Achim D. Brucker
Jacques Julliand (Eds.)

LNCS 7305

Tests and Proofs

6th International Conference, TAP 2012
Prague, Czech Republic, May/June 2012
Proceedings

Lecture Notes in Computer Science 7305

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

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
TU Dortmund University, Germany

Madhu Sudan
Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Achim D. Brucker Jacques Julliand (Eds.)

Tests and Proofs

6th International Conference, TAP 2012
Prague, Czech Republic, May 31 – June 1, 2012
Proceedings

Springer

Volume Editors

Achim D. Brucker
SAP Research
Vincenz-Priessnitz-Straße 1
76131 Karlsruhe, Germany
E-mail: achim.brucker@sap.com

Jacques Julliand
LIFC UFR ST
16 route de Gray
25030 Besançon Cedex, France
E-mail: jacques.julliand@univ-fcomte.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-30472-9 e-ISBN 978-3-642-30473-6
DOI 10.1007/978-3-642-30473-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012937676

CR Subject Classification (1998): D.2.4, D.2, D.1, D.3, F.3, F.4.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2012

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, 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.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 6th International Conference on Tests and Proofs (TAP 2012) held from May 31 to June 1, 2012, in Prague, Czech Republic, as part of the TOOLS Federated Conferences.

TAP 2012 was the sixth event in a series of conferences devoted to the convergence of tests and proofs for developing novel techniques and applications that support engineers in building secure, safe, and reliable systems. While for several decades the proof and test communities were quite distant from each other, there is a recent trend—both in academia and industry—to combine both approaches. This cross-fertilization results, on the one hand, in new insights into the fundamentals of tests and proofs and, on the other hand, to the development of novel techniques that increase the quality of large-scale systems.

The first TAP conference (held at ETH Zurich in February 2007) was an effort to provide a forum for the cross-fertilization of ideas and approaches from the testing and proving communities. For the 2008 edition we found the Monash University Prato Centre near Florence to be an ideal place providing a stimulating environment. The third TAP was again held at ETH Zurich in July 2009. Since 2010, TAP has been co-located with TOOLS, and its instance for 2010 therefore took place at the School of Informatics (E.T.S. de Ingenieria Informatica) of the University of Malaga, while TOOLS 2011 took place once more at ETH Zurich. In 2012, TAP was part of TOOLS again, this time held at the Czech Technical University in Prague.

We wish to sincerely thank all authors who submitted their work for consideration. We received 29 submissions from which we finally accepted 13, after a formal refereeing process requiring at least three reviews from the Program Committee or by a reviewer appointed by the Program Committee. The various selected papers are contributions essentially in the following four themes of research: model-based testing, scenario based-testing, complex data structure generation, and the validation of protocols and libraries. Moreover, we were grateful to host a tutorial on the static analysis platform Frama-C and the concolic test generator Pathcrawler as well as their combination.

We would like to thank the Program Committee members as well as the additional reviewers for their energy and their professional work in the review and selection process. Their names are listed on the following pages. The lively discussions during the paper selection were vital and constructive. We are also very proud that TAP 2012 featured two keynotes by Andreas Kuehlman (Coverity, Inc.) and Corina Pasareanu (NASA). Both are well-accepted experts in the fundamentals and applications of testing and proving techniques. Our thanks go to both of them.

It was a team effort that made the conference so successful. We are grateful to the TAP Conference Chairs Yuri Gurevich and Betrand Meyer for their support. Moreover, we particularly thank the organizers of the Tools Federated Conferences, Pavel Tvrdik, Michal Valenta, Jindra Vojikova, and Jan Chrastina, from Czech Technical University in Prague, for their hard work and their support in making the conference a success.

March 2012

Achim D. Brucker
Jacques Julliand

Organization

Conference Chairs

Yuri Gurevich	Microsoft Research, USA
Betrand Meyer	ETH Zurich, Switzerland

Program Chairs

Achim D. Brucker	SAP Research, Germany
Jacques Julliand	Université de Franche-Comté, France

Program Committee

Nazareno Aguirre	King's College, London, UK
Bernhard K. Aichernig	TU Graz, Austria
Paul Ammann	George Mason University, USA
Dirk Beyer	University of Passau, Germany
Nikolaj Bjorner	Microsoft Research, USA
Achim D. Brucker	SAP Research, Germany
Robert Clarisó	University of Catalonia, Spain
Marco Comini	Università di Udine, Italy
Catherine Dubois	ENSIIE-CEDRIC, France
Gordon Fraser	Saarland University, Germany
Angelo Gargantini	Università di Bergamo, Italy
Alain Giorgetti	Université de Franche-Comté, France
Patrice Godefroid	Microsoft Research, USA
Martin Gogolla	University of Bremen, Germany
Arnaud Gotlieb	INRIA, France
Reiner Hähnle	Technische Universität Darmstadt, Germany
Bart Jacobs	Université de Louvain, Belgium
Jacques Julliand	Université de Franche-Comté, France
Thierry Jéron	INRIA, France
Gregory Kapfhammer	Allegheny College, USA
Nikolai Kosmatov	CEA Saclay, France
Victor Kuliamin	Russain Academy of Sciences, Russia
Karl Meinke	University of Stockholm, Sweden
Jeff Offutt	George Mason University, USA
Holger Schlingloff	Fraunhofer FIRST and Humboldt University, Germany

T.H. Tse	University of Hong Kong, SAR China
Margus Veanes	Microsoft Research, USA
Luca Viganò	University of Verona, Italy
Burkhart Wolff	Université Paris-Sud, France
Fatiha Zaidi	Université Paris-Sud, France

Additional Reviewers

Shaukat Ali
Andra Baruzzo
Razieh Behjati
Chiara Braghin
Jens Brüning
Ming Chai
Ramona Enache
Martin Hentschel
Karthick Jayaraman
Elisabeth Joebstl
Mirco Kuhlmann
Ivan Lanese
Stefan Loewe
Olivier Ponsini
Daniel Riera
Mathias Soeken
Philipp Wendler

Table of Contents

Invited Talks

Research Papers

Short Papers

Tutorials

The Technology and Psychology of Testing Your Code as You Develop It

Andreas Kuehlmann

Senior Vice President of R&D
Coverity, San Francisco, CA
akuehlmann@coverity.com

Abstract. Much of the contemporary research in the area of software testing and verification has solely focused on advances in technology and has to a large degree ignored the fact that success in software development has as much to do with technology, as it has with psychology. For development tools to be successful in practice, they must not "get in the way of developers," play to their unique psyche, and demonstrate a measurable return of the investment, i.e., time and effort spent. Similarly, their application must fit smoothly into the existing workflow and avoid "off-cycle" processes. In this talk, we will discuss a number of technological and psychological challenges of software testing during development and argue that supporting tools must align advanced technologies with sociological and organizational aspects in order to be successful. The talk will be based on our experience in the development and deployment of static analysis technology and utilize various practical examples to demonstrate the discussed concepts.

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, p. 1, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Combining Model Checking and Symbolic Execution for Software Testing

Corina S. Păsăreanu

Carnegie Mellon Silicon Valley, NASA Ames,
M/S 269-2, Moffett Field CA 94035
corina.s.pasareanu@nasa.gov

Abstract. Techniques for checking complex software range from model checking and static analysis to testing. Over the years, we have developed a tool, Symbolic PathFinder (SPF), that aims to leverage the power of systematic analysis techniques, such as model checking and symbolic execution, for thorough testing of complex software. Symbolic PathFinder analyzes Java programs by systematically exploring a symbolic representation of the programs' behaviors and it generates test cases that are guaranteed to cover the explored paths. The tool also analyzes different thread inter-leavings and it checks properties of the code during test generation. Furthermore, SPF uses off-the-shelf decision procedures to solve mixed integer-real constraints and uses "lazy initialization" to handle complex input data structures. Recently, SPF has been extended with "mixed concrete-symbolic" constraint solving capabilities, to handle external library calls and to address decision procedures' incompleteness. The tool is part of the Java PathFinder open-source tool-set and has been applied in many projects at NASA, in industry and in academia. We review the tool and its applications and we discuss how it compares with related, "dynamic" symbolic execution approaches.

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, p. 2, 2012.
© Springer-Verlag Berlin Heidelberg 2012

From Model-Checking to Automated Testing of Security Protocols: Bridging the Gap*

Alessandro Armando[1,2], Giancarlo Pellegrino[3,4], Roberto Carbone[2], Alessio Merlo[1,5], and Davide Balzarotti[3]

[1] DIST, Università degli Studi di Genova, Italy
{armando,alessio.merlo}@dist.unige.it
[2] Security & Trust Unit, FBK-irst, Trento, Italy
{armando,carbone}@fbk.eu
[3] Institute Eurecom, Sophia Antipolis, France
{giancarlo.pellegrino,davide.balzarotti}@eurecom.fr
[4] SAP Research, Mougins, France
giancarlo.pellegrino@sap.com
[5] Università Telematica E-Campus, Italy
alessio.merlo@uniecampus.it

Abstract. Model checkers have been remarkably successful in finding flaws in security protocols. In this paper we present an approach to binding specifications of security protocols to actual implementations and show how it can be effectively used to automatically test implementations against putative attack traces found by the model checker. By using our approach we have been able to automatically detect and reproduce an attack witnessing an authentication flaw in the SAML-based Single Sign-On for Google Apps.

1 Introduction

Security protocols are communication protocols that aim at providing security guarantees (such as authentication or confidentiality) through the application of cryptographic primitives. Security protocols lie at the core of security-critical applications, such as Web-based Single Sign-On solutions and on-line payment systems. Unfortunately, security protocols are notoriously error-prone as witnessed by the many protocols that have been found vulnerable to serious attacks years after their publication and implementation. (See [13] for a survey.)

Interestingly, many attacks on security protocols can be carried out without breaking cryptography. These attacks exploit weaknesses in the protocols that are due to the complex and unexpected interleaving of different protocol sessions as well as to the possible interference of malicious agents. Since these weaknesses are very difficult to spot by traditional verification techniques

* This work has partially been supported by the FP7-ICT Project SPaCIoS (no. 257876) and by the project SIAM funded in the context of the FP7 EU "Team 2009 - Incoming" COFUND action.

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 3–18, 2012.
© Springer-Verlag Berlin Heidelberg 2012

(e.g., manual inspection and testing), a variety of novel model checking techniques specifically tailored to the analysis of security protocols have been put forward [1,18,22]. This has spurred the development of a new generation of model checkers which has proved remarkably successful in discovering (previously unknown) flaws in security protocols [4,20,28]. While in the past model checkers have been mainly used to support the analysis of security protocols at design time, recently their usage has been extended to support the discovery of vulnerabilities in actual, even deployed, systems. For instance, model checking was key to the discovery of serious vulnerabilities in the SAML-based Single Sign-On for Google Apps [3] as well as in the PKCS#11 security tokens [11]. The main limitation of the existing approaches is that reproducing attack traces found by a model checker against protocol implementations not only requires a thorough understanding of both the protocol and its implementation, but also a substantial amount of manual activity.

