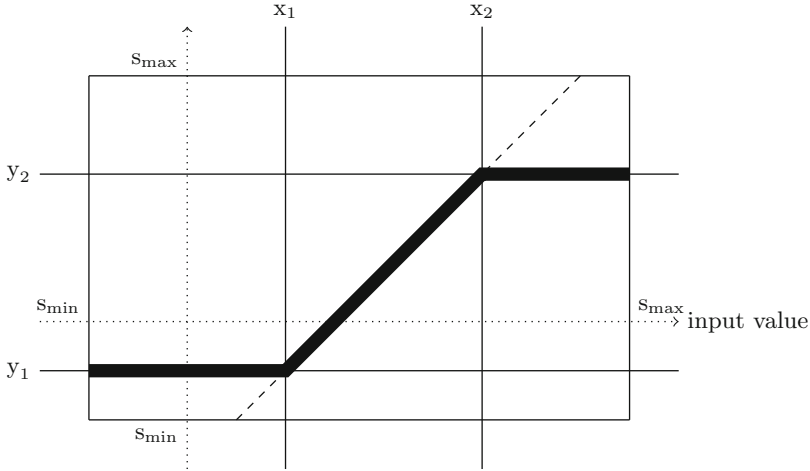


Fig. 2. Linear regression for transformation.

The function has been implemented in C and formally specified in the ACSL specification language [7]. Then, we tried to verify this code with Frama-C [20]. This framework has already been (successfully employed) for experimenting combinations of test and proof of C programs [22]. Here, E-ACSL [31], the runtime verification plug-in of Frama-C, is first run to check that the C code satisfies its formal ACSL specifications. For this purpose, we need to manually define at least 10 test cases, 7 among them for testing all situations when the constraints are invalid and 3 other used to test the expected value of output signal when the constraints are valid, to cover all the possible cases (or even more, depending on the chosen structural coverage criterion). Then, WP, the Frama-C plug-in for deductive verification, was used to (automatically) prove this function. Yet, one specification in group VALUE remained unproved because of floating-point computations in the code. Consequently, neither test nor proof alone allows us to complete the verification process with reasonable resource spent.

Careful code review allows us to argue that only a limited piece of code contributed to the unproved properties. Therefore, the intuitive goal of our approach is to complete the obtained proofs by adding only a few test cases that cover the remaining code fragments and the unproved specifications. We also aim at defining a notion of verification coverage for this use case.

4 Verification Campaign

This section provides additional details about the new kind of verification campaigns combining test and proof that we propose.

4.1 Labeled Mutant

Our technique is independent from a particular structural coverage criterion by relying on a notion of *labels* which can be used to encode most structural coverage criteria [6]. Each label is a property associated to a program point and divides the code source in two pieces that the below one is the corresponding code fragment of label. A label is satisfied when a verification activity demonstrates that there is an execution passing through this label and satisfying its associated property. A structural coverage criterion holds if and only if all the labels for this criterion are satisfied.

In order to gain additional results during a verification campaign, each label is associated to a mutant that modifies the corresponding code fragment. Such a mutant is named *labeled mutant*. Any method of mutation is possible whenever it fulfills the following condition: *the mutant shall only modify the executions that pass the label*.

In this paper, two kinds of mutation operators are used to generate labeled mutants: the *replace* and *erase* operators. The former replaces a statement by another one, while the latter removes it. Their formal definitions (omitted here) inspired by [1] and fulfill the above-mentioned condition. Mainly, they consist of introducing a conditional statement guarded by the labeled property. Figure 3 provides an example of such mutations for a small piece of code from our running example.

<pre> 1 2 if(p->y2 < smin){ 3 // label Lreturn6: 4 // label condition: true; 5 // labeled mutant: replace(0) ; 6 7 // initial statement: return (-6); 8 9 // mutant created before simplify: 10 // if (1) 11 // then {return 0;} 12 // else {return -6;} 13 14 // mutant after simplify: 15 return 1 ? 0 : -6; 16 } 17 </pre>	<pre> 1 2 if(x->v >= p->x2){ 3 // label Lstmt2: 4 // label condition: true; 5 // labeled mutant: erase; 6 7 // initial statement: y->v = p->y2; 8 9 // mutant created before simplify: 10 // if (1) 11 // then {} 12 // else {y->v = p->y2;} 13 14 // mutant after simplify: 15 if (! (1)) {y->v = p->y2;} 16 } 17 </pre>
(a) Replace	(b) Erase

Fig. 3. Example of labeled mutant.

Mutants are actually created for trying to kill them. Indeed, killing a mutant means that the corresponding statement in the initial program was meaningful for the checked criterion. Our method consists in finding out specifications that are validated in the initial program but not any more in a mutant (i.e. the mutant is killed by that specification) through various verification activities (inspired by the definition of mutation testing [16,25]). It allows us to conclude that the

mutated piece of code, denoted by a label, has a strong connection with these specifications. Full coverage is therefore achieved by establishing such a strong connection for each specification and piece of code corresponding to each label.

4.2 Verification Campaign

Our verification campaign assumes the existence of the source code, its specifications and labels encoding a particular structural coverage criterion. These labels may actually be automatically generated [5]. A verification campaign consists in a sequence of *verification activities* (either test or proof). Each of them provides two pieces of information: *verification information* indicating which specifications S is validated and *coverage information* indicating which labels L are covered. These pieces of information are grouped together through the notion of witness, as informally explained below.

Proof Activities. Automated deductive verification may provide *verification information* about the validity or the invalidity of each specification S . In our context, invalidity of S means validity of its negation $\neg S$ ¹. If S is validated in the initial code, it means that no execution passing any label contradicts the specification. This information is recorded through a *proof witness* between this specification and every label. However, if S is validated/invalidated in a labeled mutant (of label L), it means that S and L have no correlation at all (i.e. label L is associated to a piece of code that no matter how we modify this piece, specification S is still proved)/are strongly connected.

Consider our running example in which we successfully prove a specification of ERR named `error6`. Therefore, for each label L , one proof witness is recorded between `error6` and L . Furthermore, the specification is still proved in labeled mutant of label `Lstmt2` (Fig. 3b) but it is invalidated in labeled mutants of label `Lreturn6` (Fig. 3a). Thus, we conclude that there is no correlation between specification `error6` and label `Lstmt2`, while this specification and label `Lreturn6` are strongly connected.

Test Activities. Testing a specification S requires to manually define test cases. After defining them, testing the specification in source code and in mutants is an automatic process that provides us with *test witnesses* between labels and specifications. These witnesses are more precise than proof witnesses, since they assess that the corresponding label is reached.

In our running example, a particular test case activates the specification `error6` (its assumptions are fulfilled). It also covers several statements, one of them being the statement (a code fragment) associated to label `Lreturn6`. Yet, the statement associated to label `Lstmt2` is uncovered. Therefore, there is a test witness for the pair (`error6`, `Lreturn6`), but none for the pair (`error6`, `Lstmt2`).

¹ Usually, one only tries to prove S . Here, one tries to prove both S and $\neg S$ in order to get additional coverage information as explained later.

Error Detection. If a specification S is invalidated in the original source code as a result of a verification activity, we get an error which is recorded as *error witness* between S and all the existing labels.

Coverage Analysis. A specification S and a label L are strongly connected if and only if there is one verification witness that validates the pair (S, L) in the original source code and one witness that invalidates this pair in the corresponding mutant. All the strong connections between specifications and labels are stored in a *coverage matrix*. Its columns represent specifications, while its rows represent labels. Fulfilling our verification coverage means positively filling this matrix. Indeed, it means that all the specifications S are verified (ensuring functional coverage), while all labels are covered (ensuring structural coverage).

The matrix may be quite large. Thus, analyzing it may be painful. Consequently, we provide a way to *consolidate* it by merging all the cells of the same column or row into a single one whenever possible. It helps the verification engineer in the analysis.

5 Formalization of Verification Witnesses

This section formalizes the underlying concepts of a verification activity introduced in the previous section, in particular *verification witnesses*.

5.1 Basic Concepts

Execution. Given a program P with \mathcal{L} list of program point and \mathcal{M} list of possible memory state for P , \vec{x} denotes an input vector for P . An execution $P(\vec{x})$ of a program P on some input datum $\vec{x} = x_1, \dots, x_n$ is a (finite or infinite) sequence $(l_i, m_i)_{0 \leq i}$ of (program) states. Each state is a pair composed of a program point $l \in \mathcal{L}$ and a memory state $m \in \mathcal{M}$. A memory state m at a point l of an execution $P(\vec{x})$ denotes the association of a value to each variable when $P(\vec{x})$ reaches l . For a particular execution $P(\vec{x}) = (l_i, m_i)_{0 \leq i}$, a sub-sequence of states between two program points l_i and l_j ($i \leq j$) is denoted by $(l_i, m_i) \hookrightarrow_{P(\vec{x})} (l_j, m_j)$.

Specification and Functional Coverage. A program requirement R is formalized by a collection of specifications, denoted by $R \triangleq \{S_1, \dots, S_n\}$. Functional coverage is achieved once all specifications in R are verified. A specification S in our framework is an implication $H \Rightarrow C$ which consists in a hypothesis H and a conclusion C . The hypothesis H is a pair (l, h) where l is a program point and $h \in \mathcal{P}(\mathcal{M})$ denotes a property over memory states. Similarly, the conclusion C is a triplet (l, l', r) where l and l' denote two program points and r is a relation between two memory states, i.e. a subset of $\mathcal{P}(\mathcal{M} \times \mathcal{M})$. For a specification $S \triangleq H \Rightarrow C$, the first program point l of C must be the same as the one of the hypothesis H .

Given an execution $P(\vec{x})$ of program P and a specification $S = H \Rightarrow C$, $P(\vec{x}) \rightsquigarrow (l, h)$ (resp. $P(\vec{x}) \rightsquigarrow (l, l', r)$) denotes that P passes through the

hypothesis $H = (l, h)$ (resp. conclusion $C = (l, l', r)$). However, there is a difference when $P(\vec{x})$ passes through the hypothesis H and $P(\vec{x})$ passes through the conclusion C . In the case of $P(\vec{x}) \rightsquigarrow (l, h)$, it means that this execution reaches l and the corresponding memory state satisfies h . However, $P(\vec{x}) \rightsquigarrow (l, l', r)$ means that each time l' is reached from l (i.e. each sequence $(l, m) \hookrightarrow_{P(\vec{x})} (l', m')$), then its corresponding memory state satisfies r (i.e. $(m, m') \in r$). More formally, passing through an hypothesis (resp. a conclusion) is defined as follows:

$$P(\vec{x}) \rightsquigarrow (l, h) \triangleq \exists (l_i, m_i) \in P(\vec{x}), l_i \equiv l \wedge m_i \in h;$$

$$P(\vec{x}) \rightsquigarrow (l, l', r) \triangleq \forall (m, m') \in \mathcal{M}^2 \text{ s.t. } (l, m) \hookrightarrow_{P(\vec{x})} (l', m'), (m, m') \in r.$$

Label and Structural Coverage. A label exactly matches the notion of hypothesis introduced above: it is a pair of a program point l and a condition h that memory states must satisfy at l . Therefore, the notion of passing through an hypothesis is extended to a label.

Extending a label $\{l, h\}$ to a mutant M at label l (that is, the original program mutated at program point l) defines a new label $\{l, h, M\}$, named *label with mutant*. From this point, all labels in the following are considered labeled mutants. Formally, given a program P , a label $\{l, h, M\}$, and an input datum \vec{x} , $P(\vec{x})$ and $M(\vec{x})$ shall contain the same series of program states if and only if $P(\vec{x})$ does not pass through $\{l, h, M\}$, since the mutant shall modify the execution trace of the original program.

Verification Activity. In our context, a verification activity is either a unit proof or a unit test. Both of them tries to provide evidence that each program execution satisfies each program specification. However, both processes are not performed in the same way in practice. It results in difference when measuring coverage.

Consider a program P , an hypothesis $H = (l, h)$, a conclusion $C = (l, l', r)$ and a specification $S = H \Rightarrow C$. Unit test checks that the specification is satisfied in the program by observing some program executions. Each observation is a test case. A successful test case with input datum \vec{x} validates both the hypothesis H and the conclusion C . It provides us the evidence $P(\vec{x}) \rightsquigarrow H \wedge C$. A test is successful whenever all its test cases are themselves successful.

For unit proof, verifying a specification ensures that no execution violates the specification, which means either the verification passes through both H and C , or through the negation of H . Therefore, the evidence for a successful unit proof is $\forall \vec{x}$, either $(P(\vec{x}) \rightsquigarrow H \wedge C)$ or $(P(\vec{x}) \rightsquigarrow \neg H)$. Thus, even if such an evidence ensures that every possible execution satisfies the specification, it does not ensure that the goal C is actually satisfied since the hypothesis H could be invalidated. (contrary to evidence provided by unit testing).

5.2 Verification Witness About the Initial Program

A witness results from a verification activity. It consists of two pieces of information: a *verification technique* (either proof or test) and a *verdict* indicating

which specification is satisfied and by which means. We introduce one kind of witness by verification technique:

- A test witness (denoted by **T**) represents the existence of some test evidence, passing through the label L :

$$\mathbf{T}(S, L, \vec{x}) \triangleq P(\vec{x}) \rightsquigarrow H \wedge C \wedge L.$$

- A proof witness (denoted by **P**) indicates the existence of a proof for some specification S . It ensures that S is satisfied for every execution of program P :

$$\mathbf{P}(S) \triangleq \forall \vec{x}, \text{ either } (P(\vec{x}) \rightsquigarrow H \wedge C) \text{ or } (P(\vec{x}) \rightsquigarrow \neg H).$$

Another witness (less considered here than the other ones) is the error witness. It comes when the verification technique finds an error in the code.

- Error Witness (denoted by **ER**) indicates the existence of an error during the verification of some specification $S = H \Rightarrow C$:

$$\mathbf{ER}(S) \triangleq \exists \vec{x}, (P(\vec{x}) \rightsquigarrow H \wedge \neg C).$$

Hence, witnesses provide us with a formal evidence of all activities performed during our verification campaign.

5.3 Verification Witness About a Mutant

As already explained, our methodology requires a verification of mutants: combining the result of verification in source code and in labeled mutant allows us to deduce relationships between specifications and pieces of code (denoted by labels). In order to separate witnesses provided by verification of a labeled mutant M from the ones coming from the verification of the original code, we introduced additional types of witnesses. Even if the verification of a specification $S = H \Rightarrow C$ for a mutant may produce many different results, only the following two cases are useful in our contexts:

- The specification $H \Rightarrow C$ is satisfied in the mutant,
- The *opposite specification* of S , $H \Rightarrow \neg C$, is satisfied by the mutant.

Each result can be obtained by any verification activity (either test or proof). Hence, the verification of a specification S for a mutant M of a label L can lead to one of the four following witnesses:

- *Witness SP of proof for the labeled mutant*: when the specification S is *proved* on the mutant M of label L :

$$\mathbf{SP}(S, L) \triangleq \forall \vec{x}, \text{ either } (M(\vec{x}) \rightsquigarrow H \wedge C) \text{ or } (M(\vec{x}) \rightsquigarrow \neg H).$$

- *Witness ST of test for the labeled mutant*: when the specification S is *tested* on the mutant M of label L with input datum \vec{x} :

$$\mathbf{ST}(S, L, \vec{x}) \triangleq M(\vec{x}) \rightsquigarrow H \wedge C \wedge L.$$

- *Witness OP of proof for the opposite specification on the labeled mutant:* when the *opposite* specification $H \Rightarrow \neg C$ is *proved* on the mutant M of label L :

$$\text{OP}(S, L) \triangleq \forall \vec{x}, \text{ either } (M(\vec{x}) \rightsquigarrow H \wedge \neg C) \text{ or } (M(\vec{x}) \rightsquigarrow \neg H).$$

- *Witness OT of test for for the opposite specification on the labeled mutant:* when the *opposite* specification $H \Rightarrow \neg C$ is *tested* by an execution $M(\vec{x})$ that passes the label L :

$$\text{OT}(S, L, \vec{x}) \triangleq M(\vec{x}) \rightsquigarrow H \wedge \neg C \wedge L.$$

5.4 Witness Precedence

Since formal proof assesses properties for all input data, while testing only checks a sample of data, there is a natural precedence of proof witnesses (P, SP, OP) over test ones (T, ST, OT). Other combinations of witnesses are also comparable.

In particular, it is possible to have two contradicting witnesses: one witness shows that a specification S is satisfied by the mutant M , while the other one shows that S is violated by M . It could arrive in two different cases:

- Both witnesses are proof witnesses $\text{SP}(S, L)$ and $\text{OP}(S, L)$. From those witnesses, we know that the specification S and its opposite were proved. This situation only occurs when no execution satisfies the hypothesis of S (i.e. the specification is completely useless in the mutant). In this case, $\text{OP}(S, L)$ (which is required to claim that S and L are strongly connected) brings harm to the coverage measure. Therefore, we only keep witness SP and reject OP .
- Both witnesses are test witnesses $\text{ST}(S, L, \vec{x})$ and $\text{OT}(S, L, \vec{y})$. This situation can only occur when $\vec{x} \neq \vec{y}$. In this case, we only keep witness OT because it leads to better coverage.

Hence, we can define a partial ordering over witnesses, illustrated by the diagram below. It shows that SP is greater than any witness over labeled mutants.

$$\begin{array}{ccc} \text{P} & \text{SP} & \rightarrow \text{OP} \\ \downarrow & \downarrow & \downarrow \\ \text{T} & \text{ST} & \leftarrow \text{OT} \end{array}$$

6 Formalization of Verification Campaigns

6.1 Coverage Matrix

A coverage matrix allows an engineer to check the advancement of a verification campaign. Each column of such a matrix represents a specification, while each row represents a label. The matrix records the results of a (possibly still ongoing) verification campaign. Table 1 depicts the possible marks stored in the matrix cells with respect to verification witnesses of specification S that have been computed for the original program P and the mutant M associated to the label L .

Table 1. Marks recording verification and coverage information.

Spec. on P	Spec on M	Witness	Verification information	Coverage information
no witness	no witness			(empty)
error	any witness	$ER(S)$		\times
no witness	proved & opp. proved	$SP(S, L)$?
		$OP(S, L)$		
proved	no witness & tested	$P(S)$	P	?
		$P(S) \wedge (\exists \vec{x}. ST(S, L, \vec{x}))$		
tested	tested	$\exists \vec{x}. (T(S, L, \vec{x}) \wedge ST(S, L, \vec{x}))$	T	
proved	proved	$P(S) \wedge SP(S, L)$	P	–
tested	proved	$(\exists \vec{x}. T(S, L, \vec{x})) \wedge SP(S, L)$	T	
proved	opp. proved & opp. tested	$P(S) \wedge OP(S, L)$	P	\checkmark
		$P(S) \wedge (\exists \vec{x}. OT(S, L, \vec{x}))$		
tested	opp. proved & opp. tested	$(\exists \vec{x}. T(S, L, \vec{x})) \wedge OP(S, L)$	T	
		$\exists \vec{x}. (T(S, L, \vec{x}) \wedge OT(S, L, \vec{x}))$		

Marks in cells encode verification and coverage information for a pair of a specification S and a label L . Figure 4 provides a synthetic view of the possible connections between labels and specifications. An empty cell means that no verification activity occurred. Otherwise, each line contains either one or two marks. When one mark is used, mark \times means that S is invalidated in the original program (thus no coverage information is required), while mark $?$ means that we only have coverage information. When two marks are used, the first mark (either T or P) denotes verification information (either test or proof). The second mark denotes coverage information. Here, mark $?$ means that S is validated while there is no information about the connection between S and L . Mark $-$ means no correlation between S and L , while S is validated. Mark \checkmark means validation of S in source code and invalidation of S in the mutant of L , so S and L are strongly connected.

	Specification
Label	(empty)
	\times
	$?$
	$P?$ $T?$
	$P-$ $T-$
	$P\checkmark$ $T\checkmark$

Fig. 4. Possible marks in coverage matrix cells.

6.2 Consolidated Coverage Matrix

A coverage matrix is usually quite large. For instance, our running example have 11 specifications and 12 labels, with made a total 132 cells in the coverage matrix.

Consequently, it is not easy for a verification engineer to handle it in a useful way. For solving this issue, we provide a way to consolidate it by gathering columns and rows. This consolidation results in adding one column named *specification consolidation* and one row named *label consolidation* as shown in Fig. 5.

	Specification	label consolidation
	(empty)	
Label	\times	\checkmark
	?	–
	$P?$ $T?$?
	$P-$ $T-$	\times
	$P\checkmark$ $T\checkmark$	
specification consolidation	$P T ? - \times$	

Fig. 5. Consolidated coverage matrix.

Mark meanings are slightly modified in the new cells as depicted in Table 2. For a specification S , mark P (*resp.* T) denotes that S is tested (*resp.* proved) and strongly connected to at least one label. Mark \times means that S is invalidated. Mark – indicates no correlation of S with any label, which means that either one piece of code is missing (in other words, one expected functionality is probably not implemented), or the specification is absurd (e.g. it has unsatisfiable

Table 2. Consolidation rules for specifications and labels.

If specification S has ...	Then the symbol for the consolidation cell of S is ...	It means ...
At least one cell $P\checkmark$	P	S are proved
At least one cell $T\checkmark$	T	S is tested
All cells are either $P-$ or $T-$	–	Either S is absurd, or one label is missing for S
At least one cell \times	\times	S is invalidated
Other case	?	Verification of S is inconclusive
If label L has ...	Then the symbol for the consolidation cell of L is ...	It means ...
At least one cell $P\checkmark$	\checkmark	L is strongly connected to at least one specification
At least one cell $T\checkmark$		
All cells are either $P-$ or $T-$	–	No correlation between L and any specification
Other case	?	Not enough coverage information for label L

hypotheses). This kind of information may be particularly useful for debugging code and/or specification. Mark ? means that the verification is currently inconclusive: an additional verification activity is required. For a label L , mark ✓ means that L is strongly connected to at least one specification. Mark – means no correlation between L and any specification. Mark ? is used for all the other cases which are inconclusive with respect to the coverage criterion.

Full coverage (as per DO-178) is reached if, after consolidating the coverage matrix, the resulting consolidated specification table only contains marks P or T , meaning that every specification is verified, while the resulting consolidated label table only contains mark ✓, meaning that every label is covered.

6.3 Verification Campaign Automatization

This section proposes an algorithm, shown in Fig. 6 and currently being implemented as a new Frama-C plugin. It consists of two consecutive steps that automatize as much as possible a verification campaign, in order to quickly reach full coverage.

1. Proving.

- (a) *Manually* provide all the necessary data (initial code, formal specification, labels and mutants) to an automatic proof technique (e.g. plug-in WP of Frama-C with an associated SMT solver).
- (b) *Automatically*, from the results, fill the (consolidated) coverage matrix.
- (c) *Manually* choose the next action according to the coverage matrix:
 - if full coverage is reached, the verification campaign is complete;
 - if *an error is found*, correct it and restart the verification campaign;
 - if *no error is found* but coverage is yet incomplete, continue the verification campaign by using another technique (e.g. another prover) or testing. If one goes for testing, goto step 2.

2. Testing.

- (a) *Manually* define test cases in order to test the uncovered specifications (not containing any mark ✓ in the coverage matrix). In order to quickly reach full coverage, try as much as possible to choose test cases which can pass through the remaining labels (not containing any mark ✓ in the coverage matrix).
- (b) *Manually* provide all the necessary data (specification, code, label, mutant) and test case to a testing tool (e.g. plug-in E-ACSL of Frama-C).
- (c) *Automatically*, from the results, fill the (consolidated) coverage matrix.
- (d) *Manually* choose the next action according to the coverage matrix:
 - if full coverage is reached, the verification campaign is complete;
 - if *an error is found*, correct it and restart the verification campaign;
 - if *no error is found* but coverage is yet incomplete, continue the verification campaign by defining (at least) one other test case.

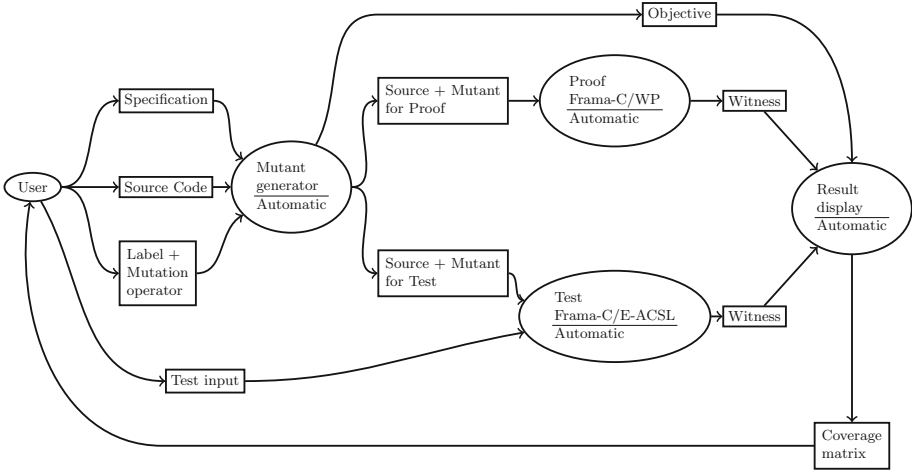


Fig. 6. Algorithm for automatizing a verification campaign.

7 Experiment

This section applies the previous algorithm to our running example of Sect. 3. To demonstrate that our verification coverage is able to detect a missing specification, we intentionally remove the VALID specification from our example.

Labels and their associated mutant are defined according to statement coverage criterion. Therefore, one label is attached to each statement. Two kinds of mutant operators, *replace* and *erase*, are used to define the labeled mutant of each label, as explained in Sect. 4.1.

We now apply the algorithm of Sect. 6.3. First, we try to prove that the function satisfies its specifications by using plug-in WP of Frama-C. The results are stored in the coverage matrix (partially) shown in Fig. 7.

Except for the four rows from label `Lstmt1` to label `Lstmt4`, all rows of the matrix contain at least one mark $P\checkmark$ which means a mark \checkmark in the consolidated label table. Similarly, each column in categories ERR and OK contains at least one mark $P\checkmark$ which means a mark P in the consolidated specification table. It means that the verification campaign already succeeds for the corresponding labels and specifications: plug-in WP was able to validate them alone.

The four rows from label `Lstmt1` to label `Lstmt4` do not contain mark $P\checkmark$. They lead to four marks ? in the consolidated label table. Also, no column in the category VALUE contains mark $P\checkmark$, hence three marks ? in the corresponding cells of the consolidated specification table. Consequently, these 4 labels and the 3 specifications in category VALUE were not covered by plug-in WP.

We now choose the plug-in E-ACSL to test them. For that purpose, we define three test cases which pass through the remaining labels `Lstmt1` to `Lstmt4`. Figure 8 shows the resulting updated cells of the coverage matrix. From the consolidated label table, we conclude that label `Lstmt4` is the only label not yet

	ERR					OK	VALUE			consol. label
	error 1	error 2	...	error 6	error 7	ok	low signal	medium signal	high signal	
Lreturn1	<i>P✓</i>	<i>P-</i>	...	<i>P-</i>	<i>P-</i>	<i>P-</i>	<i>P-</i>		<i>P-</i>	✓
Lreturn2	<i>P-</i>	<i>P✓</i>	...	<i>P-</i>	<i>P-</i>	<i>P-</i>	<i>P-</i>		<i>P-</i>	✓
		
Lreturn6	<i>P-</i>	<i>P-</i>	...	<i>P✓</i>	<i>P-</i>	<i>P-</i>	<i>P-</i>		<i>P-</i>	✓
Lreturn7	<i>P-</i>	<i>P-</i>	...	<i>P-</i>	<i>P✓</i>	<i>P-</i>	<i>P-</i>		<i>P-</i>	✓
Lstmt1	<i>P-</i>	<i>P-</i>	...	<i>P-</i>	<i>P-</i>	<i>P-</i>	<i>P?</i>		<i>P-</i>	?
Lstmt2	<i>P-</i>	<i>P-</i>	...	<i>P-</i>	<i>P-</i>	<i>P-</i>	<i>P-</i>		<i>P?</i>	?
Lstmt3	<i>P-</i>	<i>P-</i>	...	<i>P-</i>	<i>P-</i>	<i>P-</i>	<i>P-</i>		<i>P-</i>	?
Lstmt4	<i>P-</i>	<i>P-</i>	...	<i>P-</i>	<i>P-</i>	<i>P-</i>	<i>P-</i>		<i>P-</i>	?
Lreturn0	<i>P-</i>	<i>P-</i>	...	<i>P-</i>	<i>P-</i>	<i>P✓</i>	<i>P-</i>		<i>P-</i>	✓
consol. spec.	<i>P</i>	<i>P</i>	...	<i>P</i>	<i>P</i>	<i>P</i>	?	?	?	

Fig. 7. Coverage matrix after running plug-in WP.

covered. It prevents us to complete the verification campaign. Proofreading the code allows us to establish that the related piece of code has no correlation with any specification.

	low signal	medium signal	high signal	consol label
Lstmt1	<i>P✓</i>		<i>P-</i>	✓
Lstmt2	<i>P-</i>		<i>P✓</i>	✓
Lstmt3	<i>P-</i>	<i>T✓</i>	<i>P-</i>	✓
Lstmt4	<i>P-</i>	<i>T?</i>	<i>P-</i>	?
consol. spec	<i>T</i>	<i>T</i>	<i>T</i>	

Fig. 8. Interesting subset of the coverage matrix after running plug-in E-ACSL.

We now add an additional specification corresponding to that piece of code. It matches the previously removed VALID specification. Running the very same test cases again leads to an updated coverage matrix. Figure 9 shows the only interesting cells. It allows us to conclude that our verification campaign is complete: we reach full functional and structural coverage by combining proof with plug-in WP and test with plug-in E-ACSL run on only three test cases.

	VALUE	VALID	consol label
	medium signal	valid flag	
Lstmt4	<i>T?</i>	<i>T✓</i>	✓
consol spec	<i>T</i>	<i>T</i>	

Fig. 9. Coverage cells of the added specification after running E-ACSL again.

8 Conclusion and Future Work

By combining and enhancing labels [5,6] and mutations [16,25], we introduce a new notion of verification coverage that allows us to combine test and proof for verifying a group of specifications related to the same piece of code. It subsumes both the usual notions of functional coverage and structural coverage.

Our verification coverage establish connections between pieces of code represented by labels and functional specifications. It allows us to measure verification and coverage rates through the number of specifications and labels strongly connected. Thus it provides a way to decrease the influence of a particular verification method when measuring coverage.

Based on this verification coverage, we also introduce an algorithm that automatizes most parts of a verification campaign combining test and proof in order to complete the verification process as quickly as possible. This algorithm is currently being implemented as a new Frama-C plug-in.

We also formalize our work thanks to new notions of verification witnesses and coverage matrices. Coverage matrices are consolidated *per* specification and *per* label in order to synthesize the verification and the coverage results. Such consolidations help the verification engineer to decide the next verification activity to be performed.

Future work includes studying the impact of mutation and coverage criteria on verification coverage. We also aim at extending existing toolchains in order to automatize label generation, choice of mutation operators and test case generation with respect to different coverage criteria. It would reduce the parts of our algorithm that are not yet automated.

Acknowledgment. The authors would like to thank the anonymous reviewers for their helpful comments and feedbacks. This work was partly supported by project VESSEDIA, which has received funding from the European Union’s Horizon 2020 Research and Innovation Program under grant agreement No 731453. It was also partly supported by European Union’s Involvement in Midi-Pyrénées through its Development Fund.

References

1. Agrawal, H., Demillo, R.A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E.W., Martin, R.J., Mathur, A., Spafford, E.: Design of mutant operators for the C programming language. Technical report, SERC-TR-41-P, Purdue University (1999)
2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2008)
3. Artho, C., Biere, A.: Combined static and dynamic analysis. Electron. Notes Theor. Comput. Sci. (ENTCS) **131**, 3–14 (2005)
4. Bardin, S., Delahaye, M., David, R., Kosmatov, N., Papadakis, M., Traon, Y.L., Marion, J.Y.: Sound and quasi-complete detection of infeasible test requirements. In: IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015, pp. 1–10 (2015)

5. Bardin, S., Chebaro, O., Delahaye, M., Kosmatov, N.: An all-in-one toolkit for automated white-box testing. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 53–60. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09099-3_4
6. Bardin, S., Kosmatov, N., Cheyner, F.: Efficient leveraging of symbolic execution to advanced coverage criteria. In: IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014, pp. 173–182 (2014)
7. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. <http://frama-c.com/acsl.html>
8. Bishop, P., Bloomfield, R., Cyra, L.: Combining testing and proof to gain high assurance in software: a case study. In: IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 248–257 (2013)
9. Brahmi, A., Delmas, D., Essoussi, M.H., Randimbivololona, F., Atki, A., Marie, T.: Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In: European Congress on Embedded Real Time Software and Systems (ERTSS 2018) (2018)
10. Brown, D., Delseny, H., Hayhurst, K., Wiels, V.: Guidance for using formal methods in a certification context. In: European Congress on Embedded Real Time Software and Systems (ERTS 2010) (2010)
11. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage metrics for formal verification. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 111–125. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_11
12. Dadeau, F., Giorgetti, A., Bouquet, F., Enderlin, I.: Contract-based testing for PHP with Praspel. *J. Syst. Softw.* **136**, 209–222 (2018)
13. Dadeau, F., Ledru, Y., du Bousquet, L.: Measuring a Java test suite coverage using JML specifications. *Electron. Notes Theor. Comput. Sci.* **190**, 21–32 (2007)
14. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Symposium on Applied Computing (SAC 2013), pp. 1230–1235. ACM (2013)
15. Delamaro, M.E., Deng, L., Durelli, V.H.S., Li, N., Offutt, J.: Experimental evaluation of SDL and One-Op mutation for C. In: IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014, pp. 203–212 (2014)
16. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. *IEEE Comput.* **11**, 34–41 (1978)
17. Filliâtre, J.C.: Deductive software verification. *Int. J. Softw. Tools Technol. Transf.* **13**, 397 (2011)
18. Floyd, R.W.: Assigning meanings to programs. In: Colburn, T.R., Fetzer, J.H., Rankin, T.L. (eds.) Program Verification, pp. 65–81. Springer, Dordrecht (1993). https://doi.org/10.1007/978-94-011-1793-7_4
19. Hoare, C.A.R.: An axiomatic basis for computer programming. In: Gries, D. (ed.) Programming Methodology, pp. 89–100. Springer, New York (1978). https://doi.org/10.1007/978-1-4612-6315-9_9
20. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Jakobowski, B.: Frama-C: a software analysis perspective. *Formal Aspects Comput.* **27**, 573–609 (2015)
21. Kiss, B., Kosmatov, N., Pariente, D., Puccetti, A.: Combining static and dynamic analyses for vulnerability detection: illustration on Heartbleed. In: Piterman, N. (ed.) HVC 2015. LNCS, vol. 9434, pp. 39–50. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26287-1_3