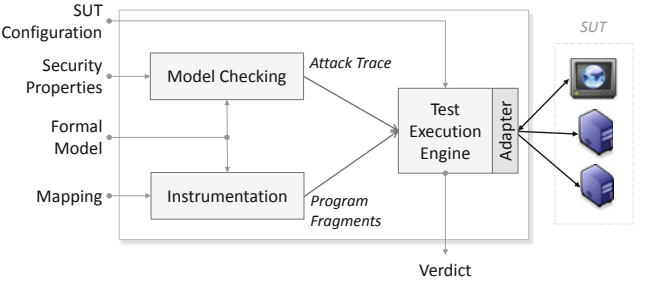


Fig. 1. Overview of the Approach

In this paper we tackle this difficulty by presenting an approach that supports *(i)* the binding of specifications of security protocols to actual implementations through model instrumentation, and *(ii)* the automatic testing of real implementations against putative attacks found by a model checker.

It is worth pointing out that most model checking techniques (and the associated tools) for security protocol analysis work on abstract models of the protocols. These models do not specify how protocol messages should be checked and generated, nor the way in which the internal state of the principals should be updated. As a consequence, the attack traces returned by these tools are not directly executable. Our paper shows that this gap can be filled in automatically. To the best of our knowledge a solution to this problem is not available.

Our approach consists of the following steps (cf. Figure 1):

Model Checking. Given a formal model of the protocol and a description of the expected security properties, a model checker systematically explores the state space of the model looking for counterexamples. Any counterexample found by the model checker is returned as an *Attack Trace*.

Instrumentation. The instrumentation step automatically calculates and provides the *Test Execution Engine* with a collection of *Program Fragments*, encoding how to verify (generate) incoming (outgoing, resp.) messages, by using the functionalities provided by the *Adapter*. The association between abstract messages and concrete ones is in the *Mapping* input.

Execution. The *Test Execution Engine* (TEE) interprets the *Attack Trace* and executes the program fragments accordingly. The *SUT Configuration* specifies which principals are part of the System Under Test (SUT) and which, instead, are simulated by the TEE. The *Verdict* indicates whether the TEE succeeded or not in reproducing the attack. Note that if the verdict is negative, the whole approach can be iterated by requesting the model checker to provide another attack trace (if any).

Our approach naturally supports both model and property-driven security testing and in doing so it paves the way to a range of security testing techniques that go beyond those implemented in state-of-the-art penetration testing tools [9,15]. For instance, prior research has shown that a number of subtle flaws found by model checkers can be exploited in real implementations as launching pad for severe attacks [3,4,11]. Moreover, even when security protocols do not suffer from design flaw, their implementations can still expose vulnerabilities. In these cases mutants can be derived from the original model [12,14] and our approach can be used to check their existence into the implementation.

In order to assess the effectiveness of the proposed approach we developed a prototype of the architecture in Figure 1 and used it to test two Web-based Single Sign-On (SSO) solutions that are available on-line, namely the SAML-based SSO for Google Apps and the SimpleSAMLphp SSO service offered by Foodle. The prototype is able to successfully execute an attack on the Google service whereby a client gets access to her own Gmail account without having previously requested it [4]. Quite interestingly, our prototype also shows that the same attack does not succeed against the SSO service of Foodle, due to specific implementation mechanisms used by SimpleSAMLphp.

2 SAML Web-Browser SSO

Browser-based Single Sign-On (SSO) is replacing conventional solutions based on multiple, domain-specific credentials by offering an improved user experience: clients perform a single log in operation to an identity provider, and are yet able to access resources offered by a variety of service providers. Moreover, by replacing multiple credentials (one per service provider) with a single one (associated with the identity provider), SSO solutions are expected to improve the overall security as users tend to use weak passwords and/or to reuse the same password on different service providers.

The OASIS *Security Assertion Markup Language* (SAML) 2.0 Web Browser SSO Profile (SAML SSO, for short) [23] is an emerging standard for Web-based SSO. Three basic roles take part in the protocol: a client C, an identity provider IdP and a service provider SP. The objective of C, typically a web browser guided by a user, is to get access to a service or a resource provided by SP. IdP is responsible to authenticate C and to issue the corresponding authentication assertions (a special type of assertion used to authenticate users). The SSO protocol terminates when SP consumes the assertions generated by IdP to grant or deny C access to the requested resource.

Figure 2 shows an excerpt of the messages exchanged during a typical execution of the SAML SSO protocol. In the first message (S1), C asks SP to provide the resource located at URI. SP then initiates the protocol by sending C a redirect response (A1) of the form:

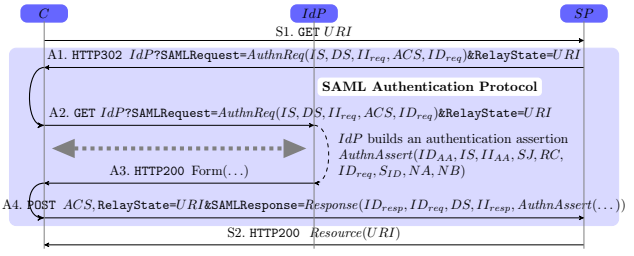


Fig. 2. SAML Web-browser SSO SP-Initiated

```
HTTP/1.1 302 Obj Moved\r\n
Location: IdP?SAMLRequest=AuthnReq(IS, DS, IIreq, ACS, IDreq)&RelayState=URI
```

where $AuthnReq(IS, DS, II_{req}, ACS, ID_{req})$ abbreviates the XML expression:

```
<AuthnRequest ID="IDreq" Version="2.0" IssueInstant="IIreq"
  Destination="DS" AssertionConsumerServiceURL="ACS"
  ProtocolBinding="HTTP-POST">
  <Issuer>IS</Issuer>
</AuthnRequest>
```

Here ID_{req} is a string uniquely identifying the request, IS is the issuer of the request, DS is the intended destination of this request, II_{req} is a timestamp, and ACS (Assertion Consumer Service) is the end-point of the SP. A common implementation choice is to use the `RelayState` field to carry the original URI that the client has requested. In step A2, C forwards the authentication request to IdP, which in turn challenges C to provide valid credentials. Note that in Figure 2 the authentication phase is abstractly represented by the dashed arrow as it is not in the scope of the SAML SSO standard. If the authentication succeeds, IdP builds the assertion $AuthnAssert(ID_{AA}, IS, II_{AA}, SJ, RC, ID_{req}, S_{ID}, NA, NB)$, where ID_{AA} is a string uniquely identifying the assertion, IS is the issuer, II_{AA} is a timestamp, SJ is the user C, RC is the intended consumer of the assertion, ID_{req} is a string uniquely identifying the request, S_{ID} is the session index, and NA and NB are `NotOnOrAfter` and `NotBefore` timestamps establishing the validity of the authentication assertion. The assertion is then included inside a SAML authentication response $Response(ID_{resp}, ID_{req}, DS, II_{resp}, AuthnAssert(\ldots))$, where ID_{resp} is the ID of the response, ID_{req} the ID of the request, DS the destination, and II_{resp} is the timestamp of the operation. Then, the response is properly encoded, placed in an HTML form equipped with a self-submitting client-side script, and returned in an HTTP 200 response to the client (step A3). Finally, C transmits back the response to SP (step A4), SP checks its validity of the assertion and if these checks are successful then sends the resource to C (step S2).

3 Model Checking

We specified SAML SSO using ASLan [7], one of the specification languages developed in the context of the AVANTSSAR Project (`www.avantssar.eu`). For

Table 1. Facts and their informal meaning

Fact	Meaning
$\texttt{state}_r(j, a, [e_1, \ldots, e_p])$	a, playing role r, is ready to execute the protocol step j, and $[e_1, \ldots, e_p]$, for $p \geq 0$ is a list of expressions representing the internal state of a.
$\texttt{sent}(rs, b, a, m, c)$	rs sent message m on channel c to a pretending to be b.
$\texttt{ik}(m)$	The intruder knows message m.

the sake of brevity in this paper we present a simplified version of ASLan, featuring only the aspects of the language that are relevant for this work. ASLan supports the specification of model checking problems of the form $M \models \phi$, where M is a labeled transition system modeling the behaviors of the honest principals and of the Dolev-Yao intruder (DY)[1] and their initial state I, and ϕ is a Linear Temporal Logic (LTL) formula stating the expected security properties. (See [3] for the details). The states of M are sets of ground (i.e. variable-free) *facts*, i.e. atomic formulae of the form given in Table 1. Transitions are represented by *rewrite rules* of the form $(L \xrightarrow{rn(v_1,\ldots,v_n)} R)$, where L and R are finite sets of facts, rn is a *rule name*, i.e. a function symbol uniquely associated with the rule, and $v_1, \ldots, v_n$ are the variables occurring in L. It is required that the variables occurring in R also occur in L. The rules for honest agents and the intruder are specified in Sections 3.1 and 3.2. Here and in the sequel we use typewriter font to denote states and rewrite rules with the additional convention that variables are capitalized (e.g. `Client`, `URI`), while constants and function symbols begin with a lower-case letter (e.g. `client`, `hReq`).

Messages are represented as follows. HTTP requests are represented by expressions $\texttt{hReq}(mtd, addr, qs, body)$, where mtd is either the constant `get` or `post`, $addr$ and qs are expressions representing the address and the query string in the URI respectively, and $body$ is the HTTP body. Similarly, HTTP responses are expressions of the form $\texttt{hRsp}(code, loc, qs, body)$, where the $code$ is either the constant `c30x` or `c200`, loc and qs are (in case of redirection) the location and the query string of the location header respectively, and $body$ is the HTTP body. In case of empty parameters, the constant `nil` is used. For instance, the message A1 in Figure 2 is $\texttt{hRsp}(\texttt{c30x}, \texttt{IdP}, \texttt{hBind}(\texttt{aReq}(\texttt{SP}, \texttt{IdP}, \texttt{id}(\texttt{N})), \texttt{URI}), \texttt{nil})$ obtained by composing `hRsp`, `hBind` and `aReq`. $\texttt{id}(\texttt{N})$ is the unique ID of the request, `hBind` binds the `SAMLRequest` `aReq` and the `RelayState` `URI` to the location header. All the other HTTP fields are abstracted away because they are either not relevant for the analysis or not used by SAML SSO protocol.