22. Kosmatov, N., Signoles, J.: Runtime assertion checking and its combinations with static and dynamic analyses. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 165–168. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09099-3_13
23. Marcozzi, M., Bardin, S., Delahaye, M., Kosmatov, N., Prevosto, V.: Taming coverage criteria heterogeneity with LTest. In: IEEE 10th International Conference on Software Testing, Verification and Validation, ICST 2017, pp. 500–507 (2017)
24. Marcozzi, M., Delahaye, M., Bardin, S., Kosmatov, N., Prevosto, V.: Generic and effective specification of structural test objectives. In: IEEE 10th International Conference on Software Testing, Verification and Validation, ICST 2017, pp. 436–441 (2017)
25. Morell, L.J.: A theory of fault-based testing. *IEEE Trans. Softw. Eng.* **16**, 844–857 (1990)
26. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B.: Testing or formal verification: DO-178C alternatives and industrial experience. *IEEE Softw.* **30**, 50–57 (2013)
27. Offutt, A.J., Voas, J., Payne, J.: Mutation operators for Ada. Technical report ISSE-TR-96-09 (1996)
28. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. *IEEE Trans. Softw. Eng.* **24**, 161–173 (1998)
29. RTCA (Firm). and EUROCAE (Agency): DO-178C, Software Considerations in Airborne Systems and Equipment Certification. Document (RTCA (Firm)) (2011)
30. RTCA (Firm). SC-205 and EUROCAE (Agency). Working Group 71: Formal Methods Supplement to DO-178C and DO-278A. Document (RTCA (Firm)), RTCA, Incorporated (2011). <https://books.google.fr/books?id=nYhyMwEACAAJ>
31. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a runtime verification tool for safety and security of C programs. Tool paper. In: International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES 2017), pp. 164–173 (2017)
32. Software and Engineering Standards Committee: IEEE Standard for Software and System Test Documentation. *IEEE Std 829-2008*, pp. 1–118 (2008)
33. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 281–292. Springer, Heidelberg (2005). https://doi.org/10.1007/11408901_21



Detection of Security Vulnerabilities in C Code Using Runtime Verification: An Experience Report

Kostyantyn Vorobyov, Nikolai Kosmatov^(✉), and Julien Signoles

CEA, LIST, Software Reliability and Security Laboratory,
PC 174, 91191 Gif-sur-Yvette, France
{kostyantyn.vorobyov,nikolai.kosmatov,julien.signoles}@cea.fr

Abstract. Despite significant progress made by runtime verification tools in recent years, memory errors remain one of the primary threats to software security. The present work is aimed at providing an objective up-to-date experience study on the capacity of modern online runtime verification tools to automatically detect security flaws in C programs. The reported experiments are performed using three advanced runtime verification tools (E-ACSL, Google Sanitizer and RV-MATCH) over 700 test cases belonging to SARD-100 test suite of the SAMATE project and Toyota ITC Benchmark, a publicly available benchmarking suite developed at the Toyota InfoTechnology Center. SARD-100 specifically targets security flaws identified by the Common Weakness Enumeration (CWE) taxonomy, while Toyota ITC Benchmark addresses more general memory defects, as well as numerical and concurrency issues. We compare tools based on different approaches – a formal semantic based tool, a formal specification verifier and a memory debugger – and evaluate their cumulative detection capacity.

The results of the experiments indicate that the selected tools cumulatively detected 84% of all seeded defects. Although for several categories of errors detection rates are higher, we observed that applying several tools is beneficial for uncovering certain issues. For instance, in detecting concurrency issues of the Toyota ITC Benchmark, the highest per-tool result was 73%, whereas cumulative detection ratio of all three tools used together was 93%.

Keywords: Runtime verification · Software security · Memory safety
Dynamic analysis · Experience report

1 Introduction

The C programming language is one of the most commonly used languages for development of critical software such as operating systems, drivers, hypervisors, cryptography libraries, etc. At the same time C lacks protection mechanisms, leaving the entire responsibility for correct management of memory and resources

to the developer. Execution of a badly written C program can lead to an *undefined behaviour*, one that is not formally specified by the C language standard [21]. Undefined behaviours in C programs are potential causes of *memory errors*, such as buffer overflows, format string vulnerabilities, double free violations and many others. Memory errors are known to be one of the main threats to software security [40] because they can be exploited by an attacker to trigger execution of malicious code capable of stealing or corrupting data, provoking system failures and even taking control of the affected machine. Detecting memory errors is therefore of the utmost importance for security of code.

One way to automatically detect memory errors in a program is by using static program analysis techniques, such as abstract interpretation [9], deductive methods [13] and model checking [16, 26]. An orthogonal approach to detecting memory errors is by means of dynamic analysis, for instance, using online runtime verification [17], a technique that observes an execution of the program and detects errors before they occur. Over recent years online runtime verification (sometimes referred to as *monitoring*) has been successfully used to detect numerous undefined behaviours in C programs. For instance, over 300 previously unknown errors have been detected in the Chromium browser using AddressSanitizer [29].

Static analysis techniques analyze programs without executing them, but typically require fine-tuning to get usable results (including a low number of false alarms for abstract interpretation, proved properties for deductive methods and termination in a reasonable amount of time for model checking). In contrast to static methods, runtime verification analyzes programs by executing them in a concrete setting, eliminating the need for fine-grained tuning and reducing engineering effort to get proper results. One of the drawbacks of runtime verification, however, is that it analyzes one behaviour at a time. This is different to static analysis techniques that typically analyze all program behaviours. It is also possible to run the program in a simulated environment. The benefit of this approach is that it does not need a concrete setting or extensive tuning. Simulated runs, however, typically suffer from significant performance overheads.

Motivation. Due to increasing interest in verification techniques and tools in academia and industry they rapidly evolve, offering novel solutions or significant improvements almost every year. As such, related experimental studies tend to become outdated very fast. Most of such studies are performed by the authors of a tool and aimed at comparing it with its previous version or a few related tools in order to demonstrate the benefits of the proposed solution. Such studies do not provide a global view of the state of the art: for instance, such an important research question as the *cumulative detection capacity* of several tools used together is often not addressed.

More importantly, the main focus of experiments with monitoring tools is about to change. Since the execution overhead used to be an important barrier to their application, many prior experimental studies [4, 29, 31, 37, 41, 42] focused primarily on tools' performance and often conducted experiments

using computationally-intensive programs rather than investigated the detection power. However, due the increasing computation power of modern computers and the recent progress of monitoring techniques, the execution overhead does not necessarily represent critical limitations for many monitoring tools today.

Finally, software security has become one of the most important concerns in many domains of software engineering and automatic detection of potential security vulnerabilities is an attractive and promising application for monitoring tools. Both researchers and practitioners need to have an objective and up-to-date vision of what kinds of security vulnerabilities can be detected by modern state-of-the-art tools and which ones are likely to remain undetected.

The Runtime Verification Competitions [2, 18] (held in the context of the International Conference on Runtime Verification since 2014) evaluate both soundness and performance of the participating tools. These events are very helpful for the tool developers allowing them to compare their tools with others and identify their weaknesses. However, the competitions cannot provide a complete global picture either. This is because the participating tools are typically evaluated on a handful of benchmarks submitted by the authors of the tools. Further, these competitions do not focus on security vulnerabilities.

Goals and Means. The purpose of this work is to provide an experience report evaluating the capacity of bug checking tools to detect security vulnerabilities with an emphasis on memory errors. We choose the C programming language as one of the most relevant languages for security-critical software. Our first objective is to give an *up-to-date vision* of this detection capacity at a large scale.

To reach this goal, we selected two existing publicly available benchmark suites with over 700 test cases in total. They are representative of security-related vulnerabilities and already classified into several subcategories. The first benchmark we used is SARD-100 [11], a test suite belonging to the Software Assurance Metrics And Tool Evaluation (SAMATE) project of the National Institute of Standards and Technology (NIST). SARD-100 targets security flaws of the Common Weakness Enumeration (CWE) taxonomy [8]. The second suite, Toyota ITC Benchmark [34], is a publicly available benchmarking suite developed at the Toyota InfoTechnology Center. Toyota ITC Benchmark mostly focusses on memory errors, but also contains a number of programs seeded with numerical and concurrency issues.

Evaluating tools of such a large number of test cases requires tool automation. We therefore choose to run the selected tools using a fixed set of documented options (when required) without any specific tuning. This approach allows us to give a *realistic and objective picture* of how effective the compared tools can be when used by a competent engineer who is not a developer of the tool. This approach particularly fits security vulnerability detection: in this context, most end-users favour automation to precision. Since static tools require such a fine-tuning, we exclude them for our experimentation and focus on runtime verification tools only. We selected a representative subset of online monitoring tools (E-ACSL,

Google Sanitizer and RV-MATCH) following a number of well-defined criteria such as *availability*, *robustness*, capacity of *online monitoring*¹. Additionally, we try to consider different scientific approaches – a formal semantic based tool, a formal specification verifier and a memory debugger – in order to objectively evaluate the cumulative detection power of these tools used together.

As we have mentioned above, previous studies often focused on tool performance that is not a critical limitation for many monitoring tools today. Also, including tool performance in our evaluation would be unfair for RV-MATCH which is based on a simulated environment. Therefore, we also exclude performance evaluation from our study in order to focus only on detection capability. This way, it allows us to provide a *global and objective view* of how effective modern runtime verification tools can be for verification engineers looking for security vulnerabilities.

Contributions. The contributions of this paper include:

- an experimental campaign aimed at evaluating the capacity to detect security-related vulnerabilities using three modern runtime verification tools, E-ACSL, Google Sanitizer and RV-MATCH, and two benchmarking suites, SARD-100 and Toyota ITC Benchmark;
- an experience report presenting the recorded results separately for each (sub-)category and each tool, as well as globally over the three tools and for all programs; detailed results of each tool for each benchmark program are available in the companion report²;
- a careful analysis of the reported results showing where we stand with runtime detection of security vulnerabilities using monitoring tools.

Outline. The rest of the paper is organized as follows. Section 2 presents related work on runtime monitoring techniques and tools. Section 3 describes our experimental setup and explains the choice of selected tools and benchmarks. Section 4 presents and discusses the experimental results, and Sect. 5 summarizes the threats to validity of the present experiment. Finally, Sect. 6 presents concluding remarks and future work directions.

2 Related Work

We now review techniques that focus on runtime detection of memory errors in C programs.

Online monitoring of C programs for memory-related software vulnerabilities goes decades back. One of the oldest (yet still active) techniques to detecting defects in C programs at runtime is Rational Purify [20]. This tool instruments

¹ Which dynamically analyzes a program run on the fly, unlike *offline monitoring* based on previously recorded execution traces.

² See <https://goo.gl/S4NF5m>.

object files with additional instructions that track memory state of an executing program and identify operations on unallocated or uninitialized memory locations. More recent techniques, such as MemCheck [31], SGCheck [32], or Dr. Memory [4] achieve a similar task using dynamic binary instrumentation, an approach that injects memory monitors to binary programs at execution time.

One of the benefits of binary-level memory monitoring is its ability to check every memory access (typically using memory shadowing) including those occurring in C library or third-party code. Binary monitors are widely used during development and testing stages in many large-scale software projects [39]. However, since such tools reason at the level of instructions they often fail to detect issues visible only at a source level of the language including such problems as misuse of pointers, type violations or use of variable addresses outside of scope of their definition³.

Source-level techniques to runtime detection of memory errors have also been developed. Seminal work of Jones and Kelly [23] enabled runtime bounds checking using a splay tree to track program pointers and bounds of objects they reference. At runtime this technique checks operations on pointers (e.g., dereference, arithmetic) by querying the splay tree-shaped metadata. This technique served as a building block for a number of runtime memory error detectors. CRED [28] added support for tracking out-of-bounds pointers using an additional auxiliary hash table. Dhurjati and Adve [12] improved bounds checking technique using Automatic Pool Allocation memory partitioning. Baggy bounds checking [1] used specialized memory allocator to constrain size and alignment of allocated blocks and used array-based lookup to improve performance. Mem-Safe [35] used a mix of static and dynamic analyses to prevent memory errors at runtime via a combination of object and pointer metadata.

Even though effective for tracking memory errors the above techniques have not gone beyond research prototypes, as such they are difficult to use in practice without expert knowledge.

Google AddressSanitizer [29] is probably the first successful attempt to create a widely-used industrial-strength source-level monitor for C/C++ programs. AddressSanitizer uses shadow memory to track program allocations at runtime using source-to-source transformations. This tool benefits from a compact shadow state encoding that tracks 8-byte sequences by only 3 bits. This allows for significant reduction of monitoring overhead costs. Initially integrated with the clang compiler AddressSanitizer has later been ported to GCC replacing mudflap [14] tool. Nowadays AddressSanitizer is a part of a bigger tool suite called Google Sanitizer that contains several tools that focus on different issues: AddressSanitizer (illegal memory accesses and memory leaks), MemorySanitizer [37] (uninitialized memory accesses), ThreadSanitizer [30] (data races and deadlocks) and UndefinedBehaviourSanitizer [38] (undefined behaviours).

³ More precisely, in some cases memory corruption errors caused by such violations are detectable by binary analysis tools but only after they are disconnected from the source code error that caused them.

One of the disadvantages of fully automatic analyzers such as AddressSanitizer or MemCheck is that they cannot easily enforce custom properties (e.g., function contracts or loop invariants). Such issues can be addressed using behaviour interface specification languages such as E-ACSL. E-ACSL [10] is a runtime verification tool for C programs built atop the Frama-C [24] framework for source-code analysis. E-ACSL transforms a C program P annotated with formal specifications in the E-ACSL specification language into a monitored program P' that behaves similar to P but aborts at runtime if any given annotation is violated. Formal E-ACSL specifications usable by the tool can be provided manually or generated automatically by another tool such as the Frama-C kernel or its RTE plug-in [24]. The present focus of E-ACSL is runtime enforcement of function contracts, detection of integer overflows and validating memory accesses made by the program at runtime.

An orthogonal way of detecting errors (including security vulnerabilities) in C programs is by using simulated environments. One such tool is RV-MATCH [19]. The aim of RV-MATCH is to ensure that a run of a C program strictly conforms to the ISO C11 [5] standard, i.e., does not rely on implementation specific or undefined behaviours described by the standard.

RV-MATCH is built using the \mathbb{K} semantics framework [27]. \mathbb{K} is a program analysis framework based on term rewriting that allows to define rigorous semantics for a target programming language. The framework also provides several tools for formal analysis of programs written in the target language including a symbolic execution engine, a semantic debugger, a model checker and a deductive verifier. RV-MATCH uses formal executable C semantics [15] to instantiate the \mathbb{K} framework for C and interprets programs according to the formal operational semantics of the language.

Another approach to enforcing memory safety of C programs is called Cyclone [22]. Cyclone is a safe dialect of the C programming language designed to retain C semantics and performance and at the same time prevent memory-related errors. To achieve this goal Cyclone imposes restrictions on C programs. For instance, Cyclone limits pointer arithmetic, enforces pointer initialization and disallows unsafe casts. This approach also uses “fat-pointers” to enable runtime bounds checks and prevent accesses to unallocated memory. Presently Cyclone is no longer supported but some of its ideas made into the Rust programming language [25] that pursues similar goals.

Overall, technique such Cyclone or Rust are compromises between safety and security and performance that prevent many issues commonly associated with C programs by design. Using such techniques for an existing program, however, may be a daunting task as it requires porting a program to one of these languages.

3 Experimental Setup

3.1 Objectives and Evaluation Approach

The key objective of this paper is to evaluate the capacity of state-of-the-art monitoring tools to detect security vulnerabilities in C programs. We address

this objective using an empirical study that analyses benchmarked code belonging to the test suite #100 of the Software Assurance Reference Dataset Project (SARD) [11] and the Toyota InfoTechnology Center dataset [6,34] using E-ACSL [10], Google Sanitizer [29,30,37,38] and RV-MATCH [19]. For each tool we compute its detection ratio (i.e. the number of discovered bugs over the total number of bugs) and report the results.

More precisely, this evaluation seeks answers to the following research questions:

(RQ1). What is the cumulative detection ratio of the selected state-of-the-art tools used together for each category of vulnerabilities?

(RQ2). What is the detection ratio of each of the selected tools for each category of vulnerabilities?

(RQ3). Are different tools complementary in the bugs they detect?

The following sections provide details on the choice of the tools and benchmarks used in this empirical study and discusses evaluation methodology.

3.2 Selected Tools

We now discuss the key selection criteria of the runtime verification tools used in the present experiment.