3.1 Specification of the Rules of the Honest Agents

The behavior of honest principals is specified by the following rule:

[1] A Dolev-Yao intruder has complete control over the network and can generate new messages both from its initial knowledge and the messages exchanged over the network.

$$\mathtt{sent}(b_{rs}, b_i, a, m_i, c_i).\mathtt{state}_r(j, a, [e_1, \ldots, e_p]) \xrightarrow{\mathtt{send}_r^{j,k}(a,\ldots)} \mathtt{sent}(a, a, b_o, m_o, c_o).\mathtt{state}_r(l, a, [e'_1, \ldots, e'_q]) \quad (1)$$

for all honest principals a and suitable terms $b_{rs}, b_i, b_o, c_i, c_o, e_1, \ldots, e_p, e'_1, \ldots, e'_q$, m_i, m_o, and $p, q, k \in \mathbb{N}$. Rule (1) states that if principal a playing role r is at step j of the protocol and a message m_i has been sent to a on channel c_i (supposedly) by b_i, then she can send message m_o to b_o on channel c_o and change her internal state accordingly (preparing for step l). The parameter k is used to distinguish rules associated to the same principal, and role. Notice that, in the initial and final rules of the protocol, the fact $\mathtt{sent}(\ldots)$ is omitted in the left and right hand sides of the rule (1), respectively. For instance, the reception of the message A1 in Figure 2 by the client and the sending of the message A2 are modeled by the following rewrite rule:

$$\begin{aligned}
&\mathtt{sent}\,(\mathtt{SP1}, \mathtt{SP}, \mathtt{C}, \mathtt{hRsp}(\mathtt{c30x}, \mathtt{IdP}, \mathtt{AReq}, \mathtt{nil}), \mathtt{C}_{\mathtt{SP2C}}).\\
&\quad \mathtt{state}_{\mathtt{c}}(2, \mathtt{C}, [\mathtt{SP}, \mathtt{IdP}, \mathtt{URI}, \mathtt{C}_{\mathtt{C2SP}}, \mathtt{C}_{\mathtt{SP2C}}, \mathtt{C}_{\mathtt{C2SP}_2}, \mathtt{C}_{\mathtt{SP2C}_2}, \mathtt{C}_{\mathtt{C2IdP}}, \mathtt{C}_{\mathtt{IdP2C}}])\\
&\qquad \xrightarrow{\mathtt{send}_{\mathtt{c}}^{2,1}(\mathtt{C},\mathtt{IdP},\mathtt{SP},\mathtt{SP1},\mathtt{URI},\mathtt{AReq},\mathtt{C}_{\mathtt{C2SP}},\mathtt{C}_{\mathtt{SP2C}},\mathtt{C}_{\mathtt{C2SP}_2},\mathtt{C}_{\mathtt{SP2C}_2},\mathtt{C}_{\mathtt{C2IdP}},\mathtt{C}_{\mathtt{IdP2C}})}\\
&\mathtt{state}_{\mathtt{c}}(4, \mathtt{C}, [\mathtt{SP}, \mathtt{IdP}, \mathtt{URI}, \mathtt{AReq}, \mathtt{C}_{\mathtt{C2SP}}, \mathtt{C}_{\mathtt{SP2C}}, \mathtt{C}_{\mathtt{C2SP}_2}, \mathtt{C}_{\mathtt{SP2C}_2}, \mathtt{C}_{\mathtt{C2IdP}}, \mathtt{C}_{\mathtt{IdP2C}}]).\\
&\qquad \mathtt{sent}\,(\mathtt{C}, \mathtt{C}, \mathtt{IdP}, \mathtt{hReq}(\mathtt{get}, \mathtt{IdP}, \mathtt{AReq}, \mathtt{nil}), \mathtt{C}_{\mathtt{C2IdP}}) \quad (2)
\end{aligned}$$

3.2 Specification of the Rules of the Intruder

The abilities of the DY intruder of intercepting and overhearing messages are modeled by the following rules:

$$\begin{aligned}
\mathtt{sent}(\mathtt{A}, \mathtt{A}, \mathtt{B}, \mathtt{M}, \mathtt{C}) &\xrightarrow{\mathtt{intercept}(\mathtt{A},\mathtt{B},\mathtt{M},\mathtt{C})} \mathtt{ik}(\mathtt{M}) \quad (3)\\
\mathtt{sent}(\mathtt{A}, \mathtt{A}, \mathtt{B}, \mathtt{M}, \mathtt{C}) &\xrightarrow{\mathtt{overhear}(\mathtt{A},\mathtt{B},\mathtt{M},\mathtt{C})} \mathtt{ik}(\mathtt{M}).\mathit{LHS}
\end{aligned}$$

where LHS is the set of facts occurring in the left hand side of the rule.

We model the inferential capabilities of the intruder restricting our attention to those intruder knowledge derivations in which all the decomposition rules are applied before all the composition rules [21]. The decomposition capabilities of the intruder are modeled by the following rules:

$$\begin{aligned}
\mathtt{ik}(\{\mathtt{M}\}_k).\mathtt{ik}(k^{-1}) &\xrightarrow{\mathtt{decrypt}(\mathtt{M},\ldots)} \mathtt{ik}(\mathtt{M}).\mathit{LHS} \quad (4)\\
\mathtt{ik}(\{\mathtt{M}\}_{\mathtt{K}}^s).\mathtt{ik}(\mathtt{K}) &\xrightarrow{\mathtt{sdecrypt}(\mathtt{K},\mathtt{M})} \mathtt{ik}(\mathtt{M}).\mathit{LHS} \quad (5)\\
\mathtt{ik}(f(\mathtt{M}_1, \ldots, \mathtt{M}_n)) &\xrightarrow{\mathtt{decompose}_f(\mathtt{M}_1,\ldots,\mathtt{M}_n)} \mathtt{ik}(\mathtt{M}_1).\ldots.\mathtt{ik}(\mathtt{M}_n).\mathit{LHS} \quad (6)
\end{aligned}$$

where $\{m\}_k$ (or equivalently $\mathtt{enc}(k, m)$) is the result of encrypting message m with key k and k^{-1} is the inverse key of k, $\{m\}_k^s$ (or $\mathtt{senc}(k, m)$) is the symmetric encryption, and f is a function symbol of arity $n > 0$.

For the composition rules we consider an optimisation [18] based on the observation that most of the messages generated by a DY intruder are rejected by the receiver as non-expected or ill-formed. Thus we restrict these rules so that the intruder sends only messages matching the patterns expected by the receiver [6]. For each protocol rule (1) in Section 3.1 and for each possible least set of messages $\{m_{1,l}, \ldots, m_{j_l,l}\}$ (let m be the number of such sets, then $l = 1, \ldots, m$ and $j_l > 0$) from which the DY intruder would be able to build a message m' that unifies m_i, we add a new rule of the form

$$\mathtt{ik}(m_{1,l}) \mathtt{.} \ldots \mathtt{.} \mathtt{ik}(m_{j_l,l}) \mathtt{.} \mathtt{state}_r(j, a, [e_1, \ldots, e_p]) \xrightarrow{\mathtt{impersonate}_r^{j,k,l}(\ldots)} \mathtt{sent}(\mathtt{i}, b_i, a, m', c_i) \mathtt{.} \mathtt{ik}(m') \mathtt{.} \mathit{LHS} \quad (7)$$

This rule states that if agent a is waiting for a message m_i from b_i and the intruder is able to compose a message m' unifying m_i, then the intruder can impersonate b_i and send m'.

3.3 Specifying the Authentication Property

The language of LTL we consider uses facts as atomic propositions, the propositional connectives (namely, $\neg$, $\vee$, $\wedge$, $\Rightarrow$), the first-order quantifiers $\forall$ and $\exists$, and the temporal operators **F** (eventually), **G** (globally), and **O** (once). Informally, given a formula ϕ, $\mathbf{F}\,\phi$ ($\mathbf{O}\,\phi$) holds if at some time in the future (past, resp.) ϕ holds. $\mathbf{G}\,\phi$ holds if ϕ always holds on the entire subsequent path. (See [3] for more details about LTL.) We use $\forall(\phi)$ and $\exists(\phi)$ as abbreviations of $\forall X_1 \ldots \forall X_n.\phi$ and $\exists X_1 \ldots \exists X_n.\phi$ respectively, where $X_1, \ldots, X_n$ are the free variables of the formula ϕ. We base our definition of authentication on Lowe's notion of *non-injective agreement* [19]. Thus, `SP` *authenticates* `C` *on* `URI` amounts to saying that whenever `SP` completes a run of the protocol apparently with `C`, then *(i)* `C` has previously been running the protocol apparently with `SP`, and *(ii)* the two agents agree on the value of `URI`. This property can be specified by the following LTL formula:

$$\mathbf{G}\,\forall(\mathtt{state}_{\mathtt{sp}}(7, \mathtt{SP}, [\mathtt{C}, \ldots, \mathtt{URI}, \ldots]) \Rightarrow \exists\,\mathbf{O}\,\mathtt{state}_{\mathtt{c}}(2, \mathtt{C}, [\mathtt{SP}, \ldots, \mathtt{URI}, \ldots])) \quad (8)$$

stating that, if `SP` reaches the last step 7 believing to talk with `C`, who requested `URI`, then sometime in the past `C` must have been in the state 2, in which he requested `URI` to `SP`.

Since we aim at testing implementations using attack traces as test cases with the purpose of detecting a violation of the authentication property, we would like to be sure that at the end of the execution of the attack trace, the property has been really violated. Thus, we need to take into account the testing scenario in terms of the observability of channels and of the internal states of each principal. This can be done by defining a set of observable facts. For instance, in case the tester can observe the messages passing through a channel `c` then, for all rs, b, a, and m, the $\mathtt{sent}(rs, b, a, m, \mathtt{c})$ facts are observable. Similarly, in case the

tester can observe the internal state of an agent $\mathtt{a}$, then for all r, j, e_1, ..., e_n the $\mathtt{state}_r(j, \mathtt{a}, [e_1, \ldots, e_n])$ facts are observable.

Once defined the set of observable facts according to the testing scenario, we rewrite the formula using them. For instance, let us suppose that the internal state of $\mathtt{sp}$ is not observable, while the channel $\mathtt{c}_{\mathtt{SP2C}}$ is observable, we rewrite the property (8) as follows:

$$\mathbf{G}\,\forall(\mathtt{sent}(\mathtt{SP}, \mathtt{SP}, \mathtt{C}, \mathtt{res}(\mathtt{URI}), \mathtt{c}_{\mathtt{SP2C}}) \Rightarrow \exists\,\mathbf{O}\,\mathtt{state}_{\mathtt{c}}(2, \mathtt{C}, [\mathtt{SP}, \ldots, \mathtt{URI}, \ldots])) \quad (9)$$

where $\mathtt{res}(\mathtt{URI})$ represents the resource returned by $\mathtt{SP}$ in step 7.