Availability and Robustness. One of our goals is to evaluate runtime verification techniques usable by most developers. For this experimentation we select freely available, robust tools capable of verifying C code at runtime with no or little manual effort. Consequently we reject research prototypes, incomplete implementations, or techniques usable only by experts.

Memory Analysis. Many vulnerabilities in C occur due to its almost unrestricted use of memory. Therefore we only consider tools capable of analysing the memory state of a running program. Another source of security flaws in C programs are executions that lead to undefined behaviours with respect to a chosen ISO C standard. Since such issues are defined at a source level of the C programming language we only consider tools using source code analysis. Consequently we reject binary monitors such as MemCheck [31] or Dr Memory [4].

Online Monitoring. In a security-oriented analysis it is important to prevent errors, as otherwise a vulnerability can be exploited before it can be reported. To address this requirement for this experimentation we consider only online runtime verification tools capable of detecting vulnerabilities before they occur.

Potential Complementarity. To have a global vision of the cumulative detection capacity of different state-of-the-art techniques, we select tools using different approaches to runtime error detection. In other words, we reject tools that approach a problem similarly but differ in implementation.

Based on the above requirements for this experimentation we select the following tools (in alphabetic order):

- E-ACSL** [10] – a verifier of a rich specification language;
- Google Sanitizer** [29,30,37,38] – a source-level memory debugger;
- RV-Match** [19] – a verifier of formal language semantics.

More detailed descriptions of the selected tools is given in Sect. 2.

3.3 Selected Benchmarks

We now briefly discuss the source code benchmarks used in this empirical study.

SARD-100 [11] is a test suite belonging to the Software Assurance Metrics And Tool Evaluation (SAMATE) project of the National Institute of Standards and Technology (NIST). Each SARD-100 program contains a vulnerability of the Common Weakness Enumeration (CWE) taxonomy [8]. Initially developed for testing against source code security analyzers based on Source Code Security Analysis Tool Functional Specification [3] SARD-100 explores such important security issues as SQL and command injections, buffer overflows, format string vulnerabilities, use-after-free errors and others (21 CWE vulnerabilities in total).

Toyota ITC Benchmark [6,34] is a publicly available benchmarking suite developed at Toyota InfoTechnology Center, USA. The suite is based on Annex A (Source Code Weaknesses) of Source Code Security Analysis Tool Functional Specification [3]. Toyota ITC Benchmark consists of 638 test cases exploring 9 defect types and 51 sub-types. Toyota ITC Benchmark focusses on memory defects (e.g., static, dynamic, stack, pointer arithmetic), numerical defects (such as division by zero or integer overflows) and concurrency issues (race conditions, deadlocks).

One of the key factors for selecting SARD-100 and Toyota ITC Benchmark for the present experimentation is that both contain code samples originating from reliable sources with clearly marked vulnerabilities. The test cases belonging to these suites allow to explore a broad range of security-related issues typical to C programs.

3.4 Evaluation Methodology

During the present experimentation we perform a series of program runs under E-ACSL, Google Sanitizer and RV-MATCH monitoring and compute detection ratio of each tool. The percent detection ratio of a tool run over programs containing N defects is computed as $D/N * 100$, where D is the number of defects detected by the analyzer.

In this experiment we used latest stable versions of the tools available at the time of the experiment⁴. The RV-MATCH and Google Sanitizer monitored

⁴ At the time of this writing (March, 2018) E-ACSL-0.9 is not available publicly yet and was obtained from the developers of the tool. E-ACSL-0.9 is scheduled to be released in May, 2018.

programs were obtained via `kcc-1.0` and `clang-4.0.1` compilers respectively. Since these tools make use of built-in analyses no external specifications were provided. Programs monitored via E-ACSL (version 0.9) were obtained via its driver script called `e-acsl-gcc.sh` that takes a C program, automatically annotates it using the RTE [24] plugin of Frama-C and finally compiles it using the `gcc` compiler (`gcc` version 5.4.0 was used). For E-ACSL analysis we also used partial function contracts provided by the Frama-C standard library.

During this experimentation a defect is considered detected if it is reported either during compile stage or before its occurrence at runtime. We consider runs of monitors that failed due to internal errors or by intercepting signals as missing defects. Also, since Google Sanitizer consists of 4 separate tools (AddressSanitizer, UndefinedBehaviourSanitizer, ThreadSanitizer and MemorySanitizer) we consider a defect detected if it is reported by either of the tools.

The monitored programs were run once per tool except for the cases using random number generators that potentially surpass the execution of vulnerable code. In such cases the benchmarks were run continuously until erroneous paths were explored.

The tools used in this experiment were run using inputs provided via the benchmarking suites. Where such inputs were not available (several programs from SARD-100) we used inputs that explored vulnerabilities in the benchmarked code.

The platform for all results reported here was 2.30 GHz Intel i7 processor with 16 GB RAM, running 64-bit Gentoo Linux.

False Positives. Even though some of the tools used in this experimentation can produce false alarms (in particular AddressSanitizer [29]), we do not report false positive detection rate for the tools. This is because no false alarms were detected during this experimentation. In other words, for this particular study the rate of false positive defects equates to 0% in all cases. We manually verified that all defects reported by the tools during this study correspond to actual defects.

Runtime Overheads. In dynamic analysis performance overhead of a technique is an important issue, however, this experimentation does not report runtime or memory overheads of the tools. This is because of the following reasons.

The goal of this experimentation is in identifying a technique's capability to detect different security-related issues rather than evaluating its applicability to large and computationally intensive programs. Consequently, the experiment uses small programs whose execution time in most cases does not exceed one second. The size of the programs used during this experiment is not representative for evaluating the tools' robustness with respect to performance overhead.

Performance overhead of the tools used in this experimentation was assessed in prior work. For instance, [42] reports on runtime and memory overheads of both E-ACSL and AddressSanitizer using computationally intensive benchmarks for CPU testing developed by the Standard Performance Evaluation Corporation

(SPEC CPU) [36]. This experimentation has shown that for the selected SPEC CPU programs the runtime overheads of E-ACSL (on average 19 times compared to execution of unmonitored programs) of AddressSanitizer (1.58 times). Further, the experiment has shown that E-ACSL was more memory efficient comparing to AddressSanitizer (2.48 vs. 4.22 times on average).

Thorough empirical study evaluating performance overheads of RV-MATCH has not been conducted. This is mainly because at the present stage of implementation RV-MATCH is not yet ready to deal with large programs. A preliminary result reported in [19] shows that analysis of a small example program consisting of 10,000 loop iterations took 11 s. Taking into account that an unobserved execution of the same program is under 0.01 s, such a result might suggest that overheads of RV-MATCH are presently too high to be used in practice with real-world programs.

4 Experimental Results

We now discuss detection results of using E-ACSL, Google Sanitizer and RV-MATCH over SARD-100 dataset and Toyota ITC Benchmark.

4.1 Results for SARD-100

Table 1 shows detection results of E-ACSL, Google Sanitizer and RV-MATCH over C benchmarks belonging to the SARD-100 dataset. The leftmost column of the table shows a given CWE vulnerability, the rest of the columns show error detection ratio in percent followed by the number of discovered and overall defects.

The overall results indicate that E-ACSL, Google Sanitizer and RV-MATCH detected 67%, 56% and 54% of bugs respectively. Cumulative error detection ratio (i.e., with respect to defects detected by at least one of the tools) is 67%.

All tools missed security vulnerabilities that do not lead to memory errors (*Non-memory defects* category), namely resource, command and SQL injections, cross-site scripting and issues related to the use of hard-coded passwords. Such result is because the focus of both Google Sanitizer and RV-MATCH is analysis aiming to detect memory errors and undefined behaviors. The executions exploring these vulnerabilities did not lead to such issues.

The results further indicate that all tools are well equipped for detection of memory-related errors including buffer overflows, null pointer dereferences, use-after-free and similar issues. During analysis of memory-related defects (*Memory Defects* category) E-ACSL detected all seeded defects except for one test case utilizing bad input to `scanf` (*CWE-391*). Google Sanitizer and RV-MATCH detected less issues. For instance, in *CWE-121* both tools missed a defect that uses a fixed size buffer to store user-supplied input via `puts` function. Furthermore, RV-MATCH does not support detection of heap memory leaks when memory is allocated via library functions (such as `strdup`). We should also note that

Table 1. Detection results of E-ACSL, Google Sanitizer and RV-MATCH over SARD-100 test suite

	E-ACSL	Sanitizer	RV-MATCH	Cumulative
Non-memory defects				
CWE-078: Command Injection	0% (0/6)	0% (0/6)	0% (0/6)	0% (0/6)
CWE-080: Basic XSS	0% (0/5)	0% (0/5)	0% (0/5)	0% (0/5)
CWE-089: SQL Injection	0% (0/4)	0% (0/4)	0% (0/4)	0% (0/4)
CWE-099: Resource Injection	0% (0/4)	0% (0/4)	0% (0/4)	0% (0/4)
CWE-259: Hard-coded Password	0% (0/5)	0% (0/5)	0% (0/5)	0% (0/5)
CWE-489: Leftover Debug Code	0% (0/1)	0% (0/1)	0% (0/1)	0% (0/1)
Memory defects				
CWE-121: Stack Buffer Overflow	100% (11/11)	91% (10/11)	91% (10/11)	100% (11/11)
CWE-122: Heap Buffer Overflow	100% (6/6)	100% (6/6)	100% (6/6)	100% (6/6)
CWE-416: Use After Free	100% (9/9)	100% (9/9)	100% (9/9)	100% (9/9)
CWE-244: Heap Inspection	0% (0/1)	0% (0/1)	0% (0/1)	0% (0/1)
CWE-401: Memory Leak	100% (5/5)	80% (4/5)	60% (3/5)	100% (5/5)
CWE-468: Pointer Scaling	50% (1/2)	50% (1/2)	50% (1/2)	50% (1/2)
CWE-476: Null Dereference	100% (7/7)	100% (7/7)	100% (7/7)	100% (7/7)
CWE-457: Uninitialized Variable	100% (4/4)	75% (3/4)	100% (4/4)	100% (4/4)
CWE-415: Double Free	100% (6/6)	100% (6/6)	67% (4/6)	100% (6/6)
CWE-134: Format String	100% (8/8)	0% (0/8)	0% (0/8)	100% (8/8)
CWE-170: String Termination	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
CWE-251: String Management	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
CWE-391: Unchecked Error	0% (0/1)	0% (0/1)	0% (0/1)	0% (0/1)
Concurrency defects				
CWE-367: Race Condition	0% (0/4)	0% (0/4)	0% (0/4)	0% (0/4)
CWE-412: Unrestricted Lock	0% (0/1)	0% (0/1)	0% (0/1)	0% (0/1)
Overall	67% (67/100)	56% (56/100)	54% (54/100)	67% (67/100)

Google Sanitizer missed one memory leak because this tool does not treat memory that has not been freed but available via a global variable as a leak. Finally, Google Sanitizer and RV-MATCH do not support runtime detection of format string vulnerabilities (via standard library functions such as `printf`).

Even though Google Sanitizer and RV-MATCH include functionality allowing to discover concurrency issues (e.g., race conditions), both tools missed a handful of such defects in SARD-100. E-ACSL does not support monitoring of multi-threaded programs.

4.2 Results for Toyota ITC Benchmark

Table 2 shows detection results of E-ACSL, Google Sanitizer and RV-MATCH over programs from Toyota ITC Benchmark. The presentation of the results is similar to that of Table 1 in Sect. 4.1.

Table 2. Detection results of E-ACSL, Google Sanitizer and RV-MATCH over Toyota ITC Benchmark

Defect type	E-ACSL	Sanitizer	RV-MATCH	Cumulative
Dynamic Memory	94% (81/86)	78% (67/86)	94% (81/86)	94% (81/86)
Static Memory	100% (67/67)	96% (64/67)	100% (67/67)	100% (67/67)
Pointer-related	56% (47/84)	32% (27/84)	99% (83/84)	99% (83/84)
Stack-related	35% (7/20)	70% (14/20)	100% (20/20)	100% (20/20)
Resource	99% (95/96)	60% (58/96)	98% (94/96)	100% (96/96)
Numeric	93% (100/108)	59% (64/108)	98% (106/108)	98% (106/108)
Miscellaneous	94% (33/35)	49% (17/35)	71% (25/35)	97% (34/35)
Inappropriate Code	0% (0/64)	0% (0/64)	0% (0/64)	0% (0/64)
Concurrency	0% (0/44)	73% (32/44)	66% (29/44)	93% (41/44)
Overall	71% (430/604)	57% (343/604)	84% (505/604)	87% (530/604)

The results indicate that E-ACSL, Google Sanitizer and RV-MATCH have detected 71%, 57% and 84% of defects respectively. Cumulative error detection ratio over Toyota ITC Benchmark is 87%.

All tools detected most defects related to improper use of dynamic and static memory that include such issues as buffer over- and underflows (top two rows of Table 2). 78% detection rate in *Dynamic* defects of Google Sanitizer is because this tool could not detect a number of heap buffer-underruns where negative offsets were used to access unallocated memory.

In *Pointer-related* defects E-ACSL detected 56% of bugs. The tool could not identify defects related to improper use of function pointers. Detection ratio of Google Sanitizer is 32%. This result is the lowest of all tools in the *Pointer-related* category. This tool had issues detecting defects related to passing a null pointer to `free`, incorrect pointer arithmetic and the use of function pointers.

RV-MATCH detected 99% of errors related to improper use of pointers. This tool missed only one defect related to using an uninitialized pointer.

Stack-related defect type includes three defect sub-types: stack overflow, cross-thread access and static buffer overrun. E-ACSL has little support for detecting stack overflows and cannot monitor multi-threaded programs. The detection rate of E-ACSL in this case is only 35%. Google Sanitizer and RV-MATCH provide better support and identified 70% and 100% of defects respectively.

Resource Management defects of Toyota ITC Benchmark contain such issues as double free, freeing non-dynamic memory, return of local addresses and memory leaks. For this vulnerability type Google Sanitizer has the lowest detection ratio of 60%. This tool has missed several bugs related to freeing static memory and returning local variables. E-ACSL on the contrary has been able to identify most such defects (99%). RV-MATCH has similar result of 98%.

E-ACSL and RV-MATCH detected most errors of *Numeric* defect type (integer overflows, bit-field overflows, division by zero), 93% and 98%. Both tools missed defects involving floating point overflow. Additionally, E-ACSL failed to detect overflows via bit-field values. Google Sanitizer has detected fewer defects (59% via UndefinedBehaviourSanitizer). This tool had issues detecting errors involving integer precision loss because of cast and loss of integer sign because of unsigned casts.

Miscellaneous defects of Toyota ITC Benchmark describe endless loops, invalid extern variable declarations, missing return statements and similar issues. E-ACSL detected 94% of such issues. Detection ratio for Google Sanitizer and RV-MATCH were lower – 49% and 71% respectively.

All tools missed all defects of the *Inappropriate Code* type. Such issues consist of mostly syntactic and stylistic issues such as left-over debug code, specifying same condition twice and so on. Even though they contribute to the overall score such defects do not lead to undefined behaviours or memory errors and can hardly be regarded as security-related.

Finally, E-ACSL missed all defects of *Concurrency* type. As noted earlier E-ACSL does not support detection of such issues. Google Sanitizer and RV-MATCH detected 73% and 66% respectively. A notable result is that these tools miss defects of different subtypes and the detection cumulative rate over both tools is 93%.

4.3 Summary of Results

The overall results of our experiment show that the present state-of-the-art runtime verification tools for C programs provide strong support for detection of issues stemming from improper use of memory and undefined behaviours. On the other hand the results indicate lack of support for such important security flaws as command injections that remain one of the most dangerous software bugs [7] and still found even in well-tested C applications [33].

With respect to **RQ1** that considers the cumulative capacity of error detection, we can conclude that in tracking non-concurrent memory errors using one

tool may be sufficient to detect most defects. This is supported by the results for SARD-100 and Toyota ITC Benchmark that show consistently high detection ratios for typical memory errors such as buffer overflows, double free violations, null pointers dereferences, use of initialized values and so on. Regarding **RQ3**, the results indicate that combining the results of all tools one can achieve higher detection ratio. For instance, in detection of resource-management memory defects of Toyota ITC Benchmark, E-ACSL detected 99% of all defects yet the cumulative detection ratio is 100%. Further, while E-ACSL detected 94% defects of miscellaneous type of Toyota ITC Benchmark, the combined result (with RV-MATCH) is 97%. The most interesting result is for detection of concurrency issues. Notably, the detection ratios of both Google Sanitizer and RV-MATCH are relatively low – 73% and 66%, cumulative result however is 93%, almost all seeded defects of that type.

Per-tool analysis of results (see **RQ2**) shows that for detection of memory-related vulnerabilities in single-threaded executions E-ACSL shows superior performance. One reason for such result is the tool’s memory tracking model that allows tracking bounds of memory blocks and identify more vulnerabilities involving illegal memory accesses. E-ACSL is also the only tool that enables runtime analysis for format string vulnerabilities. On the other hand both Google Sanitizer and RV-MATCH show good scores for detecting defects in multi-threaded executions (e.g., deadlocks and race conditions). E-ACSL has no support for such analysis. Furthermore, both tools have also shown better support for detecting improper use of function pointers and stack overflows.

5 Threats to Validity

We now discuss issues that might have affected validity of the experiment presented in this paper.

The first issue refers to the choice of source code benchmarks used in evaluating precision of runtime verification tools. We aimed to select representative code covering a broad range of defects typically leading to security vulnerabilities in C. Different choice of programs might affect the results. For instance, neither SARD-100 nor Toyota ITC Benchmark explores issues related to information flow leakage in its full generality. Since the analyzers used during the experiment have no support for such analysis the precision results of all tools could be lower if such issues were present.

Another issue refers to the choice of the runtime verification tools used in the experiment. We aimed to select popular online monitoring tools capable of detecting typical security issues occurring in C program with little manual effort. However, we cannot claim that our tool selection was representative for detecting security vulnerabilities. For instance, as shown by the experiment E-ACSL, Google Sanitizer and RV-MATCH have good support for detecting defects related to the use of memory and undefined behaviours but do not support detection of SQL or command injections. Furthermore, we might have overlooked some of the tools that address similar issues and also freely available and easy to use.

The third issue refers to generating monitored programs using Google Sanitizer. This tool has a number of compile- and runtime options affecting the results of its analysis. Even though we tried to study this tool and use all documented features there might be some options that we overlooked. Furthermore, Google Sanitizer is under active development and we used the latest released version. It is therefore possible that the latest development version of Google Sanitizer has a different precision over the same code samples.

The fourth issue refers to collecting results. During our experimentation we discovered several bugs in programs from SARD-100 and Toyota ITC Benchmark. To ensure the correctness of the precision results we manually verified that programs under analysis are correct (i.e., contain defects of the claimed types and the provided inputs lead to execution of erroneous path). However, as we had to deal with large number of defects (over 700) we might have overlooked some of the issues that might have also affected the final results.

Finally, the authors of this paper have been involved in design and the development of the E-ACSL runtime verification tool. While we did our best to stay impartial and aimed at providing a fair and unbiased study we might have had the *developers' advantage* when reasoning about the results produced by E-ACSL.

6 Conclusions and Future Work

This experience report provides a global view of the capacity of modern runtime verification tools to detect security vulnerabilities with an emphasis on memory errors. We consider different approaches – a formal semantic based tool, a formal specification verifier and a memory debugger – in order to evaluate the cumulative detection power of these tools used together. We have presented the experimental protocol, the selected tools and benchmarks, and provided and analyzed the recorded results. Detailed results are available online and can be used for a more detailed analysis. They indicate the level of support by the selected tools for various kinds of issues. Overall, the cumulative detection rate of the three selected tools over all defects of the considered benchmark suites is 84%. Although detection rates achieve highest values for several categories of errors, we observed that applying several tools appears to be beneficial for detecting several categories of issues. For instance, in detecting concurrency issues in the Toyota ITC Benchmark the highest per-tool result is 73%, whereas the cumulative rate is 93%. Future work includes experiments with other categories of tools, using larger benchmark suites of security related issues, as well as further analysis of failures and improvement of their support in the compared tools.

Acknowledgments. The authors thank the Frama-C team for providing the tools and support. Many thanks to the anonymous referees for their helpful comments.

References

1. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the USENIX Security Symposium, pp. 51–66. USENIX Association, August 2009
2. Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 1–9. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_1
3. Black, P.E., Kass, M., Koo, M., Fong, E.: Source code security analysis tool functional specification version 1.1. Technical report 500–268 v1.1, Information Technology Laboratory (2011)
4. Bruening, D., Zhao, Q.: Practical memory checking with Dr. Memory. In: Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2011, pp. 213–223. IEEE Computer Society, Washington, DC (2011)
5. ISO/IEC 9899:2011. <https://www.iso.org/standard/57853.html>
6. Texas Instruments Center: Specification of test bench (2015). <https://github.com/regehr/itc-benchmarks>
7. Christey, S.: 2011 CWE/SANS top 25 most dangerous software errors. Technical report 1.0.3, The MITRE Corporation, September 2011. <http://www.mitre.org>
8. Common Weakness Enumeration: A community developed dictionary of software weakness types. <http://cwe.mitre.org>
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages (POPL 1977), pp. 238–252 (1977)
10. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Proceedings of the ACM Symposium on Applied Computing, pp. 1230–1235. ACM, March 2013
11. Delaitre, A.: Test Suite #100: C test suite for source code analyzer v2 - vulnerable (2015). <https://samate.nist.gov/SRD/view.php?tsID=100>
12. Dhurjati, D., Adve, V.S.: Backwards-compatible array bounds checking for C with very low overhead. In: Proceedings of the International Conference on Software Engineering, pp. 162–171. ACM, May 2006
13. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
14. Eigler, F.C.: Mudflap: pointer use checking for C/C++. In: Proceedings of the GCC Developers Summit, pp. 57–70, May 2003
15. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. SIGPLAN Not. **47**(1), 533–544 (2012)
16. Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. In: de Bakker, J., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 169–181. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10003-2_69
17. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Engineering Dependable Software Systems, pp. 141–175 (2013)
18. Falcone, Y., Ničković, D., Reger, G., Thoma, D.: Second international competition on runtime verification. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 405–422. Springer, Cham (2015). <https://doi.org/10.1007/s10009-017-0454-5>

19. Guth, D., Hathhorn, C., Saxena, M., Roşu, G.: RV-match: practical semantics-based program analysis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 447–453. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_24
20. Hastings, R., Joyce, B.: Purify: fast detection of memory leaks and access errors. In: Proceedings of the Winter USENIX Conference, pp. 125–136, January 1992
21. ISO/IEC 9899:1999, 1430 Broadway, New York, NY 10018, USA: Programming languages - C, March 2013. www.open-std.org/jtc1/sc22/wg14/www/standards
22. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: a safe dialect of C. In: Proceedings of the General Track: 2002 USENIX Annual Technical Conference, pp. 275–288. USENIX, June 2002
23. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proceedings of the International Workshop on Automatic Debugging, pp. 13–26. Linköping University Electronic Press, September 1997
24. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. *Formal Aspects Comput.* **27**(3), 573–609 (2015)
25. Matsakis, N.D., Klock II, F.S.: The Rust language. *ADA Lett.* **34**(3), 103–104 (2014)
26. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) *Programming 1982*. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_22
27. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. *J. Logic Algebraic Program.* **79**(6), 397–434 (2010)
28. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proceedings of the Network and Distributed System Security Symposium. The Internet Society, December 2004
29. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: a fast address sanity checker. In: Proceedings of the USENIX Annual Technical Conference, pp. 309–319. USENIX Association, June 2012
30. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic race detection with LLVM compiler. In: Khurshid, S., Sen, K. (eds.) *RV 2011*. LNCS, vol. 7186, pp. 110–114. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_9
31. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the USENIX Annual Technical Conference, pp. 17–30. USENIX (2005)
32. SGCheck: an experimental stack and global array overrun detector. <http://valgrind.org/docs/manual/sg-manual.html>
33. Vulnerability summary for CVE-2014-6271. National Institute of Standards and Technology: National Vulnerability Database, September 2014. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271>
34. Shiraishi, S., Mohan, V., Marimuthu, H.: Test suites for benchmarks of static analysis tools. In: IEEE International Symposium on Software Reliability Engineering Workshops, pp. 12–15. IEEE (2015)
35. Simpson, M.S., Barua, R.: MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Softw.: Pract. Exp.* **43**(1), 93–128 (2013)
36. Standard Performance Evaluation Corporation: SPEC CPU (2006). <http://www.spec.org/benchmarks.html>

37. Stepanov, E., Serebryany, K.: MemorySanitizer: fast detector of uninitialized memory use in C++. In: Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 46–55. IEEE Computer Society, February 2015
38. Undefinedbehaviorsanitizer (2017). <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>
39. Projects using Valgrind (2017). <http://valgrind.org/gallery/users.html>
40. van der Veen, V., dutt-Sharma, N., Cavallaro, L., Bos, H.: Memory errors: the past, the present, and the future. In: Balzarotti, D., Stolfo, S.J., Cova, M. (eds.) RAID 2012. LNCS, vol. 7462, pp. 86–106. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33338-5_5
41. Vorobyov, K., Kosmatov, N., Signoles, J., Jakobsson, A.: Runtime detection of temporal memory errors. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 294–311. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_18
42. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: Proceedings of the International Symposium on Memory Management, pp. 47–58. ACM, June 2017

Tool Demonstration and Short Papers



Formalizing (Web) Standards

An Application of Test and Proof

Achim D. Brucker^(✉)  and Michael Herzberg 

Department of Computer Science, The University of Sheffield, Sheffield, UK
{a.brucker, msherzberg1}@sheffield.ac.uk
<http://www.brucker.ch/>, <https://www.mherzberg.de/>

Abstract. Most popular technologies are based on informal or semi-formal standards that lack a rigid formal semantics. Typical examples include web technologies such as the DOM or HTML, which are defined by the Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C). While there might be API specifications and test cases meant to assert the compliance of implementations, the actual standard is rarely accompanied by a formal model that would lend itself for, e. g., verifying the security or safety properties of real systems.

Even when such a formalization of a standard exists, two important questions arise: first, to what extent does the formal model comply with the standard and, second, to what extent does a concrete implementation comply with the formal model and the assumptions made during the verification of certain properties?

In this paper, we present an approach that brings all three involved artifacts—the (semi-)formal standard, the formalization of the standard, and the implementations—closer together by combining verification, symbolic execution, and specification-based testing.

Keywords: Standard compliance · Compliance tests · DOM

1 Introduction

Most popular technologies are only specified by standards using a semi-formal or, worse, an informal notation. Moreover, the tools used for writing standards only support, if at all, trivial consistency checks. Thus, it is no surprise that such standards usually contain inconsistencies (e. g., different sections of the same standard that contradict each other) or unwanted under-specifications (e. g., where the authors of the standard omit the specification of important properties that, e. g., the defined API should fulfill).

Even if a standard is developed formally, or contains a (often non-normative) formalization, two important questions arise: 1. to what extent does the formal model comply with the semi-formal parts of the standard, and 2. to what extent does an actual implementation comply with the formal model? If the formal

model was used for verifying properties, one also needs to validate that the real system fulfills the assumptions made during the verification.

Neither the problem of glitches and inconsistencies of standards nor the use of testing for showing compliance of implementations are new (see, e. g., [1, 7, 8] for examples of formalizations of standards, outside the Web domain, and the use of testing for showing the compliance of implementations.) Still, most standard development today is based on semi-formal specifications. Prominent examples of such a semi-formal standard development are the standards for the common web technologies, such as the Document Object Model (DOM) or HTML. Both are defined by the Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C). These web standards are developed in an open process, e. g., everybody can read and comment on upcoming versions of the standard, and they usually include type-checked interfaces for the defined APIs. These interfaces are specified in Web IDL [11]. Moreover, these standards are complemented by a *manually* defined *compliance* test suite that can be used by developers to check their implementation. Additionally, due to the manual process of developing the compliance tests, their quality mostly depends on expert knowledge and their quality varies greatly, depending on who wrote the test cases.

In this paper, we present an approach that brings all three involved artifacts—the standard, the formalization of the standard, and the implementations—closer together by combining verification, symbolic execution, and specification-based testing. Moreover, we report on a case study applying this approach to the Document Object Model (DOM) standard [10, 13] that specifies *the* central data structure of all modern web browsers as well as algorithms for querying and updating the DOM. Our case study is based on the official DOM standard, the compliance test suite provided by the authors of the standard (which is used by browser vendors to show that their browsers faithfully implement the standards), and our own formalization of the standard [3, 4] in Isabelle/HOL [9].

The rest of the paper is structured as follows: in Sect. 2 we present our approach for linking formal and informal parts as well as implementations using test and proof. In the next section (Sect. 3) we report on our experience in applying our approach to the DOM standard [13]. We conclude in Sect. 4.

2 Using Test and Proof for Formalizing Standards

In this section, we present an approach using and combining *test and proof* for providing strong links between semi-formal standards (and their compliance test suites) and a formalization. Figure 1 illustrates the overall scenario for both traditional development of standards (upper part of the figure) and the integration of “test and proof”-activities (bottom part).

First, let us recall the process and challenges of developing informal or semi-formal standards and implementations that should comply with such a standard: most standards are developed as a text document that contains technical details,

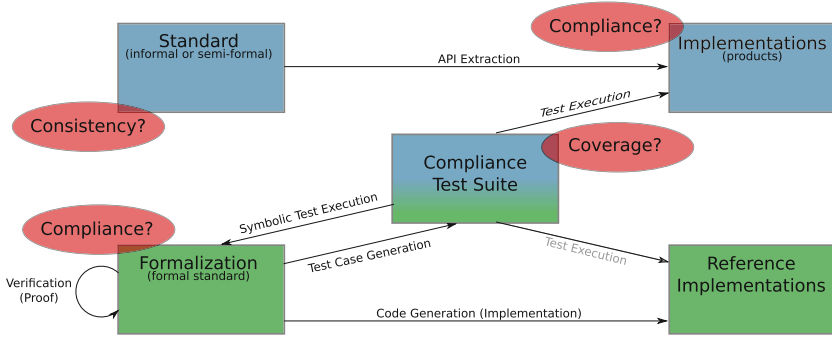


Fig. 1. Using test and proof for establishing strong links between formal standards, compliance test suites, and implementations.

e. g., in the form of interface specifications or pseudo-code, that implementations need to comply with. Such semi-formal or informal standards usually contain many inconsistencies; tool support for ensuring the *syntactic consistency* of the standard is sometimes available in a limited form, but the *semantic consistency* is an open problem. Also, linking standards to implementations is, in the best case, only supported by the possibility to automatically extract interface definitions (APIs), if the standard defines a (software) system. Alternatively, if the standard defines a data format (or a language) it might possible to extract grammar definitions for the abstract or concrete syntax of the defined data format or language. A good standard also includes an extensive set of compliance test cases. These compliance test cases are usually specified manually by experts. Hence, manually developed test cases cannot guarantee to *cover* all important cases and, thus, they can only provide a weak *compliance*-relationship between standard and implementation. Nevertheless, they are the only machine-checkable artifact for vendors to validate the compliance of their product to the standard.

Second, let us discuss how *test and proof* can improve the situation and address the consistency and compliance challenges of semi-formal and informal standard development. In the following, we assume that an executable formalization (e. g., expressed in Isabelle/HOL) of the standard exists. Of course, if we start with an informal standard, the question arises to which extent the formalization is a faithful representation of the informal (or semi-formal) standard, i. e., the *compliance* of the formalization. As we assume an executable model, we can—similarly to implementations—use *symbolic execution* to show the compliance of the formal model to the semi-formal standard (or, more precisely, the manually developed compliance test suite). In addition, we can use the formal model to actually *prove* important properties of the standard (e. g., proving the correctness of the algorithms presented in the standard). We can also generalize test cases provided in the compliance test suite and turn them into proof obligations for our formal model. Using *symbolic specification-based* test case generation techniques (e. g., as presented in [6]), we can automatically generate

new compliance test cases that, e. g., guarantee branch coverage on the level of the specification. Finally, we could generate a reference implementation using code generators available in systems such as Isabelle [9] or Coq [2].

3 Case Study: The Document Object Model (DOM)

We successfully applied this approach to our formal model [3,4] of the DOM standard [13]. This increases the confidence that our formalization faithfully represents the official standard.

3.1 Formalizing the DOM Standard

We illustrate our approach using the `insertBefore` method as an example. The interface of `insertBefore` is given in Web IDL [11]:

```
interface Node {
  Node insertBefore(Node node, Node? child);
}
```

The behavior of this method is described using structural English:

insertBefore:
The `insertBefore(node, child)` method, when invoked, must return the result of *pre-inserting* node into *context object* before child.

This descriptions refers, using hyperlinks, to the concepts *pre-inserting* and *context object*. Without a clear understanding of these concepts, we cannot formalize `insertBefore`. The concept *pre-inserting* is described as follows:

pre-insert:
To pre-insert a node into a parent before a child, run these steps:
 1) Ensure *pre-insertion* validity of node into parent before child.
 2) Let reference child be child.
 3) If reference child is node, set it to node's *next sibling*.
 4) *Adopt* node into parent's *node document*.
 5) *Insert* node into parent before reference child.
 6) Return node.