When the model does not satisfy the expected security property, a counterexample (i.e. an *attack trace*) is generated and returned by the model checker. A violation of the authentication property (9), as discussed in [4], is witnessed by the attack depicted in Figure 3. The attack involves four principals: a client ($\mathtt{c}$), an honest IdP ($\mathtt{idp}$), an honest SP ($\mathtt{sp}$), and a malicious service provider ($\mathtt{i}$) and comprises the following steps: $\mathtt{c}$ initiates the protocol by requesting a resource $\mathtt{uri\_i}$ at the SP $\mathtt{i}$; $\mathtt{i}$, pretending to be $\mathtt{c}$, requests a different resource $\mathtt{uri\_sp}$ at $\mathtt{sp}$ and $\mathtt{sp}$ reacts by generating an Authentication Request, which is then returned to $\mathtt{i}$; $\mathtt{i}$ maliciously replies to $\mathtt{c}$ by sending an HTTP redirect response to $\mathtt{idp}$ containing $\mathtt{aReq}(\mathtt{sp}, \mathtt{idp}, \mathtt{id})$ and $\mathtt{uri\_sp}$ (instead of $\mathtt{aReq}(\mathtt{i}, \mathtt{idp}, \mathtt{id\_i})$, and $\mathtt{uri\_i}$ as the standard would mandate); the remaining steps proceed according to the standard. The attack makes $\mathtt{c}$ consume a resource from $\mathtt{sp}$, while $\mathtt{c}$ originally asked for a resource from $\mathtt{i}$.

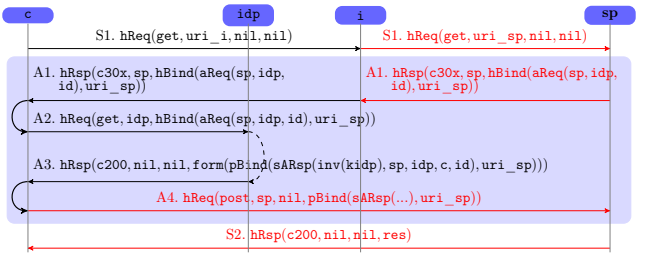


Fig. 3. Authentication Flaw of the SAML 2.0 Web Browser SSO Profile

4 Instrumentation

The model instrumentation is aimed at instructing the TEE on the generation of outgoing messages and on the checking of incoming ones. Instrumenting a model consists in calculating program fragments p associated to each rule of the model. Program fragments are then evaluated and executed by the TEE (See Section 5) in the order established by the Attack Trace.

Before providing further details we define how we relate expressions with actual messages. As seen in Section 3, messages in the formal model are specified abstractly. For instance, a SAML request $AuthnReq(IS, DS, II_{req}, ACS, ID_{req})$ is modeled by the expression $\mathtt{aReq}(\mathtt{SP}, \mathtt{IdP}, \mathtt{ID})$ thereby abstracting II_{req}. A further abstraction step is done by modeling two fields such as IS and ACS with

only one variable SP. Let D be the set of data values the messages exchanged and their fields. For instance, if $AuthnReq(is, ds, ii, acs, id)$ is an element in D, then also id, ds, ii, acs, and id are in D. Let E be the set of expressions used to denote data values in D. An *abstraction mapping* α maps D into E.

Let $D^{\perp}$ be an abbreviation for $D \cup \{\perp\}$ with $\perp \notin D$. Let f be a user defined function symbol of arity $n \geq 0$. Henceforth we consider constants as functions of arity $n = 0$. We associate f to a constructor function and a family of selector functions:

Constructor: $\overline{f} : D^n \rightarrow D$ such that $\alpha(\overline{f}(d_1, \ldots, d_n)) = f(\alpha(d_1), \ldots, \alpha(d_n))$ for all $d_1, \ldots, d_n \in D$;
Selectors: $\pi^i_f : D \rightarrow D^{\perp}$ such that $\pi^i_f(d) = d_i$ if $d = \overline{f}(d_1, \ldots, d_n)$ and $\pi^i_f(d) = \perp$ otherwise, for $i = 1, \ldots, n$.

with the following exceptions. With $K \subseteq D$ we denote the set of cryptographic keys. If $k \in K$, then $inv(k)$ is the inverse key of k. If $f =$ enc (asymmetric encryption), then

1. $\pi^1_{\mathtt{enc}}$ is undefined and
2. $\pi^2_{\mathtt{enc}} : K \times D \rightarrow D^{\perp}$, written as $decrypt$, is such that $decrypt(inv(k), d') = d$ if $d' = encrypt(k, d)$ and $decrypt(inv(k), d') = \perp$ otherwise.

If $f =$ senc, $sdecrypt$ is defined similarly as above, replacing $inv(k)$ with k. We assume that the Adapter provides constructors and selectors as program procedures. The association between symbols and procedures are specified in the Mapping (See Figure 1).

In the specification of security protocols, the behavior of the principals is represented in an abstract way, and thus the operations to check incoming messages and to generate outgoing ones are implicit. For example, in ASLan, message checks are realized by pattern matching and fields of the received message must match with some expressions stored in the state of the agent. Outgoing messages are calculated without specifying which operations are performed to compute it. Therefore, in order to interact with a system under test, we need to make explicit these procedures. We write these procedures as well as the TEE in a pseudolanguage composed of statements such as *if-then-else*, *foreach*, and the like. We also assume that the pseudolanguage has a procedure eval(p) in order to evaluate a program fragment p. Let e be a ground expression in E. We call ℓ_e a memory location in which a data value $d \in D$ is stored such that $e = \alpha(d)$.

A data value d could be the result of the evaluation of a program fragment p, i.e. $d =$ eval(p). For the sake of simplicity, in the sequel we sometimes use indifferently the data value notation and the memory location containing it. We use memory locations to refer to channels as well. Let ℓ_{c_i} and ℓ_{c_o} be two memory locations for the channel constants c_i and c_o, respectively. Besides the common operation of reading and writing on channels as memory locations, we define two operators to access them as pipes in order to send (i.e. ℓ_c >> ℓ_m) and to receive data values (i.e. ℓ_c << ℓ_m). Also, we consider a further operation to peek the first data value available in the pipe without removing it (i.e. ℓ_c |> ℓ_m). The use of the latter

operator will be clear to the reader when we explain the Instrumentation for the intruder's rules.

4.1 Instrumentation of the Rules of the Honest Agents

Let us consider the following example of ASLan rule:

$$\begin{aligned}\texttt{sent}(\texttt{A},\texttt{A},\texttt{B},\texttt{f}(\{\texttt{g}(\texttt{A},\texttt{B},\texttt{m})\}^{\texttt{s}}_{\texttt{K}},\{\texttt{h}(\texttt{A},\texttt{K})\}_{\texttt{Kb}}),\texttt{C}_{\texttt{A2B}})\,.\\ \texttt{state}_{\texttt{b}}(1,\texttt{B},[\texttt{B},\texttt{Kb},\texttt{inv}(\texttt{Kb}),\texttt{m},\texttt{C}_{\texttt{A2B}},\texttt{C}_{\texttt{B2A}}]) \xrightarrow{\texttt{send}^{1,1}_{\texttt{b}}(\texttt{B},\texttt{A},\texttt{Kb},\texttt{K},\texttt{C}_{\texttt{A2B}},\texttt{C}_{\texttt{B2A}})}\\ \texttt{state}_{\texttt{b}}(2,\texttt{B},[\ldots,\texttt{A},\texttt{K}])\,.\,\texttt{sent}(\texttt{B},\texttt{B},\texttt{A},\texttt{f}(\texttt{B},\texttt{m}),\texttt{C}_{\texttt{B2A}}) \qquad (10)\end{aligned}$$

This rule can be executed only if the message received on the channel $\ell_{C_{A2B}}$ is $\overline{\texttt{f}}(d_1,d_2))$, where d_1 can be decrypted only after having decrypted d_2, containing the data value of the decryption key K. Moreover d_1 must be $\overline{\texttt{g}}(d_3,d_4,d_5))$, where d_3 is simply stored in ℓ_A, while d_5 must be equal to $\ell_{\texttt{m}}$, and d_4 must be equal to ℓ_B, given that the variables B belongs to the internal state of the agent. As said, these checks are implicit in the ASLan semantics (pattern matching), as well as the procedure necessary to construct the message $\ell_{\texttt{f}(B,\texttt{m})}$, which is sent on the channel $\ell_{C_{B2A}}$. Nevertheless, for the testing purpose, we need to explicit these procedures. They only depend on the structure of the rule and thus can be precomputed. A program fragment $p_{\texttt{send}^{j,k}_r(a,\ldots,c_i,c_o)}$ encoding a rule (11) is as follows:

ℓ'_{m_i} := ℓ_{m_i};
ℓ_{c_i} >> ℓ_{m_i};
if ℓ'_{m_i} is not empty and ℓ_{m_i} != ℓ'_{m_i} then: return False;
eval(p_{m_i});
ℓ_{m_o} := eval(p_{m_o});
ℓ_{c_o} << ℓ_{m_o};

where m_i and m_o are the incoming and outgoing message respectively. The fragment p_{m_i} checks whether ℓ_{m_i} is such that $m_i = \alpha(\ell_{m_i})$ and p_{m_o} computes a message ℓ_{m_o} such that $m_o = \alpha(\ell_{m_o})$. In the sequel, we describe how to generate automatically p_{m_i} and p_{m_o} for a generic ASLan rule (1).

We define an association between an ASLan expression e and the fragment p used to retrieve –accessing directly to memory locations or using selectors operating on them– the corresponding data value denoted by e. We call $p : e$ an *associated expression* where $e \in E$ and p is a program fragment –containing selectors operating on memory locations– such that $e = \alpha(\text{eval}(p))$.

With reference to the send rule (11), just after the reception of ℓ_{m_i}, the knowledge of the principal is represented by the following set of associated expressions: $Ms = \{\ell_{m_i} : m_i, \ell_{e_1} : e_1, \ldots, \ell_{e_n} : e_n\}$. Given Ms we need compute the associated expressions of each sub-term of m_i.