Again, several new concepts are introduced and to fully understand the behavior of `insertBefore`, we need to understand and formalize these concepts as well. We formalize the `insertBefore` using monads in Isabelle/HOL:

```
definition insert_before :: "_ object_ptrCore_DOM ⇒ _ node_ptrCore_DOM
  ⇒ _ node_ptrCore_DOM option ⇒ _ dom_prog"
where
  "insert_before ptr node child = do {
    ensure_pre_insertion_validity node ptr child;
    reference_child ← (if Some node = child
      then next_sibling node
      else return child);
    owner_document ← get_owner_document ptr;
    adopt_node owner_document node;
    insert_node ptr node reference_child
  }"
```

3.2 Showing Standard Compliance

While our formalization tries to stay as close as possible to the description in the standard, it is not obvious that it complies to it. To show this compliance, we first selected all relevant test cases from the official DOM compliance test suite [12], i. e., the test suite used by web browser vendors to show that their DOM implementation complies to the standard. These test cases are written in JavaScript, which is embedded into the DOM document under test. We then automatically translated these tests into higher-order logic (HOL) to *symbolically execute* (the test cases can be “evaluated” by Isabelle’s simplifier using a set of simplifier rules optimized for code generation) them on our model of the DOM. For example, consider the following test case (the left-hand site shows the official specification in JavaScript, the right-hand site our formalization in Isabelle/HOL):

<pre>test(function() { var a = document.createElement('div'); var b = document.createElement('div'); var c = document.createElement('div'); assert_throws('NotFoundError', () => { a.insertBefore(b, c); }); }, 'Calling insertBefore with a reference + child whose parent is not the context + node must throw a NotFoundError.')</pre>	<pre>lemma "test (do { a ←document.createElement('div'); b ←document.createElement('div'); c ←document.createElement('div'); assert_throws(NotFoundError, (cast a).insertBefore(cast b, Some (cast c))) }) Node_insertBefore_heap" by code_simp (* 'Calling insertBefore with a reference child whose parent is not the context node must throw a NotFoundError.' *)</pre>
--	--

This test checks whether the DOM method `insertBefore` throws a certain exception if called with a certain combination of arguments. We formalized this test into a state-exception-monad and show the error-freeness by symbolic execution.

Tests are, of course, a very limited way of showing such important properties, as they only show the property for concrete input values (here, a simple DOM instance). To overcome this limitation, we generalize such test cases in to generic theorems that show the corresponding property for all possible inputs. In our example, we generalize the test into the following theorem, which we prove formally in Isabelle/HOL:

```
lemma insert_before_non_child_reference_node:
  assumes "heap_is_wellformed h" and "is_known_ptrCore_DOM ptr"
  and "¬ (h †reference_child.parentNode →r Some element)"
  and "¬ (is_character_data_ptr element)"
  and "∧ancestors. h †get_ancestors element →r ancestors
  ⇒ cast new_child †set ancestors"
  shows "h †element.insertBefore(new_child, Some reference_child)
  →e NotFoundError"
```

Instead of creating three concrete elements, we can quantify over all possible elements. The two assumptions give additional insight; the test would fail if the argument were a `CharacterData` or included in the reference’s ancestors, because these circumstances are checked earlier and cause different exceptions.

Using this approach, we formalized all non-type-related test cases from the official test suite to “test” our model. Table 1 shows the number of formalized

Table 1. The number of tests regarding our supported DOM methods that are available from the official suite and *not* related to type checks. Additionally, we present a rough estimate of the complexity of the tested function along with the coverage of the tests to estimate how much each function would benefit from automatically generated tests.

	# Test cases in scope	Function complexity	Function coverage
assignedNodes	24	High	High
assignedSlot	24	High	High
insertBefore	5	High	Low
getElementById	10	Medium	Medium
removeChild	8	Medium	Medium
attachShadow	2	Medium	Medium
createElement	49	Medium	Low
adoptNode	2	Medium	Low
getRoot	3	Medium	Low
childNodes	2	Low	Medium
parentNode	3	Low	Medium
shadowRoot	2	Low	Low
host	1	Low	Low
getOwnerDocument	0	Low	–
getAttribute	0	Low	–
setAttribute	0	Low	–
nextSibling	0	Low	–

tests per DOM function that we support. We cannot easily utilize test cases regarding type checks, as we decided to formalize a strongly typed model. The official compliance test suite contains many typing-related tests, mainly due to two reasons:

1. Dynamic typing and prototype-based inheritance of JavaScript leads to many tests that, for example, check the behavior of functions when passed `null` or `undefined`, whereas we in HOL only allow `None` in places where the DOM standard actually permits it.
2. We model a simplified version of the core DOM. We turned many classes that extend the `Node` interface and, thus, participate in the node tree, into attributes of other interfaces. For example, the DOM standard defines `DocumentType` as a node that must appear in exactly one location of the node tree—it must be the first child of a `Document`. We model the document type as a field of a `Document`. Many tests of the official suite test that constraint, which we therefore did not formalize.

The official test suite is developed manually and, thus, it is not surprising that the test cases vary in style and quality. For example, the compliance test for

the tree-modifying method `insertBefore` consists of 26 test cases, of which only five are relevant for our formalization. This indicates that the test authors' concern is mostly testing the absence of run time errors and, to a lesser extent, the correctness of this rather complex method.

3.3 Formal Verification: Analyzing the Standard

In many cases, the methods defined in the DOM standard need to fulfill important properties. These properties are neither spelled out explicitly nor does the compliance test suite contain test cases for them. During the formalization of the standard, these properties often emerge as proof obligations that need to be shown to be able to prove the high-level properties specified in the standard.

An example for such an important property is that after a successful call of `insert_before`, the list of child nodes remains distinct, even if the new child was already a child of that node:

```

lemma insert_before_children_remain_distinct:
  assumes "heap_is_wellformed h" and "is_known_ptrCore_DOM ptr"
  and " $\wedge$ parent. h  $\vdash$ get_parent new_child
         $\rightarrow_r$  Some parent  $\implies$ is_known_ptrCore_DOM parent"
  and "h  $\vdash$ insert_before ptr new_child child_opt  $\rightarrow_h$  h2"
  shows " $\wedge$ ptr children. is_known_ptrCore_DOM ptr
         $\implies$ h2  $\vdash$ get_child_nodes ptr  $\rightarrow_r$  children
         $\implies$ distinct children"

```

This is true because `insert_before` first removes the new child from its old parent before inserting it into the child node list of the new parent.

While the verification as such is important to ensure the consistency and implementability of the standard, it also forms the basis for developing an improved compliance test suite. Using a specification-based or theorem prover-based test-case generation approach [6], the proven lemmas can be systematically turned into additional compliance test cases that ensure that actual implementations fulfill these crucial properties.

4 Conclusion

We reported on a first case study combining test and proof for formalizing a standard that is the core of modern web-browsers. We can show the compliance of our formal model to the standard by symbolically executing the official compliance test suite. Our manual analysis of this test suite revealed several important properties that not sufficiently covered, or not covered at all, by the compliance test suite.

As future work, we plan to automatically generated test cases from our formal model (e.g., using HOL-TestGen [5]) and to contribute them to the official compliance test suite. We also plan to enrich our model with a security model formalizing common web-related security measures to verify and test the security guarantees of modern web browsers (and applications running on top of them).

References

1. Feo-Arenis, S., Westphal, B., Dietsch, D., Muñiz, M., Andisha, A.S., Podelski, A.: Ready for testing: ensuring conformance to industrial standards through formal verification. *Formal Asp. Comput.* **28**(3), 499–527 (2016). <https://doi.org/10.1007/s00165-016-0365-3>
2. Bertot, Y., Castéran, P.: *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, Heidelberg (2004). <https://doi.org/10.1007/978-3-662-07964-5>
3. Brucker, A.D., Herzberg, M.: The core DOM. *Archive of Formal Proofs* (2018, submitted). http://www.isa-afp.org/entries/Core_DOM.shtml. Formal proof development
4. Brucker, A.D., Herzberg, M.: A formal semantics of the core DOM in Isabelle/HOL. In: *WWW 2018 Companion: The 2018 Web Conference Companion*. ACM Press (2018). <https://doi.org/10.1145/3184558.3185980>
5. Brucker, A.D., Wolff, B.: HOL-TestGen: an interactive test-case generation framework. In: Chechik, M., Wirsing, M. (eds.) *FASE 2009*. LNCS, vol. 5503, pp. 417–420. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00593-0_28
6. Brucker, A.D., Wolff, B.: On theorem prover-based testing. *Formal Aspects Comput.* **25**(5), 683–721 (2013). <https://doi.org/10.1007/s00165-012-0222-y>
7. Horl, J., Aichernig, B.K.: Validating voice communication requirements using lightweight formal methods. *IEEE Softw.* **17**(3), 21–27 (2000). <https://doi.org/10.1109/52.896246>
8. Kristoffersen, F., Walter, T.: TTCN: towards a formal semantics and validation of test suites. *Comput. Netw. ISDN Syst.* **29**(1), 15–47 (1996). [https://doi.org/10.1016/S0169-7552\(96\)00016-5](https://doi.org/10.1016/S0169-7552(96)00016-5)
9. Nipkow, T., Paulson, T.C., Wenzel, M.: *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
10. W3C: W3C DOM4 (2015). <https://www.w3.org/TR/dom/>
11. W3C: Web IDL (2017). <https://heycam.github.io/webidl/>
12. W3C: Web platform test: DOM. <https://github.com/w3c/web-platform-tests/tree/master/dom>. Accessed 10 Nov 2017
13. WHATWG: DOM - living standard (2017). <https://dom.spec.whatwg.org/commit-snapshots/6253e53af2fbfaa6d25ad09fd54280d8083b2a97/>. Accessed 24 Mar 2017



Automated Test Case Generation for Java EE Based Web Applications

Andreas Fuchs^(✉)

Department of Information Systems, University of Münster,
Leonardo-Campus 3, 48149 Münster, Germany
andreas.fuchs@wi.uni-muenster.de
<https://www.wi.uni-muenster.de/>

Abstract. Automated testing is important for validating the behavior of programs with complex user interfaces, such as web applications. In the enterprise context, web applications are popular client-server programs that provide rendered web pages as a user front-end, and the business logic is typically implemented on the server-side. In this paper, we present an approach to automatically generate test cases for component-based user interfaces for web applications built on the Java EE platform. We generate a sequence of user actions to navigate through the web application. For each supported user action, we gather constraints from the view template describing the web page (e.g. a button must be enabled in order to be clicked by a user), as well as constraints that are introduced while executing a server-side component. We have implemented our approach in a tool to determine its practical use in an experiment.

Keywords: Automated test case generation · Web applications
Symbolic execution · Java Enterprise Edition

1 Introduction

Many modern enterprise organizations use web applications as a front-end to provide a user-friendly access to business functionalities. Such an application can be characterized as a client-server program, in which the client runs in a web browser and displays the user interface. The displayed content of these applications is typically generated dynamically, e.g. because they depend on the state of other systems such as database systems. The connection to these systems is established on the server-side. A client sends a request for a page via the Hypertext Transfer Protocol (HTTP) to a server, and the server generates a rendered web page (e.g. in HTML) that is sent as a response back to the client.

There are many tools and frameworks that support the development of web applications, and Java is one of the most popular programming languages [3]. The Java Enterprise Edition (Java EE) is a platform that supports the development of enterprise software systems by providing a great set of application programming interfaces (APIs). The JavaServer Faces (JSF) are part of one of these APIs. JSF

is a specification for building user interfaces for JavaServer web applications¹. In its latest version, it uses *Facelets* as a templating system that reflects the views of a JSF application. Facelets are XML-based documents that describe the user interface (UI) of the web pages. Facelets “extend” the traditional plain HTML document with a more powerful set of definitions. For instance, it can be defined that clicking on a button in a web page triggers the invocation of Java method that is implemented in an Enterprise JavaBean (EJB) on the server-side, or that data displayed in the rendered web page are the result of a method invocation of an EJB. Figure 1 shows an abstract overview of a client that requests a web page (`view.xhtml`). Before the web page can be rendered to plain HTML, an EJB named `dataBean` is used to query some data of a connected database (DB). The rendered web page is then sent back to the client, which can display it to the user in a browser.

Testing a software program can provide information about the quality of the tested program. Often, this includes the execution of the program with the intent of finding defects in the software [12]. Test case generation is one of the most labor-intensive tasks in software testing, and as a result an automation of that task has been the subject to an intensive research effort [1].

In this paper, we propose an approach to automatically generate executable test cases for JSF web applications. We generate a sequence of user actions that navigate through the web application, and during this generation we take constraints both from the JSF web page, as well as from invoked EJB methods into account. We have implemented our approach in a tool that has two main components: (1) a JSF web application analyzer including a *walker* that simulates a user interacting with the application, and (2) a prototype of a symbolic execution system for Java EE programs. Both components can share their constraint stacks as well as their symbolic heap representation.

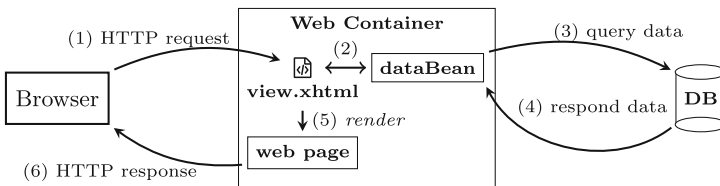


Fig. 1. A client interacting with a JSF application.

Contributions. In summary, this paper makes the following contributions:

- We present an approach for an automated generation of test cases for JSF web applications, that symbolically execute all invoked EJB methods.
- We have implemented the approach in a tool that is available as open source on a public repository [5].

¹ <https://javaee.github.io/javaxserverfaces-spec/>, accessed March 2018.

- We show that our tool generates executable test cases that simulate a use navigating through the web application.

Outline. The rest of the paper is organized as follows. In the next section, we motivate our approach with an example and explain the key components of our tool. Section 3 presents a formal definition of our approach. In Sect. 4, we present experimental results and statistics showing that our tool generates executable test cases. We present related work in Sect. 5 and conclude the paper in Sect. 6.

2 Motivating Example

We motivate our approach with an example web application² for maintaining issues. Figure 2 shows three rendered pages of that application that (1) lists all existing issues, (2) edit an existing issue, and (3) create a new issue.

(1) *View Issues*

Create Issue

ID	Author	Title	Status	Action
1	John	Issue1	OPEN	edit archive
2	Max	Issue2	OPEN	edit archive
3	Lisa	Issue3	ARCHIVED	edit

(2) *Edit Issue*

View Issues Create Issue

ID: 1

Author: John

Title*: Issue1

Status*: OPEN

submit cancel

(3) *Create Issue*

View Issues

Author*:

Title*:

submit cancel

Fig. 2. Rendered pages of the example web application.

The *View Issues* page has a link to navigate to the *Create Issue* page on the top. Additionally, it has a table that shows all three issues that exists so far. For each issue, the user can click either on a button labeled *edit* which navigates to the *Edit Issue* page, or on an button labeled *archive* to change the status of that issue to *ARCHIVED*, though this button is only available for issues that have not yet been archived. On the *Edit Issue* page, a user can change the title and status of an existing issue by clicking on the button labeled *submit*, or cancel the editing by clicking *cancel*. The page has also two links that navigate to the *View Issues* and to the *Create Issue* page respectively. On the *Create Issue* page, a user can insert the author name and the title of a new issue. Both fields are required. The constraints (e.g. a button is rendered based on the value of an entry in a table) are either defined in the XHTML file, or introduced while executing a method of a backing bean. In our approach, we support both sources of constraints.

Figure 3 shows a part of the XHTML code of the *View Issues* page from Fig. 2. We parse the XHTML files with an ANTLR parser [13]. Line 1 defines a link to navigate to the *Create Issue* page. In Line 3 the data table of that page is defined. The table gets its values as a result of the invocation of the method

² The application *issue-tracker* is available as open-source on a public repository [5].

`getAllIssues` from a bean named `requestBean`. The relevant method is shown in Fig. 4a, and it returns a list λ of objects of type `Issue`. In order to click on a button labeled `edit` (see Line 11), there must be at least one element e in the list, and we identify the constraint $\lambda.\text{length} \geq 1$. In order to click on the button labeled `archive` (see Line 13), that element must also have a status that is equal to `OPEN`, and thus we identify the constraint $e.\text{status} = s$, with s being an object-reference to a string constant with the value “OPEN”.

```

1 <h:link outcome="create.xhtml" value="Create Issue" />
2 ...
3 <h:dataTable value="#{requestBean.allIssues}" var="issue" ...
4   <h:column>
5     <f:facet name="header">ID</f:facet >
6     <h:outputText value="#{issue.id}" />
7   </h:column>
8   ...
9   <h:column>
10    <f:facet name="header">Actions</f:facet >
11    <h:commandButton value="edit"
12      action="#{issueManager.editIssue(issue)}" />
13    <h:commandButton value="archive"
14      rendered="#{issue.status == 'OPEN'}"
15      action="#{issueManager.archive(issue)}" />
16  </h:column>
17  ...

```

Fig. 3. A data-table definition in JSF, getting data from a backing bean.

<pre> 1 @Stateless @Named("requestBean") 2 public class RequestBean { 3 @PersistenceContext 4 EntityManager em; 5 6 List<Issue> getAllIssues() { 7 return em.createQuery(8 "FROM Issue", Issue.class) 9 .getResultList(); 10 } 11 12 void archive(long id) { 13 Issue issue = em.find(Issue. 14 class, id); 15 issue.setStatus("ARCHIVED"); 16 } 17 ... </pre>	<pre> 1 @SessionScoped 2 @Named("issueManager") 3 public class IssueManager { 4 @Inject RequestBean reqBean; 5 6 String archive(Issue i) { 7 reqBean.archive(i.getId()); 8 return "view.xhtml"; 9 } 10 11 String edit(Issue i) { 12 reqBean.edit(i); 13 return "view.xhtml"; 14 } 15 ... </pre>
--	--

(a) An EJB named `requestBean` that gets data from a database.

(b) An EJB named `issueManager` that navigates to a page `view.xhtml` when an `issue` has been archived.

Fig. 4. EJBs used in the web application that introduce additional constraints.

When a user clicks on the button `archive`, the method `archive` of the bean named `issueManager` is invoked with an argument `issue` that is the current object of type `Issue` displayed in the current row of the data table. The method

is shown in Fig. 4b. It calls the method `archive` of the injected EJB `requestBean` (see Line 12 in Fig. 4a), which then updates the status of the given object `issue` to `ARCHIVED`. As a result, the button labeled `archive` in the `View Issues` is not rendered for that row any more. Hence, the invocation of EJB methods strongly influence the navigation through the web application and must be considered when generating test cases. In the next section, we describe our test case generation approach in more detail.

3 Proposed Approach

Our approach requires access to the compiled Java bytecode of the EJBs used in the web application, as well as to the XHTML files that define the web pages of the application. We generate a sequence of user actions starting in an initial state at a given start page. A state $q = (\omega, db, \mathcal{C}, h)$ is defined as a current web page ω , a database state db , a set of constraints \mathcal{C} , and a heap state h . We represent db similar to the approach presented in [6] with a set of required entities that must exist before a test case is executed, as well as a set of entities that exist after test execution.

We illustrate our approach with the example application of Fig. 2, and we start at the page `View Issues`. First, we parse that page (see Fig. 3 for a snippet) with an ANTLR parser to identify three possible user actions $\{a_1, a_2, a_3\}$. The action a_1 is the action when a user clicks on the `Create Issue` link. This action does not require any additional constraints. For each action that navigates to a new web page, we add an assertion that we have successfully navigated to it. Hence, we set $a_1.\text{asserts} = \{\alpha_1\}$, with α_1 being the assertion that we have successfully navigated to the `Create Issue` web page.

The action a_2 represents a click on the button labeled `edit` in the web page. Since that button is included in a row of a data-table, there must be at least one element in the value-list of the data table. Hence, we analyze the `value` attribute of the `dataTable` definition in the XHTML page (see Line 3 in Fig. 3) to identify where the data of that table is queried from. In the example, we identify the method `getAllIssues` from the bean named `requestBean` as a data provider (see Fig. 4a). Hence, we symbolically execute that method and identify that the result λ originates from a database query (see Lines 7ff in Fig. 4a). Hence, we set $\lambda.\text{length} \geq 1$. Moreover, we generate a new symbolic object e of type `Issue`, and add e to both λ , as well as to the required set of entities in db . After adding these constraints to \mathcal{C} , we analyze the value of the `action` attribute of the button (see Line 12 in Fig. 3). Since the action in Line 12 references a EJB bean, we symbolically execute that method as well. We pass the symbolic object e as a parameter for the argument `issue`, since it is the referenced variable of that row (see Line 3). Figure 4b shows the invoked method `edit`. It returns a string `view.xhtml`, and thus the action a_2 navigates to the `View Issue` page. We therefore add an assertion α_2 for navigating to that page to a_2 . Additionally, we add an assertion α_3 to a_2 which asserts that the current value of e is displayed in the data table on the `View Issues` page.

As a symbolic executor of Java EE programs, we have implemented a tool that is similar to the symbolic execution system proposed in [6], though it has a more powerful API to pass a given set of constraints as well as a symbolic representation of a heap into the system. In our approach, we need those features in order to share the constraints from the analysis of the web-application with the symbolic execution system. Our symbolic execution tool uses Choco [14] as a constraint solver and is also available as open source on a public repository [5].

The action a_3 represents a click on the button labeled *archive*. The action is only rendered when the value of the field `status` in the current `issue` (i.e. the instance of the current row in the data table) has a value that equals the string “OPEN”. Similar to the action a_2 , we generate a new symbolic object e' , and add e' to the data-value list λ . Similar to the approach in [6], we represent the attribute-values of an object as logic variables. Hence, we generate a new logic variable s' , add the constraint $s' = \text{“OPEN”}$, and set $e'.status = s'$. We represent primitive data types as either integer or real values. For example, a logic variable of type `char` is represented as an ASCII encoded integer value, and the value of a `String` object is represented as an array of logic variables of type `char` (exactly as it is implemented in the `java.lang.String` class). Hence, s' is internally represented with four logic variables s'_1, \dots, s'_4 having the constraints $s'_1 = 79$, $s'_2 = 80$, $s'_3 = 69$, and $s'_4 = 78$. We generate the test cases based on the solution a constraint solver has found, and we re-transform the integer values for the variables in the solution to its original type, e.g., s'_2 has the value “P”.

For each identified user action a , we generate a copied state q' from the current state q , add a to its action-sequence, and update the current web page of q' to the page ω' that the action a navigates to. Then, we start the process of identifying actions on ω' again. We stop our test case generation when either a maximum action-sequence length has been reached, or when there is no action on the current page that changes the state q . Consider the example with two web pages ω_1 and ω_2 that both link to each other, and we start our test case generation in ω_1 . Since a click on a link only changes the current web page of the state q – but adds no additional constraints to q – we only generate two actions a_1 and a_2 . In a_1 , a user clicks on the link that navigates to ω_2 , and a_2 navigates to ω_1 respectively.

4 Tool Implementation and Experimental Evaluation

We have implemented our approach in a tool that is available as open source on a public repository [5]. Figure 5 shows an overview of the key components of that tool. The figure shows our system in a state in which four execution-states q_1, \dots, q_4 have already been generated (left-hand side). Currently, we use a first-in-first-out principle to take the next state q^* of that list. First, we parse the content of the current web page $q^*.page$ of that state with an ANTLR [13] parser for XML documents. As a result, we identify a set \mathcal{A} of possible user actions on $q^*.page$. Second, we filter \mathcal{A} based on the coverage $q^*.cov$ in order to identify user actions that have not yet been executed in the state q^* . Each of these filtered

actions is then executed based on the current state q^* . As a result, we generate $n \geq 1$ new states q^*_1, \dots, q^*_n , which are then added to the list of states shown on the left-hand side of Fig. 5.

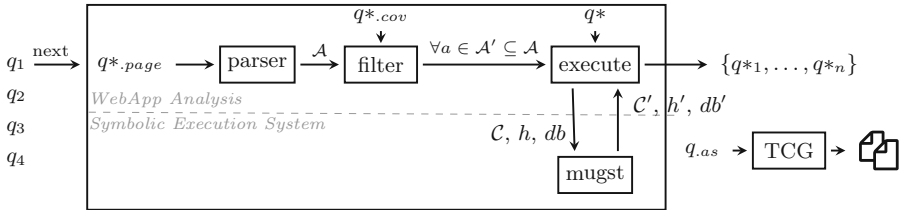


Fig. 5. An overview of the key components of the tool implementing our approach. On the top part is the web-application parser, on the bottom part the symbolic execution system for Java EE programs.

When the action a invokes a method of an EJB in the web application, we use a symbolic execution system called *mugst* [5] to symbolically execute that Java method. If that execution has more than one distinct result based on the given constraint set \mathcal{C} , symbolic heap h , and database state db , we generate for each of these results a different state as output. Every executed action is added to the action-sequence of q^* . We stop the generation of new states when either a fixed time budget is achieved (e.g. five minutes execution time) or when there are no new uncovered actions. Additionally, we discard states from the left-hand side when the length of their action-sequence is greater than a configurable maximum length. Finally, we use a test-case generator (TCG) to generate executable test cases based on a given action-sequence $q.as$.

We use the example application from Fig. 3 as an application under test in our experiment. Figure 6 shows a snippet of a generated test case with an action-sequence of length three. The test start on the page *View Issues*. First, it clicks on the first *edit* button of the data-table displayed on that page. Then, an assertion of that action is generated in Lines 7ff. It checks if we have navigated to the *Edit Issue* page by identifying elements on that page, e.g. a text field with *author* as identifier and A as input text.

In order to click on the *edit* button, an entity e of type **Issue** must exist in the database. Line 3 shows that required entity e with concrete values for its attributes. Similar to the approach proposed in [6], we represent the values of an entity as logic variables. For instance, we represent the value of the attribute **status** with a logic variable s . In order to click on the button **archive**, the value of s must be equal to the string constant *OPEN* (see constraint Line 14 in Fig. 3), and therefore we add such a constraint on s to the constraint stack \mathcal{C} . Our tool automatically parses the *xhtml* page for such constraints, generates them as explained before, and adds them to \mathcal{C} .

The second action clicks on the *submit* button on the *Edit Issue* page and navigates back to the *View Issue* page (see Line 13 in Fig. 4b). Similar, the third

```

1 public class IssueTrackerTest extends WebappTest {
2 // required entities:
3 // w.it.entity.Issue(0): id=1, status="OPEN", name="A"
4 @Test public void testIt() throws Throwable {
5     driver.findElement(By.xpath(
6         "//input[@type='submit' and @value='edit'] [1]")).click();
7     try {
8         driver.findElement(
9             By.xpath("//input[@id='author', contains(text(), 'A')]"));
10    } catch(NoSuchElementException e) { fail(); }
11
12    driver.findElement(
13        By.xpath("//input[@type='submit' and @value='submit']")).click();
14    try {
15        driver.findElement(By.xpath("//tr/td[contains(text(), '1')]"));
16        driver.findElement(
17            By.xpath("//tr/td[contains(text(), 'OPEN')]"));
18    } catch(NoSuchElementException e) { fail(); }
19
20    driver.findElement(
21        By.xpath("//input[@type='submit' and @value='archive']")).click();
22    try {
23        driver.findElement(By.xpath("//tr/td[contains(text(), '1')]"));
24    } catch(NoSuchElementException e) { fail(); }
25 }

```

Fig. 6. A snippet of a test case generated by our tool.

action clicks on the *archive* button on the *View Issue* page. In order to click on that button, the constraint $s = OPEN$ is added to the constraint stack.

Table 1. Statistics for generating test cases for the example application.

n	2	3	4	5	6	7	8	9	10
t	1	2	4	7	11	23	42	91	191
#TC	21	44	99	206	446	940	1963	4107	8416

In contrast to Java Standard Edition (SE) applications, a Java EE application typically runs inside a *container* that provides additional functionalities, such as dependency injection. In Line 3 of Fig. 4a, an instance of type **EntityManager** is injected into the field **em** by the container. However, the concrete injected instance must not be known by the owning class. For example, both Hibernate [8] and EclipseLink [4] are valid implementations of an **EntityManager**, and an instance of both implementations could be injected into the field **em**. A symbolic execution of Java EE applications must take that container-managed initialization of EJB classes into account. In our prototype, we inject instances of special classes – such as of type **EntityManager** – similar to the approach proposed in [6]. Further special handling is the subject of future work.

We use JUnit [10] as a framework to execute the tests, and Selenium [15] as a framework to automate a web browser, e.g. to simulate user-actions in a real browser such as Firefox or Chrome.

Table 1 shows the results of our experiment on the example application. The first column shows the fixed action-sequence length, the second column the time t to generate the test cases in seconds, and the third column the amount of generated distinct test cases. The data shows that we can generate large test suites in a reasonable amount of time. The reason why we generate much more test cases when n increases, is that we currently see each element in a data-table as a distinct entity that results in a different system state, and thus influences our coverage criteria.

5 Related Work

The literature proposes many approaches and tools for an automated test case generation of user-driven applications. Similar to our approach, the work of Jensen et al. [9] uses symbolic execution to systematically explore all paths of a program invoked by the user-interface. In contrast to that work, our approach combines constraints from the web-page definition (`xhtml` file), as well as the invoked program methods. Additionally, we also implemented initial support for common Java EE features (e.g. dependency injection) in our symbolic execution system. Other works [2,6,7] generate unit tests for EJB classes, though they do not take constraints into account that result from a user-driven application which invokes methods of those EJBs.

Similar to Mirshokraie [11], we generate assertions for detecting faults in the application, and we generate executable JUnit test cases with Selenium as a web-driver. Currently, our assertions are verifying that specific UI elements are displayed in the web browser when a specific user-action has been executed. MIRSHOKRAIE mutates the Domain Object Model (DOM) that represents the rendered HTML page in order to expose weaknesses in the generated test suite. Currently, such an evaluation of the generated test cases is not implemented by our tool, but may be the subject of future work.

6 Conclusion, Limitations and Future Work

We have presented an approach for an automated test case generation for web applications based on the Java EE platform. Our approach has been implemented in a prototype that is available as open-source on a public repository.

The symbolic execution system that we have used in our prototype is in early-stage, yet under ongoing development. For the generation of Java entity objects, we are currently able to generate a set of objects for one type, but not be able to query a database with complex SQL queries (such as those having subqueries, etc.). The main contribution of this paper is presenting an approach to generate test cases for web applications, as well as a tool that implements that approach. As a coverage criteria, we analyze if an action changes the system state that has not yet been covered. We symbolically represent such a state as a set of constraints, as well as a symbolic heap space including generated objects. As a

hard termination criteria, we stop the test case generation when a fixed (user-defined) action-sequence length has been reached. However, our implementation has a well defined API that allows to add additional coverage strategies.

As future work, we would like to improve our symbolic execution system and run more experiments with the prototype on large real-world web applications.

References

1. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P., Bertolino, A., et al.: An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.* **86**(8), 1978–2001 (2013)
2. Arcuri, A., Fraser, G.: Java enterprise edition support in search-based JUnit test generation. In: Sarro, F., Deb, K. (eds.) *SSBSE 2016*. LNCS, vol. 9962, pp. 3–17. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47106-8_1
3. Diakopoulos, N., Cass, S.: *Interactive: the top programming languages 2016*. IEEE Spectr. (2016). <http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016>
4. EclipseLink: Comprehensive open-source Java persistence solution addressing relational, XML, and database web services. The Eclipse Foundation (2018). <http://www.eclipse.org/eclipselink/>
5. Fuchs, A.: WWU WebApp GitHub Repository. WWU Münster (2018). <https://github.com/wwu-pi/webapp>
6. Fuchs, A., Kuchen, H.: Unit testing of database-driven Java enterprise edition applications. In: Gabmeyer, S., Johnsen, E.B. (eds.) *TAP 2017*. LNCS, vol. 10375, pp. 59–76. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61467-0_4
7. Fuchs, A., Kuchen, H.: Test-case generation for web-service clients. In: *Proceedings of the Symposium on Applied Computing*. ACM (2018, accepted)
8. Hibernate: Your relational data. Objectively. Redhat (2018). <http://hibernate.org/or/>
9. Jensen, C.S., Prasad, M.R., Møller, A.: Automated testing with targeted event sequence generation. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 67–77. ACM (2013)
10. JUnit: JUnit 5. JUnit Team (2018). <https://junit.org/junit5/>
11. Mirshokraie, S.: Effective test generation and adequacy assessment for JavaScript-based web applications. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 453–456. ACM (2014)
12. Myers, G.J., Sandler, C., Badgett, T.: *The Art of Software Testing*. Wiley, Hoboken (2011)
13. Parr, T.: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh (2013)
14. Prud’homme, C., Fages, J.G., Lorca, X.: *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2016). <http://www.choco-solver.org>
15. Selenium: Selenium - Web Browser Automation. Selenium (2018). <https://www.seleniumhq.org/>



Ghosts for Lists: From Axiomatic to Executable Specifications

Frédéric Loulergue¹, Allan Blanchard², and Nikolai Kosmatov³

¹ School of Informatics Computing and Cyber Systems,
Northern Arizona University, Flagstaff, USA
`frederic.loulergue@nau.edu`

² Inria Lille — Nord Europe, Villeneuve d'Ascq, France
`allan.blanchard@inria.fr`

³ Software Reliability and Security Lab, CEA, LIST, PC 174,
Gif-sur-Yvette, France
`nikolai.kosmatov@cea.fr`

Abstract. Internet of Things (IoT) applications are becoming increasingly critical and require formal verification. Our recent work presented formal verification of the linked list module of Contiki, an OS for IoT. It relies on a parallel view of a linked list via a companion ghost array and uses an inductive predicate to link both views. In this work, a few interactively proved lemmas allow for the automatic verification of the list functions specifications, expressed in the ACSL specification language and proved with the FRAMA-C/WP tool.

In a broader verification context, especially as long as the whole system is not yet formally verified, it would be very useful to use runtime verification, in particular, to test client modules that use the list module. It is not possible with the current specifications, which include an inductive predicate and axiomatically defined functions. In this early-idea paper we show how to define a provably equivalent non-inductive predicate and a provably equivalent non-axiomatic function that belong to the executable subset E-ACSL of ACSL and can be transformed into executable C code. Finally, we propose an extension of FRAMA-C to handle both axiomatic specifications for deductive verification and executable specifications for runtime verification.

Keywords: Linked lists · Executable specification
Deductive verification · Runtime verification · FRAMA-C
Internet of Things

1 Introduction

Among distributed systems, connected devices and services, also referred to as the Internet of Things (IoT), have proliferated very quickly in the past years. There are now billions of interconnected devices, and this number is growing. It is anticipated that by 2021, about 46 billion devices will be in use.

Some of these devices are in service in security-critical domains, but even in domains that are not necessarily critical, privacy issues may arise with devices collecting and transmitting a lot of personal information. Moreover, insufficiently secured devices can be used, for example, for massive distributed denial of service attacks [8]. This raises important security challenges. Formal methods – that have been successfully used for years in highly critical domains – can help today to bring security into the IoT field.