Definition 1 (Closure under decomposition). *Given a set Ms of associated expressions, the closure of Ms under decomposition, in symbols $\downarrow Ms$, is the smallest set such that:*

1. $Ms \subseteq \downarrow Ms$,
2. *if* $p_1 : \mathtt{enc}(k, e) \in \downarrow Ms$ *and* $p_2 : \mathtt{inv}(k) \in \downarrow Ms$, *then* $(decrypt(p_2, p_1) : e) \in \downarrow Ms$,
3. *if* $p_1 : \mathtt{senc}(k, e) \in \downarrow Ms$ *and* $p_2 : k \in \downarrow Ms$, *then* $(sdecrypt(p_2, p_1) : e) \in \downarrow Ms$,
4. *if* $p : f(e_1, \ldots, e_n) \in \downarrow Ms$, *then* $(\pi_f^j(p) : e_j) \in \downarrow Ms$ *for* $j = 1, \ldots, n$.

Let us provide an example of closure. With reference to the rule (10), the set Ms contains the associated expression for the incoming message $\ell_{\mathtt{f(senc(\ldots),enc(\ldots))}} : \mathtt{f}(\mathtt{senc}(K, \mathtt{g}(A, B, \mathtt{m})), \mathtt{enc}(Kb, \mathtt{h}(A, K)))$ and other expressions known by the agent $\ell_B : B$, $\ell_{Kb} : Kb$, $\ell_{\mathtt{inv}(Kb)} : \mathtt{inv}(Kb)$, $\ell_{\mathtt{m}} : \mathtt{m}$, $\ell_{C_{A2B}} : C_{A2B}$, and $\ell_{C_{B2A}} : C_{B2A}$. By definition $\downarrow Ms$ contains Ms and other associated expressions. For example, we have $\ell_{\mathtt{f(senc(\ldots),enc(\ldots))}} : \mathtt{f}(\mathtt{senc}(\ldots), \mathtt{enc}(Kb, \mathtt{h}(A, K))) \in Ms \subseteq \downarrow Ms$ then $\pi^1_{\mathtt{f}}(\ell_{\mathtt{f(senc(\ldots),enc}(Kb,\mathtt{h}(A,K)))}) : \mathtt{senc}(\ldots)$ and $\pi^2_{\mathtt{f}}(\ell_{\mathtt{f(senc(\ldots),enc}(Kb,\mathtt{h}(A,K)))}) : \mathtt{enc}(Kb, \mathtt{h}(A, K))$ are in $\downarrow Ms$ (case 4 of the definition). Given that $\ell_{Kb} : Kb$ is in $\downarrow Ms$, the case 2 is applicable, thus $decrypt(\ell_{\mathtt{inv}(Kb)}, \pi^2_{\mathtt{f}}(\ldots)) : \mathtt{h}(A, K) \in \downarrow Ms$ as well. The example can be easily extended to the other sub-terms of the message. However, it already clarifies why we need the closure of the knowledge. Indeed, the first part of the message $\mathtt{f}(\ldots)$ is encrypted with K and it can be decrypted only after having decrypted the second part, containing the key K. Notice that, for the sake of simplicity, in this paper we assume atomic keys. Nevertheless the approach described can be readily generalized to support composed keys.

After having computed all the associated expressions, we need to either check or store the data values, according to the list of expressions representing the internal state of the principal. With reference to the send rule (1), let $kn = \{e_1, \ldots, e_n\}$, and $Ms' = \downarrow Ms - \{\ell_{e_1} : e_1, \ldots, \ell_{e_n} : e_n\}$.

Definition 2 (Atomic checks). *The set of* atomic checks P_{m_i} *for a message* $m_i \in E$ *over a knowledge* kn *is defined as follows:*

1. *for each* $p : e$ *in* Ms', *if either* e *is a constant or* e *is a variable, and* $e \in kn$ *then the following fragment is in* P_{m_i}*:*
   `if` eval(p) `!=` ℓ_e `then: return false;`
2. *for each* $p_1 : e, \ldots, p_n : e$ *in* Ms', *if* e *is a variable, and* $e \notin kn$ *then the following fragment is a member of* P_{m_i}*:*
   ℓ_e `:=` eval(p_1)`;`
   `if (`ℓ_e`!=`eval(p_2) `or` ℓ_e`!=`eval(p_3) `or ... or` ℓ_e`!=`eval(p_n)`)`
   `then: return false;`

For instance, let us consider the rule (10), the following checks are in $P_{\mathtt{f}(\ldots)}$:

1. `if` eval($\pi^3_{\mathtt{g}}(sdecrypt(\pi^2_{\mathtt{h}}(\ldots), \pi^1_{\mathtt{f}}(\ldots))))$ `!=` $\ell_{\mathtt{m}}$ `then: return false;`
   `if` eval($\pi^2_{\mathtt{g}}(sdecrypt(\pi^2_{\mathtt{h}}(\ldots), \pi^1_{\mathtt{f}}(\ldots))))$ `!=` ℓ_B `then: return false;`
2. ℓ_A `:=` eval($\pi^1_{\mathtt{h}}(decrypt(\ell_{\mathtt{inv}(Kb)}, \pi^2_{\mathtt{f}}(\ldots))))$`;`
   `if (`ℓ_A`!=`eval($\pi^1_{\mathtt{g}}(sdecrypt(\pi^2_{\mathtt{h}}(\ldots), \pi^1_{\mathtt{f}}(\ldots))))$ `) then: return false;` ...

Program fragment p_{m_i} is a sequence of all the items in P_{m_i}.

Definition 3 (Message generation function). *We call* message generation function *over a set of expressions* kn *a function* MsgGen *defined as follows:*

1. $\text{MsgGen}(e) = \ell_e$ *if* $e \in kn$;
2. $\text{MsgGen}(f(e_1, \ldots, e_n)) = \overline{f}(\text{MsgGen}(e_1), \ldots, \text{MsgGen}(e_n))$

With reference to the send rule (1), the program fragment p_{m_o} is calculated by $\text{MsgGen}(m_o)$ over $kn = \{e'_1, \ldots, e'_q\}$.

4.2 Instrumentation of the Rules of the Intruder

Intercept and Overhear Rules. Let us consider the intercept rule (4) in Section 3. Let M be the message. The fragment $p_{\texttt{intercept}(A,B,M,C)}$ of pseudocode encoding the rule is as follows:

ℓ'_M := ℓ_M;
ℓ_c >> ℓ_M;
if ℓ'_M is not empty and ℓ_M != ℓ'_M then: return False;

where ℓ'_M contains the previous value (if any) in ℓ_M, before the reception of the new message. The fragment of pseudocode encoding the overhear rule (4) in Section 3 is the same as the one defined above, except from the operator |> in place of >>.

Decomposition Rules. Let us consider the rules modeling the ability of decomposing messages (i.e. `decrypt`, `sdecrypt`, and `decompose`).

The fragment of pseudocode $p_{\texttt{decrypt}(M,\ldots)}$ encoding the rule (4) is as follows:

ℓ_M := eval($decrypt(\ell_{\texttt{inv}(K)}, \ell_{\{M\}_K})$);

where M and K are two ASLan expressions for the message and the public key, $\{M\}_K$ is the asymmetric encryption of M with K, and *decrypt* is the selector function associated to `enc`. Similarly for $p_{\texttt{sdecrypt}(\ldots)}$ encoding the rule (5).

The fragment $p_{\texttt{decompose}_f(M_1,\ldots,M_n)}$ encoding the rule (6) is as follows:

ℓ_{M_1} := eval($\pi^1_f(\ell_{f(M_1,\ldots,M_n)})$);
$\vdots$
ℓ_{M_n} := eval($\pi^n_f(\ell_{f(M_1,\ldots,M_n)})$);

where $f(M_1, \ldots, M_n)$ is the message the intruder decomposes, and π^i_f for $i = 1, \ldots, n$ are the selector functions associated to the user function symbol f.

Composition Rules. Let us consider the impersonate rule (7) in Section 3. The fragment of pseudocode $p_{\texttt{impersonate}_r^{j,k,l}(\ldots)}$ encoding this rule is computed by $\text{MsgGen}(m')$ over the knowledge $kn = \{m_{1,l}, \ldots, m_{j_l,l}\}$.

5 Test Case Execution

The Test Execution Engine (TEE) takes as input a SUT Configuration, describing which principals are part of the SUT, and an Attack Trace. The operations performed by the TEE are as follows:

```
1  procedure TEE(SUT:Agent Set;[step_1,...,step_n]:Attack Trace)
2   for i:=1 to n do:
3    if not(step_i == send_r^{j,k}(a,...) and a ∈ SUT) then:
4      if not eval(p_{step_i}) then:
5       printf("Test execution failed in step %s", step_i);
6        halt;
```

The TEE iterates over the attack trace provided as input. During each iteration it checks whether the rule $step_i$ must be executed (line (3)). Namely, if $step_i$ is either an intruder's rule or a rule concerning an agent that is not under test, then the program fragment p_{step_i} is executed. If p_{step_i} is executed without any errors the procedure continues with the next step, otherwise (lines (5)–(6)) notifies that an error occurred.

6 Experimental Results

In order to assess the effectiveness of the proposed approach, we have developed a prototype of the architecture depicted in Figure 1.

We have implemented the Instrumentation, the TEE and the Adapter components in Java. The Model Checking module is the SATMC model checker tool [2] taken off-the-shelf from the AVANTSSAR Platform. The Instrumentation component takes an ASLan model and the Mapping as input. It produces program fragments in a Java class. The TEE instantiates the class and executes the attack trace as described in Section 5. The Adapter implements the constructor and selector functions defined in Section 4. For example, constructors and selectors for the HTTP protocol are available in a Java class called `adapter.Http` that is built upon the Apache HttpComponents (`http://hc.apache.org/`). Those for the SAML SSO protocol in a class called `adapter.Saml` that is based on OpenSAML (`https://wiki.shibboleth.net/confluence/display/OpenSAML/Home`). These functions are used by program fragments as described in Section 4.

We extended the formal model of the SAML SSO we developed in previous work [4] by modeling messages using ASLan expressions as seen in Section 3. We provided the formal model to the model checker together with the authentication property (9). The model checker found the attack trace depicted in Figure 3.

We have tested two Web-based SSO solutions freely available on-line, the SAML-based SSO for Google Apps (`http://code.google.com/googleapps/domain/sso/saml_reference_implementation.html`) and the SimpleSAMLphp SSO service offered by Foodle, a surveys and polls on-line service (`https://foodl.org`). We have specified two mappings, one for each solution. For example, the mapping for testing SAML-based SSO for Google Apps contains associations as $\overline{\texttt{uri}_{\texttt{sp}}}$ = "http://mail.google.com/a/ai-lab.it/h" and $\overline{\texttt{hReq}}$ = adapter.Http where $\overline{\texttt{uri}_{\texttt{sp}}}$, $\overline{\texttt{hReq}}$ are constructor functions.