While the correctness of an implementation with respect to a formal functional specification provides a very strong form of guarantee, it can be very costly to achieve, and is currently mostly reserved to domains where it is required by regulations or offers a competitive advantage. In practice, it is very useful to rely on a combination of formal methods to achieve an appropriate degree of guarantee: static analysis to ensure the absence of runtime errors, deductive verification to prove functional correctness, and runtime verification for parts of code that cannot be (or are not yet) proved using deductive verification.

This work uses FRAMA-C [7], a framework for source code analysis of industrial-size programs written in C. FRAMA-C offers combined formal methods approaches, by providing its users with a collection of plugins that perform static and dynamic analysis for safety and security critical software. Collaborative verification across plugins is enabled by their integration on top of a shared kernel, and their compliance to a common specification language: ACSL [1].

Recently, FRAMA-C has been applied in the context of the IoT for verification of several modules of Contiki [5], an open-source operating system for the IoT. In our previous work, we formally verified the linked list module of Contiki [2]. The verification technique relies on companion ghost arrays to provide an alternative view of the lists and a *linking predicate* relating a list and its companion array. A small set of lemmas (proved using COQ [12]) allow us to verify the specifications of the list module functions automatically using the FRAMA-C/WP tool.

In a broader verification context, especially when some parts of the system are not yet proven, it would be desirable to benefit of the formal specification of the proven module while testing its (yet unproven) client modules, e.g. to check that the preconditions of proven functions are always satisfied along the tests of a client module. A FRAMA-C plugin, called E-ACSL2C in this paper, can automatically transform specifications into executable C code, verifiable at runtime, but only if they belong to the executable subset of ACSL, named E-ACSL [4, 11]. In our work [2], the formal specification of the list module relies on an inductive linking predicate and an axiomatically defined function, which are convenient for deductive verification but do not belong to this subset. Of course, we do not want to loose the effort put in conducting deductive verification.

This early-idea paper explores a solution inspired by verification by program transformation: instead of generating an executable definition from an axiomatic one, we propose to define an executable one and prove its equivalence with the axiomatic definition. To support this methodology, the already implemented E-ACSL2C tool could be extended as follows: E-ACSL2C would look for known equivalences when encountering a non-executable element in order to produce

its executable counterpart. This extension is simple enough to be quickly and safely added to E-ACSL2C, but of course requires the user to state the executable definitions and prove their equivalence with the axiomatic ones. We present a proof-of-concept application of this approach to the list case study [2].

The remaining part of the paper is organized as follows. In Sect. 2 we give an overview of FRAMA-C, and its WP and E-ACSL2C plugins. Section 3 briefly presents the verification of the Contiki list module, with a particular emphasis on two axiomatic definitions that cannot be handled by E-ACSL2C. We then present equivalent executable specifications and discuss the proofs of equivalence with the axiomatic specifications (Sect. 4). In Sect. 5 we discuss an extension of E-ACSL2C to support such an approach and related work.

2 FRAMA-C Platform and Its WP and E-ACSL2C Plugins

FRAMA-C [7] offers various plugins built around a kernel providing basic services required for any analysis. It relies on the CIL frontend [9] extended to treat ACSL annotations. ACSL, for ANSI/ISO C Specification Language, is based on the notion of contract like in Eiffel or JML. It allows users to specify functional properties of programs through pre/post-condition, and provides different ways to define predicates and logic functions. Some useful built-in predicates and logic functions are provided, to handle for example pointer validity or separation.

WP is a deductive verification plugin provided with FRAMA-C. It is based on weakest precondition calculus. Given a C program annotated in ACSL, WP generates the corresponding proof obligations that can be discharged by SMT solvers or with interactive proof. A combination of automatic and interactive proofs often offers a good trade-off for a complete proof. Indeed, some properties can only be defined recursively, and in this case, SMT solvers often become inefficient, trying to unroll them. By using inductive or axiomatically defined functions, we can prevent this behavior but reasoning about them still requires induction, a task that SMT solvers are not good at. Thus, the last step is generally to state lemmas that can be directly instantiated by SMT solvers. These lemmas can be easily used by SMT solvers to verify specifications, but their proofs require to reason by induction: they are proved interactively.

The E-ACSL2C plugin transforms annotations that belong to the executable subset E-ACSL of ACSL into C code in order to verify them at runtime [4]. This subset [4, 11] restricts ACSL to executable features: quantifications over finite intervals only, no axioms or lemmas, no inductive predicates or axiomatic definitions of logic functions. Mathematical (unbounded) integer arithmetic is supported via a translation to larger types or using a dedicated library (GMP). Pointer properties (such as validity) are handled thanks to a dedicated memory model [14].

3 The List Module of Contiki

The linked list module of Contiki is a critical module of the kernel intensively used in the core part of the OS. Its formal verification was thus necessary.

The formal verification we have performed [2] relies on a view of each list via a ghost array that mirrors it, and the following *linking predicate* defining their relation.

```

1 inductive linked_n{L}(struct list *root, struct list **cArr,
2   ℤ index, ℤ n, struct list *bound){
3 case linked_n_bound{L}:
4 ∀ struct list **cArr, *bound, ℤ index;
5   0 ≤ index ≤ MAX_SIZE ⇒
6     linked_n(bound, cArr, index, 0, bound);
7 case linked_n_cons{L}:
8 ∀ struct list *root, **cArr, *bound, ℤ index, n;
9   0 < n /\ 0 ≤ index /\ 0 ≤ index + n ≤ MAX_SIZE /\
10  \valid(root) /\ root == cArr[index] /\
11  linked_n(root->next, cArr, index + 1, n - 1, bound) ⇒
12  linked_n(root, cArr, index, n, bound);
13 }

```

This predicate inductively relates a list starting at `root` to a segment of companion array `cArr`, starting from an offset `index` and having `n` elements, that ends with the excluded cell address `bound` (either `NULL` or a pointer to the first non-represented list element if any). This relation is verified (cf. axiom `linked_n_cons`, lines 6–11) if `root` is a valid memory location, if we find this value at offset `index` of `cArr`, and if, recursively, the list that starts at `root->next` is linked to the segment starting from `index+1` with `n-1` elements. That is, for all i , the address of the i^{th} cell of the list can be found at `index+i` of `cArr`. The empty list (cf. axiom `linked_n_bound` lines 3–5), that starts and ends with `bound`, is related to a `cArr` segment from any `index` for a length of 0 elements. The linking relation between the list and its ghost array is maintained as an invariant by the functions of the list API. Thus, for verification we add some ghost code that updates the companion array when needed.

Some lemmas allow us to split (or merge) a list into sub-lists related to consecutive subranges of the companion array. It allows, for example, to prove properties about the removal of an element of the list, where we have to show that the beginning of the list did not change, and that all elements starting from the item to remove have been shifted, so the list does not contain the item anymore. Of course, that means that we need a way to specify the location of an element in the list. This is done using the `index_of` function presented below:

```

1 axiomatic Index_of_item {
2   logic ℤ index_of(struct list *item, struct list **cArr,
3     ℤ down, ℤ up) reads cArr[down .. up-1];
4   axiom no_more_elements:
5   ∀ struct list *item, **cArr, ℤ d, u; 0 ≤ u ≤ d ⇒
6     index_of(item, cArr, d, u) == u;
7   axiom found_item:
8   ∀ struct list *item, **cArr, ℤ d, u;
9     0 ≤ d < u /\ cArr[d]==item ⇒ index_of(item, cArr, d, u)==d;
10  axiom not_the_item:
11  ∀ struct list *item, **cArr, ℤ d, u;
12    0 ≤ d < u /\ cArr[d]≠item ⇒
13    index_of(item, cArr, d, u) == index_of(item, cArr, d+1, u);
14 }

```

This function searches an `item` in the companion array `cArr` (therefore, in the list), between two indices `down` and (excluded) `up` and returns the corresponding offset. If the element is not in the list, the function returns `up`. This definition is also recursive. For an empty range, `down` equals `up`, the offset is `up` (cf. axiom `no_more_elements`, lines 5–6). For a non-empty range, if the element is the first one (cf. axiom `found_item`, lines 7–9), the offset is the index of this element `down`. Finally, for a non-empty range, if the first element, at offset `d`, is not the one we are searching (cf. axiom `not_the_item`, lines 10–12), the function has to search the item in the subrange that starts at `d+1`.

Finally, additional properties (cf. [2]) are required to specify memory separation between the different elements of the list and between the elements of the list and the ghost array. For lack of space, we do not present them here.

4 From Axiomatic to Executable Specifications

We design new specifications in the E-ACSL subset of ACSL, and prove their equivalence with the axiomatic specifications presented in the previous section.

An executable linking predicate `linked_exec` equivalent to `linked_n` follows:

```

1 logic boolean array_view(struct list *root,
2                           struct list **cArr,
3                           ℤ idx, ℤ size, struct list *bound) =
4   (size==0)? root==bound: (root==cArr[idx] ∧
5     array_view(root->next, cArr, idx+1, size-1, bound));
6
7 predicate linked_exec{L}(struct list *root,
8                           struct list **cArr, ℤ idx,
9                           ℤ size, struct list *bound) =
10  0 ≤ size ∧ 0 ≤ idx ∧ idx + size ≤ MAX_SIZE ∧
11  (∀ ℤ k; idx ≤ k < idx + size ⇒ \valid(cArr[k])) ∧
12  array_view(root, cArr, idx, size, bound) == \true;

```

The idea is to replace an inductive predicate, which is not supported by E-ACSL, by a non-inductive predicate and a recursive logical function. Informally, the validity stated in `linked_exec` as a bounded quantification (line 9) and the equality between `root` and `cArr[idx]` imply the validity of `root` (as stated line 9 of `linked_n`) and the equality in `linked_n_cons`. The other conditions, in `linked_exec` and `linked_n` respectively, are identical.

An executable function almost equivalent to the axiomatic `index_of` follows:

```

1 logic ℤ index_of_exec(struct list *item, struct list **cArr,
2                       ℤ down, ℤ up) =
3   (down < 0 ∨ up < 0) ? -1: (0 ≤ up ∧ up ≤ down) ? up:
4   (0 ≤ down ∧ down < up ∧ cArr[down] == item) ? down:
5   index_of_exec(item, cArr, down+1, up);

```

This function is not fully equivalent to the inductive axiomatic function `index_of` previously presented because the axiomatic definition says nothing when one of the bound is negative. An executable version could lead to runtime errors in that case, thus it includes an additional check to prevent them.

Therefore we add a new case to the axiomatic version `index_of` to ensure the equivalence with the new logical function and its (stricter) bound checks:

```

1 axiom invalid_bounds:
2   ∀ struct list *item, **cArr, ℤ down, up;
3   (down < 0 ∨ up < 0) ⇒ index_of(item, cArr, down, up) == -1;

```

The deductive verification of the list module is not impacted by this modification.

We have proved the following two lemmas that state respectively that the axioms defining the function `index_of` and the recursive function `index_of_exec` are equivalent, and that the inductive predicate `linked_n` and the non inductive predicate `linked_exec` are equivalent:

```

1 lemma equiv_index_of:
2   ∀ struct list *item, struct list **cArr, ℤ down, ℤ up;
3   index_of(item, cArr, down, up) ==
4   index_of_exec(item, cArr, down, up);
5 lemma equiv_linked:
6   ∀ struct list *root, struct list **a, struct list *b,
7   ℤ index, ℤ size;
8   linked_n(root, a, index, size, b) ⇔
9   linked_exec(root, a, index, size, b);

```

The first lemma is proved using COQ as follows: after case reasoning to assure that $0 \leq \text{down} < \text{up}$, the result is established by induction on a value `len` equal to `up - down` and replacing `down` with `up - len`.

The second lemma is also proved in COQ. The first implication is proved by induction on the inductive predicate `linked_n`. The second implication is proved by induction on `size` and using two lemmas on `array_view` themselves proved by induction on `size`. The proofs are not very simple but comparable to some of the lemmas for `linked_n` proved for the deductive verification of the list API.

5 Discussion

The E-ACSL2C plugin currently does not support the full E-ACSL subset, but it is evolving rapidly. The proposed approach assumes a better support of the E-ACSL language to make the specifications described in the previous section executable.

Since this support is still partial, in order to apply the proposed approach, we proceed by manual transformation: as predicate definitions (such as `linked_exec`) are not supported yet, we inline them, i.e. instead of applying `linked_exec` in a specification, we copy its body into it. As logical function definitions are not supported yet, we define corresponding C functions (and specify their equivalence with their logical counterparts, these specifications being automatically checked by WP) and hard-coded calls to these functions in the body of the C list functions we specified, using C `assert`. Our study provides a proof-of-concept for the proposed approach of creating a (provably equivalent) executable specification.

Note that even if the `linked_exec` predicate is proved to be equivalent to the previous `linked_n` predicate, the inductive version is still needed for the formal

proof to ensure that we get an induction principle on the Coq side, but also to prevent the SMT solvers or WP to unfold the predicates during automatic proof.

Assuming the E-ACSL specification language is fully supported, we envision an extension of E-ACSL2C to support this approach in a convenient and semi-automatic way. We propose a new annotation `equivalent` taking two names as arguments (either two predicates or two functions), that would:

- generate the corresponding equivalence lemma (to be proved),
- make E-ACSL2C replace the first argument with the second argument for the generation of C code.

Note that this annotation could be transitive: if the second argument is still not executable, the system could look for another equivalence relating the second name to a third one, and so on.

The proposed approach enables combined verification, very helpful in practice as many real-life systems are not fully proved. Its benefit is the possibility to use both axiomatic specification, more convenient for deductive verification, and equivalent executable specification, usable for dynamic verification, without the need for a large and complex extension of E-ACSL2C. For more automation – that would require much more implementation work – it could be possible to build on the work of Tollitte et al. [13].

For higher-level languages, such as Eiffel or Java, a related approach is to associate a model to a class [10]. The model plays the same role as the companion array in our approach. Such a model is also a class, but an immutable one used only for verification purposes. Model classes are valid classes of the considered language, and can therefore be used in dynamic verification tasks. For deductive verification purposes, the classes may be translated to elements of theories of the underlying theorem provers. In this case, the faithfulness of the mapping may be checked [3].

As future work we also plan to experiment with alternative specifications in the spirit of the work of Gladisch and Tyszberowicz [6]. In the case of JML, they used a pure observer method that takes a list object and an index, and returns the object at that index in the list, to specify Java methods on a linked list data structure. While the methods they consider are simpler than the list API of Contiki, our ghost arrays can essentially be seen as observations of the linked lists. We could consider such an observer directly written as a logical function in E-ACSL. C pointers are however not Java references and can lead to some complications.

Acknowledgment. This work was partially supported by a grant from CPER DATA and the project VESSEDIA, which has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 731453. The authors thank the FRAMA-C team for providing the tools and support. Many thanks to the anonymous referees for their helpful comments.

References

1. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. <http://frama-c.com/acsl.html>
2. Blanchard, A., Kosmatov, N., Loulergue, F.: Ghosts for lists: a critical module of Contiki verified in Frama-C. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 37–53. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_3
3. Darvas, Á., Müller, P.: Proving consistency and completeness of model classes using theory interpretation. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 218–232. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12029-9_16
4. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Proceedings of the ACM Symposium on Applied Computing, SAC 2013, pp. 1230–1235. ACM (2013)
5. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the 29th Annual IEEE Conference on Local Computer Networks, LCN 2004, pp. 455–462. IEEE Computer Society (2004)
6. Gladisch, C., Tyszbrowicz, S.S.: Specifying linked data structures in JML for combining formal verification and testing. *Sci. Comput. Program.* **107–108**, 19–40 (2015)
7. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Jakobowski, B.: Frama-C: a software analysis perspective. *Form. Asp. Comput.* **27**(3), 573–609 (2015)
8. Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.M.: DDoS in the IoT: Mirai and other botnets. *IEEE Comput.* **50**(7), 80–84 (2017)
9. Nacula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_16
10. Polikarpova, N., Furia, C.A., Meyer, B.: Specifying reusable components. In: Leavens, G.T., O’Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 127–141. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_9
11. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language. <http://frama-c.com/download/e-acsl/e-acsl.pdf>
12. The Coq Development Team: The Coq proof assistant. <http://coq.inria.fr>
13. Tollitte, P.-N., Delahaye, D., Dubois, C.: Producing certified functional code from inductive specifications. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 76–91. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35308-6_9
14. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: Proceedings of the International Symposium on Memory Management, ISMM 2017, pp. 47–58. ACM (2017)

Author Index

- Beckert, Bernhard 83
Bernard, Julien 27
Beyer, Dirk 3
Blanchard, Allan 177
Blatter, Lionel 44
Brucker, Achim D. 159
- Correnson, Loïc 120
- Dangl, Matthias 3
- Fuchs, Andreas 167
- Héam, Pierre-Cyrille 27
Herda, Mihai 83
Herzberg, Michael 159
- Julliand, J. 63
- Keller, Chantal 103
Kosmatov, Nikolai 44, 139, 177
Kouchnarenko, Olga 27, 63
- Le Gall, Pascale 44
Le, Viet Hoang 120
Lemberger, Thomas 3
Loulergue, Frédéric 177
- Masson, P.-A. 63
- Petiot, Guillaume 44
Prevosto, Virgile 44
- Signoles, Julien 120, 139
- Tautschnig, Michael 3
Tyszberowicz, Shmuel 83
- Voiron, G. 63
Vorobyov, Kostyantyn 139
- Wiels, Virginie 120