We have run the prototype against the SAML-based SSO for Google Apps by using the set $\{\texttt{idp}, \texttt{sp}\}$ as SUT Configuration. The SP is the Google GMail service while the IdP is a local identity provider service at the AI-Lab. The TEE

automatically executed the attack traces till the message S2 of Figure 3 and, as expected, the message S2 contains the mailbox of the user. Therefore, the prototype was able to automatically detect the authentication flaw.

We have used the same SUT Configuration in the experiment with SimpleSAMLphp. In this case we used Foodle as SP and Feide OpenIdP identity provider (`https://openidp.feide.no`) as IdP. The execution of the attack failed when message S2 was received. The analysis of exchanged messages has revealed that SimpleSAMLphp returns an error message instead of the message S2. We identified the cause in additional checks that reinforce the binding between authentication requests and responses. These checks are based on cookies and, since the authentication request is never routed through `c`, no cookies are installed in `c`. Therefore, when `c` presents an authentication response at `sp`, it fails in restoring the local user session for `c`.

7 Related Work

Automated analysis of security protocols has been studied and several analysis tools have been developed (see e.g., [1,10,24]). Also, there have been applications of model checking to the security analysis of Web Services (e.g., [8,17,27]). These approaches mostly focus on design time verification, and fall short in validating whether the real systems satisfy the desired properties in later life stages. Model-based testing has been applied to security-relevant systems in the recent past, e.g., [25,26]. These approaches do not propose a coherent generic methodology for security testing. Also, mappings between the abstract and concrete levels are currently managed in an ad-hoc manner only [30].

Model-checkers have been already proposed for testing by interpreting counterexamples as test cases. (See [16] for a survey). However there is no systematic approach for execution and interpretation of counterexamples.

Security-specific mutation operators have been considered in order to introduce implementation-level vulnerabilities into models [12,14]. These approaches focus on detecting implementation-level vulnerabilities. They extend and complete the one we presented. Indeed, when a model is secure with respect to a security property, it is mutated by using a security-specific mutation operator. Moreover, it does not only consider logical flaw but also vulnerabilities at the implementation level.

TorX is an automated model-based testing tool that aim at improving the quality of the software in an on-the-fly manner [29]. Its architecture has a module providing a connection with the SUT in order to send input and receiving output. However, more generic approaches for implementing adapters are needed.

An approach for model-checking driven security testing is proposed in [5]. Although the approach is protocol independent, it is strictly focused on the concretization of abstract messages in order to derive concrete test cases.

The automated tool Tookan [11] is based on an approach similar to the one we described. It reverse-engineers a real PKCS#11 token to deduce its functionality, constructs a model of its API for the SATMC model checker, and then executes any attack trace found by the model checker directly on the token. Nevertheless, this approach is specific for the PKCS#11 security tokens.

8 Conclusions

In this paper we proposed an approach that supports the binding of specifications of security protocols to actual implementations through model instrumentation, and the automatic testing of real implementations against putative attacks found by a model checker. The approach consists in model checking a formal model looking for a counterexample (i.e. attack trace) violating a security property. In case an attack is returned, it calculates automatically program fragments encoding how to verify and generate protocol messages. The attack trace is interpreted and the program fragments are executed accordingly.

In order to assess the effectiveness of the proposed approach we developed a prototype and used it to test two Web-based Single Sign-On (SSO) solutions that are available on-line, namely the SAML-based SSO for Google Apps and the SimpleSAMLphp SSO service offered by Foodle. The prototype is able to successfully execute an attack on the Google service. The prototype also shows that the same attack does not succeed against the SSO service of Foodle, due to specific implementation mechanisms used by SimpleSAMLphp.

Application of our techniques on other protocols (e.g. OpenID, OAuth) is under way and confirms the viability of the approach.

References

1. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P.H., Heám, P.C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., Vigneron, L.: The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
2. Armando, A., Carbone, R., Compagna, L.: LTL Model Checking for Security Protocols. Journal of Applied Non-Classical Logics (2009)
3. Armando, A., Carbone, R., Compagna, L., Cuellar, J., Abad, L.T.: Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In: Proc. of ACM FMSE 2008 (2008)
4. Armando, A., Carbone, R., Compagna, L., Cuellar, J., Pellegrino, G., Sorniotti, A.: From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure? In: Camenisch, J., Fischer-Hübner, S., Murayama, Y., Portmann, A., Rieder, C. (eds.) SEC 2011. IFIP AICT, vol. 354, pp. 68–79. Springer, Heidelberg (2011)
5. Armando, A., Carbone, R., Compagna, L., Li, K., Pellegrino, G.: Model-checking driven security testing of web-based applications. In: Proc. of ICSTW 2010 (2010)
6. Armando, A., Compagna, L.: Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 210–225. Springer, Heidelberg (2002)
7. AVANTSSAR. Deliverable 2.1: Requirements for modelling and ASLan v.1 (2008), http://www.avantssar.eu
8. Backes, M., Mödersheim, S., Pfitzmann, B., Viganò, L.: Symbolic and Cryptographic Analysis of the Secure WS-ReliableMessaging Scenario. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 428–445. Springer, Heidelberg (2006)

9. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the art: Automated black-box web application vulnerability testing. In: 2010 IEEE Symposium on Security and Privacy, SP (2010)
10. Blanchet, B.: Automatic verification of cryptographic protocols: A logic programming approach. In: Proc. of PPDP 2003 (2003) (invited talk)
11. Bortolozzo, M., Centenaro, M., Focardi, R., Steel, G.: Attacking and fixing PKCS#11 security tokens. In: ACM Conf. on CSS
12. Büchler, M., Oudinet, J., Pretschner, A.: Security Mutants for Property-Based Testing. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 69–77. Springer, Heidelberg (2011)
13. Clark, J., Jacob, J.: A Survey of Authentication Protocol Literature: Version 1.0 (November 17, 1997), www.cs.york.ac.uk/~jac/papers/drareview.ps.gz
14. Dadeau, F., Héandam, P.-C., Kheddam, R.: Mutation-based test generation from security protocols in HLPSL. In: ICST 2011 (2011)
15. Doupé, A., Cova, M., Vigna, G.: Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In: Kreibich, C., Jahnke, M. (eds.) DIMVA 2010. LNCS, vol. 6201, pp. 111–131. Springer, Heidelberg (2010)
16. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Softw. Test., Verif. Reliab. 19 (2009)
17. Hondo, M., Nagaratnam, N., Nadalin, A.: Securing web services. IBM Systems Journal 41(2), 228–241 (2002)
18. Jacquemard, F., Rusinowitch, M., Vigneron, L.: Compiling and Verifying Security Protocols. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 131–160. Springer, Heidelberg (2000)
19. Lowe, G.: A hierarchy of authentication specifications. In: Proc. of the 10th IEEE CSFW 1997 (1997)
20. Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 167–181. Springer, Heidelberg (1996)
21. Marrero, W., Clarke, E.M., Jha, S.: Model checking for security protocols. tech. report cmu-scs-97-139. Technical report, CMU (May 1997)
22. Millen, J.K., Shmatikov, V.: Constraint solving for bounded-process cryptographic protocol analysis. In: Proc. of ACM CCS 2001 (2001)
23. OASIS. SAML V2.0 (2005), http://docs.oasis-open.org/security/saml/v2.0/
24. Ryan, P., Schneider, S., Goldsmith, M., Lowe, G., Roscoe, B.: Modelling and Analysis of Security Protocols. Addison Wesley (2000)
25. Salas, P.A.P., Krishnan, P., Ross, K.J.: Model-based security vulnerability testing. In: Australian Software Engineering Conf., pp. 284–296 (2007)
26. Salas, P.A.P., Krishnan, P.: Testing privacy policies using models. In: Proc. of SEFM 2008 (2008)
27. Salaün, G., Bordeaux, L., Schaerf, M.: Describing and reasoning on web services using process algebra. In: Proc. of ICWS 2004 (2004)
28. Shmatikov, V., Mitchell, J.C.: Finite-state analysis of two contract signing protocols. Theoretical Computer Science 283(2), 419–450 (2002)
29. Tretmans, G.J., Brinksma, H.: Torx: Automated model-based testing. In: First European Conf. on Model-Driven Software Engineering
30. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing. Technical report, University of Waikato, New Zealand

Using Coverage Criteria on RepOK to Reduce Bounded-Exhaustive Test Suites

Valeria Bengolea[1,4], Nazareno Aguirre[1,4],
Darko Marinov[2], and Marcelo F. Frias[3,4]

[1] Department of Computer Science, FCEFQyN,
Universidad Nacional de Río Cuarto, Argentina
{vbengolea,naguirre}@dc.exa.unrc.edu.ar
[2] Department of Computer Science, University of Illinois at Urbana-Champaign, USA
marinov@illinois.edu
[3] Department of Software Engineering,
Buenos Aires Institute of Technology, Argentina
mfrias@itba.edu.ar
[4] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract. Bounded-exhaustive exploration of test case candidates is a commonly employed approach for test generation in some contexts. Even when small bounds are used for test generation, executing the obtained tests may become prohibitive, despite the time for test generation not being prohibitive. In this paper, we propose a technique for reducing the size of bounded-exhaustive test suites. This technique is based on the application of coverage criteria on the representation invariant of the structure for which the suite was produced. More precisely, the representation invariant (which is often implemented as a `repOK` routine) is executed to determine how its code is exercised by (valid) test inputs. Different valid test inputs are deemed equivalent if they exercise the `repOK` code in a similar way according to a white-box testing criterion. These equivalences between test cases are exploited for reducing test suites by removing from the suite those tests that are equivalent to some test already present in the suite.

We present case studies that evaluate the effectiveness of our technique. The results show that by reducing the size of bounded-exhaustive test suites up to two orders of magnitude, we obtain test suites whose efficacy measured as their mutant-killing ability is comparable to that of bounded-exhaustive test suites.

1 Introduction

Testing is the primary approach to detect bugs in software. It consists of executing a piece of software under assessment for a variety of test cases. These cases often correspond to instantiating parameters of the software with different inputs. Moreover, in order to increase the chances of detecting bugs, one typically seeks these inputs to be as many and as varying as possible [19].

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 19–34, 2012.
© Springer-Verlag Berlin Heidelberg 2012

An essential task in testing is test-input generation. It is a difficult task because one has to come up with inputs exercising the software in many different ways, and it has been typically done manually. In the last few years, various approaches and tools have been developed to perform *automated* test-input generation. A particularly challenging task is generating test inputs for code that manipulates complex data structures, e.g., directed graphs or AVL trees, because these inputs need to satisfy complex constraints to be valid. In this and other related contexts, the *bounded-exhaustive* exploration of possible inputs is an approach that has been quite successful [2,7,10,13,17]. This technique consists of generating all the inputs that satisfy the constraints corresponding to the well-formedness of the generated structures, within certain prescribed bounds. Tools following this approach usually involve some form of constraint-solving process, e.g., based on search, model checking, or combinations of these.

The rationale behind bounded-exhaustive testing dwells on the *small-scope hypothesis* [8], which conjectures that (in some contexts) if a program has bugs, then most of these bugs can be reproduced using small inputs. However, the exploration of all possible structures within the given bounds is a costly task that, even for small scopes, may produce very large test suites. Moreover, the time required to execute the obtained test suite may be many times prohibitive. For instance, for testing a merge routine on binomial heaps, the bounded exhaustive test-suite bounded by 6 nodes for each binomial heap has 57,790,404 tests. Also, there are situations where larger scopes are necessary to achieve coverage and detect bugs, e.g., some insertion/deletion processes in balanced trees require structures of larger sizes to force rotations or enable other rebalancing mechanisms.

In this paper we propose a technique for reducing the size of bounded-exhaustive test suites. This technique is based on the application of coverage criteria on the representation invariant of the structure for which the suite was produced. More precisely, the representation invariant, i.e., the constraint indicating whether a structure is well-formed or not, is employed to define an equivalence relation between valid test inputs. The technique requires the representation invariant to be provided as a `repOK` routine [11], and consists of analysing how the code of this routine is exercised by different test inputs. Different valid test inputs will be considered equivalent if they exercise the `repOK` code in a similar way, according to some white-box testing criterion. These equivalences between test cases are exploited for filtering tests, leaving out of the suite those tests that are equivalent to some test already present in the suite.

Essentially, our proposal involves the definition of a *black-box testing criterion with respect to the code under test*, defined in terms of *white-box testing criteria with respect to the representation invariant for the inputs of the code under test*. Namely, our criterion specifies when two different inputs are to be considered equivalent disregarding the structure of the code under test (hence, black-box), by considering only the structure of `repOK` routine (hence, white-box). We present a particular application of this criterion to the reduction of bounded-exhaustive test suites for *imperative/executable* representation invariants. However, the approach presented in this paper can also be adapted to *declarative*

representation invariants, which are becoming popular in various object-oriented languages, e.g., invariants as specified in Eiffel or via contract languages such as JML [3] and Code Contracts [4]; the adaptation is straightforward when these invariants are involved in run-time contract-checking environments, where they are made "executable" and the code corresponding to their run-time evaluation would correspond to an imperative `repOK` routine.

To assess the effectiveness of the reduced test suites produced using our approach, we present some case studies comparing bounded-exhaustive suites with suites whose size is reduced employing a variety of white-box testing criteria on `repOK`, for various data structures. We find that the reduction of up to two orders of magnitude still largely preserves the mutant-killing capability of test suites for various operations on these data structures.

2 Preliminaries

Test Coverage Criteria. A test coverage criterion is a means for measuring how well a test suite exercises a program under test. Coverage criteria are mainly classified into *black-box* and *white-box* [6,19]; the former disregard the structure of the program under test, while the latter may pay special attention to the structure of the program under test. Black-box coverage criteria "see" the code under test as a black box, taking into consideration only the specification of the program. An example of a known black-box criterion is equivalence partitioning coverage, which consists of partitioning the space of program inputs into equivalence classes, defined in terms of the specification of the expected inputs for the program under test. White-box coverage criteria analyse the program under test, and how the tests in the test suite exercise it, in order to measure coverage. A simple well-known white-box coverage criterion is decision coverage, which, in order to be satisfied, requires each decision point in the program under test (conditions in if-then-else statements, loops, etc.) to evaluate to true and false when different tests in the suite are exercised.

Test-Input Generation for Complex Structures. In the context of test-input generation for complex structures, two approaches can be distinguished, the *generative* approach and the *filtering* approach [7]. The former works by generating instances of the input structure by calling a *generator* routine, that combines calls to constructors and insertion routines on the structure. The latter builds candidate structures using only its structural definition, and then employs a predicate that characterises valid structures, known as a representation or class invariant, in order to filter out the invalid candidates. The representation invariant can be defined declaratively, e.g., using some contract-specification language such as JML [3], or operationally, i.e., via a routine that, when applied to a candidate, returns true if and only if the candidate is a valid one. The latter are typically called `repOK` routines [11]. As put forward in [11], developers should equip their complex structures implementations with `repOK` routines, since these routines will greatly help in debugging the implementations.

Bounded-Exhaustive Testing. Bounded-exhaustive testing is a testing technique that has proved useful in certain testing contexts, in particular, testing code that manipulates complex data structures. Examples of such code include libraries of data structures such as AVL trees, graphs, linked lists, etc., and programs that manipulate source code (where source code can be viewed as data with a complex structure) such as compilers, type checkers, refactoring engines, etc.

Bounded-exhaustive testing produces, for a given program under test and a user-provided bound k on the size of inputs, all valid inputs whose size is bounded by k, and then tests the program using the produced test suite. The rationale behind the approach is that many bugs in programs manipulating complex structures can be reproduced using small instances of the structure. Thus, by testing the program on all possible structures bounded in size by some relatively small scope one would be able to exhibit many bugs.

3 Reducing Bounded-Exhaustive Test Suites

In this section, we present an approach to help in reducing bounded-exhaustive test suites. The approach assumes that we have an imperative implementation of the representation invariant of the structure for which the bounded-exhaustive suite was produced; thus, it fits better with filtering approaches to test generation (for which such a representation invariant is often a requirement). The reduction process works by defining a family of coverage criteria and employing the `repOK` routine (i.e., the imperative implementation of the representation invariant) to define an equivalence between inputs. Then, according to some reduction rate on the bounded-exhaustive suite, test cases are discarded if they are "equivalent" to some test cases remaining in the suite.

To describe how the technique works, let us first describe how we define coverage criteria using `repOK`. Let C be a class, and let `repOK` be a parameterless boolean imperative routine, characterising the representation invariant of C. The representation invariant is the property that distinguishes well-formed instances from ill-formed ones. A property expected of C is that its constructors must establish `repOK` after their execution, and public methods of C must preserve it. As an example, let us consider the following Java classes, implementing binary trees of integers:

```
public class BinaryTree {
    private Node root;
    private int size;

    ...
}
```

```
public class Node {
   private int key;
   private Node left;
   private Node right;

   ...
   // setters and getters
   // of the above fields
   ...
}
```

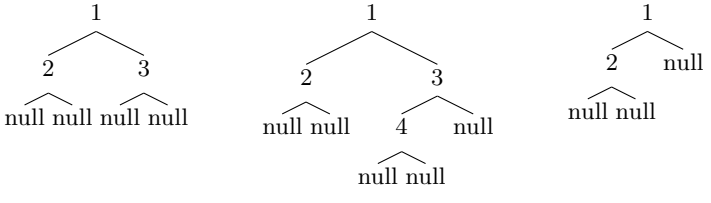
The representation invariant for this class should check that the linked structure starting with `root` is indeed a tree, i.e., that it is acyclic and with a single parent

for every reachable node except the root, and that the value of `size` agrees with the number of nodes in the structure. Checking that this property holds for a binary tree object can be implemented as in the following method from class `BinaryTree` (taken from the examples distributed with the Korat tool [2]):

```
public boolean repOK() {
    if (root == null) return size == 0;
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        if (current.getLeft() != null) {
            if (!visited.add(current.getLeft())) return false;
            workList.add(current.getLeft());
        }
        if (current.getRight() != null) {
            if (!visited.add(current.getRight())) return false;
            workList.add(current.getRight());
        }
    }
    return (visited.size() == size);
}
```

Now suppose that one needs to test a routine that receives as a parameter a binary tree, e.g., binary tree traversal routine. Notice that, as a (black-box) criterion for testing the traversal routine, we can define a partition of all possible binary tree structures according to the way the different structures "exercise" the `repOK` routine. The motivation is basically that tests that exercise the code of `repOK` in the same way can be considered as similar, and therefore can be thought of as corresponding to the same class.

We still have to define what we mean by "exercise in a similar way". This can be done, in principle, by choosing any white-box coverage criterion, to be applied to `repOK`. For instance, we can consider *decision coverage* on `repOK`; in this case, two inputs to the traversing routine (the code under test) would be considered equivalent if they make the decision points in `repOK` to evaluate to the same values. Thus, for instance, of the following three trees:



the first and the second would be considered equivalent, but none of these would be equivalent to the third one (notice that, as opposed to these other two, predicate `current.getRight() != null` never evaluates to true in this case).

In general, notice that any white-box testing criterion *Crit* gives rise to a *partition* of the input space of the program under test, with each class in the partition usually capturing some path or branch condition expressed as a constraint on the inputs. Given a program under test P, a criterion *Crit*, and an input c, we will denote by $[\![c]\!]^P_{Crit}$ the partition c belongs to, i.e., the set of all inputs that exercise the code of P in the same way c does, according to *Crit*. Our technique works by defining an equivalence between inputs. Let C be a `repOK`-equipped class, and let *Crit* be a selected white-box coverage criterion. Given two valid objects c_1 and c_2 of C, i.e., two objects satisfying C's representation invariant, we will say that c_1 is equivalent to c_2 (according to `repOK` under *Crit*), if and only if $[\![c_1]\!]^{\texttt{repOK}}_{Crit} = [\![c_2]\!]^{\texttt{repOK}}_{Crit}$.

In the above example we picked one of the simplest white-box coverage criteria to be applied to `repOK`; of course, choosing more sophisticated coverage criteria (e.g., path coverage, condition coverage, MCDC, etc.) would yield finer grained equivalence relations on the state space of the input data type.

Once one has decided the white-box criterion to be applied to `repOK`, one can use it to reduce bounded-exhaustive suites. The approach we followed for doing so is the following. Suppose that you have used some mechanism for generating a bounded-exhaustive test suite, to be used for testing, with N tests in it. Moreover, you have realised that you will not have enough resources to analyse the program under test for all these cases. Instead, you have resources to test your system for a fraction of this suite, let us say $N/10$. In this case, we do as follows:

- Determine the number of possible equivalence classes of inputs (depends both on the white-box criterion chosen on `repOK` and the complexity of `repOK`'s code).
- Set a maximum max_q for the number of tests for every single equivalence class q. For instance, divide the size of the test suite to be built (in the example $N/10$) by the number of equivalence classes, and set this as a maximum.
- Process the bounded-exhaustive test suite, leaving at most max_q tests for each equivalence class q of inputs.

As we mentioned, the result of applying the above process strongly depends on the selected white-box criterion. Moreover, this process strongly depends on the structure of the `repOK` routine too. For instance, an if-then-else with a composite condition could alternatively be written as nested if-then-else statements with atomic conditions; such structurally different but behaviourally equivalent programs may have very different equivalence classes, for the same white-box criterion, and therefore our approach may result in different reduced suites.

4 On the Effectiveness of Reduced Test Suites

In this section we evaluate the effectiveness of test suites reduced using the approach presented in the previous section. The evaluation is based on several case studies, corresponding to analyses of various routines on selected heap-allocated data structures, namely *binomial heaps*, *binary search trees*, *doubly linked lists*,

and *red black trees*. We have used the implementation of these structures provided in the Roops benchmark [15]. We are not dealing in this paper with bounded-exhaustive generation, so the approach would work with any generation tool. It is worth mentioning however that we generated the bounded-exhaustive suites on which reductions are applied, using Korat [2]. Also, we experimented with different coverage criteria on `repOK`, in order to perform the reductions. We selected three coverage criteria: *decision coverage*, *path coverage* and a variant of decision coverage, that we call *counting decision coverage*. Notice that, since we are comparing with bounded-exhaustive suites, we are able to determine precisely which are the coverable equivalence classes for each criterion (e.g., we are able to determine precisely which `repOK` paths the bounded-exhaustive suites cover), which is necessary for the reduction process. Of course, this requires executing `repOK` for *all* tests in the bounded-exhaustive suite, a task which would anyway be done at test generation time, prior to suite reduction and the testing of the program under test.

We also used counting decision coverage (CDC). This criterion takes into account the number of times each decision in the program evaluates to true and false. More precisely, given a program under test P and two inputs c_1 and c_2 for P, c_1 and c_2 are equivalent according to P under CDC if and only if, for every decision point *cond* in P, the number of times *cond* evaluates to true (resp. false) when P is executed for c_1 equals the number of times *cond* evaluates to true (resp. false) when P is executed for c_2. We believe CDC to be useful in our context since, in general, there is a relationship between the size of a structure and the number of times a particular decision point in the corresponding `repOK` evaluates to true or false (think of conditions inside loops). As a consequence, as the size of a structure increases, the number of equivalence classes will also increase, and hence the variety of cases in the reduced suite. For instance, while decision coverage considers as equivalent the first two trees in the example of the previous section, CDC will distinguish them.

Structure of the Experiments. We took the `repOK` code for each of the above mentioned structures, and we automatically instrumented it to obtain, from a `repOK` call on a given valid structure, the equivalence class the structure belongs to, for each of the selected criteria. We ran the instrumented `repOK` methods on tests of the bounded-exhaustive test suite to collect their equivalence class information. We then built reduced test suites that select from a bounded-exhaustive test suite some test cases for each (coverable) equivalence class corresponding to the criterion. In particular, we reduced the bounded-exhaustive test suites by one and two orders of magnitude, i.e., 10% and 1% of the starting test suite size. The test cases selected for the reduced test suite are the *first* generated/encountered test cases for each of the coverable equivalence classes. Note that other selections could be possible, e.g., randomly selecting an appropriate number of test cases for each equivalence class. The selection has been made taking at most N_r/M test cases for each equivalence class, where N_r is the size of the reduced test suite (e.g., 10% of the bounded-exhaustive suite) and M is the number of equivalence classes. In both cases (10% reduction and 1% reduction), when the

bounded-exhaustive test suite was too small to reduce it to 10% (or 1%) of its original size, we have taken at least one test case for each covered equivalence class.

To measure the effectiveness of the approach, we took some sample routines manipulating the data structures selected for analysis. These routines were `merge`, `insert`, `delete` and `find` for binomial heaps, `isPalindromic` for doubly linked lists, `insert`, `delete` and `search` for search trees, and `add`, `remove` and `contains` on red-black trees. We generated mutants of these routines, and measured the effectiveness of the different suites, bounded-exhaustive and reduced, in mutant killing. We also included in this assessment the "one per class" suites, consisting of exactly one test per coverable equivalence class (i.e., a minimal suite with the same coverage as the corresponding bounded-exhaustive suite). We used muJava [14] to generate mutants. The mutants we got are those obtained by the application of 12 different method-level mutation operators [12], including arithmetic, logical and relational operator replacement, when these ones were applicable to the selected routines.

We have tried to foresee potential threats to the validity of our experimental results. The case studies represent, in our opinion, typical testing situations in the context of the implementation of complex, heap allocated data structures (a main target for bounded-exhaustive testing). We chose case studies of varying complexities, including data structures with simple, intermediate, and complex constraints (e.g., linked lists, search trees and binomial heaps, respectively). Since the approach depends on the structure of `repOK`, we took implementations of these routines as provided in Korat, instead of providing our own. Also, for the evaluation we selected coverage criteria of varying complexities: the rather simple decision coverage, the more thorough path coverage, and an intermediate one, counting decision coverage.

4.1 Case Studies

Binomial Heaps (`merge`). This case study involves testing `merge`, a routine manipulating binomial heaps. This routine takes as parameters a pair of binomial heaps, and produces a binomial heap corresponding to the union of the two parameters. This is an example of a case in which the bounded-exhaustive suites quickly become too large, making bounded-exhaustive testing impractical. Figure 1 shows, for various scopes, the sizes of bounded-exhaustive (BE) suites and suites with `repOK`-based reductions to 10% and 1%, for the three mentioned white-box coverage criteria applied to `repOK`. For each criterion, it is also indicated the number of equivalence classes of inputs that have been covered (CC, for covered classes). The scope in this case specifies the maximum number of elements for both heaps, and the range for nodes' keys, from zero to the specified value. Since the bounded-exhaustive suites have been generated using Korat, these exclude symmetric cases on reference fields (Korat provides a symmetry-breaking mechanism as part of its generation process).

The `merge` routine was mutated, obtaining a total of 117 mutants. Then, the ability to kill mutants of the bounded-exhaustive, the reduced test suites

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
2,2	36	3	3	3	9	9	9	9	9	9
3,3	784	76	4	4	59	16	16	59	16	16
4,4	14,400	1200	144	4	1060	119	25	1060	119	25
5,5	876,096	49,420	7506	4	42,500	6460	36	42,500	6460	36
6,6	57,790,404	2,455,826	342,166	4	1,993,860	315,698	49	1,993,860	315,698	49

Fig. 1. Sizes of bounded-exhaustive and suites with `repOK`-based reductions, for testing binomial heap's `merge`

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
2,2	38	100	100	100	42	42	42	42	42	42
3,3	8	11	86	86	8	14	14	8	14	14
4,4	7	7	11	86	7	7	12	7	7	12
5,5	7	7	7	86	7	7	12	7	7	12
6,6	7	7	7	86	7	7	12	7	7	12

Fig. 2. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant killing for `merge` (table reports mutants remaining live)

and the minimal "one per equivalence class", was assessed. Figure 2 reports the results indicating the remaining live mutants, and highlighting the cases in which the mutation score of the reduced suites matched that of the corresponding bounded-exhaustive suite. Out of the 7 mutants that remained live with the largest bounded-exhaustive suite, 4 are equivalent to the original program. Notice that in this case, the reduced test suites for all the coverage criteria analysed were in most cases as effective as the bounded-exhaustive suites, for mutant killing, even with suites of 1% the size of the bounded-exhaustive ones.

Binomial Heaps (`insert`, `delete` and `find`). Our second case study corresponds to routines manipulating a single binomial heap, namely *insert*, *delete* and *find*. Figure 3 shows, for various scopes, the sizes of the various suites. The scopes in this case simply indicate the sizes of the corresponding binomial heaps.

Routines `insert`, `delete` and `find` were mutated (the number of mutants obtained were 99, 184 and 28, respectively), and the effectiveness of the different suites on mutant killing was assessed. Figure 4 reports the results of the analysis for this case study. Out of the 25 and 31 mutants that remained live with the largest bounded-exhaustive suite for `insert` and `delete`, 2 and 14 are equivalent to the respective original program. In this case, the reduced suites were not as effective as the previous case study, especially for the `delete` routine. However, notice that the results are still very good, taking into account the reduction in size of the suites. For instance, for scope 8 and counting decision coverage, the 10%-reduced suite only misses one mutant (32 vs. 31 out of 184) compared to the bounded-exhaustive suite.

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
2	12	3	3	3	3	3	3	3	3	3
3	84	8	4	4	8	4	4	8	4	4
4	480	40	4	4	40	5	5	40	5	5
5	4680	264	38	4	339	40	6	339	40	6
6	45,612	1938	270	4	2772	367	7	2772	367	7
7	751,912	37,650	3814	4	33,052	4947	8	33,052	4947	8
8	4,829,952	241,568	24,220	4	217,662	29,494	9	217,662	29,494	9

Fig. 3. Sizes of bounded-exhaustive and suites with `repOK`-based reductions, for testing binomial heap's operations `insert`, `delete` and `search`

Doubly Linked Lists (`isPalindromic`). Our next case study corresponds to the routine `isPalindromic`, which checks whether a given sequence of integers (implemented over a doubly linked list) is a palindrome. Figure 5 shows, for various scopes, the sizes of the various suites and the number of equivalence classes covered. The scopes in this case correspond to the number of entries in the list, the range for the size of the list, and the number of integer values allowed in the list. The routine `isPalindromic` was mutated, obtaining 23 mutants. Figure 6 reports the results of the analysis for this case study. Out of the 13 mutants that remained live with the largest bounded-exhaustive suite, 2 are equivalent to the original program. In this case study, reduced test suites are again as effective as the bounded-exhaustive ones, in most of the cases, even reduced to 1% of the size of the bounded-exhaustive ones.

Search Trees (`insert`, `delete` and `search`). Our next case study regards the data structure search trees, and the main routines for insertion, deletion and search. Figure 7 shows, for various scopes, the sizes of the various suites, and the number of covered classes. The scopes indicate the maximum number of nodes in the tree, the range for the size field of the tree, and the number of keys allowed in the tree.

Routines `insert`, `delete` and `search` were mutated (the number of mutants obtained were 9, 24 and 4, respectively). Table 8 reports the results obtained for the analysis. In this case study, reduced test suites are again as effective as the bounded-exhaustive ones, in most of the cases, with less effectiveness in the `delete` routine. Notice however that the mutant-killing score is still very good for `delete` in the reduced suites, with counting decision coverage at a 10% almost matching the bounded-exhaustive suite in scope 6,0,6,9 (2 vs. 0 out of 24 mutants).

Red-Black Trees (`remove`, `add` and `contains`) The last case study we present involves routines manipulating red-black trees. There routines are `remove`, `add` and `contains`. Figure 9 shows, for various scopes, the sizes of the corresponding suites and the number of equivalence classes covered in each case. The scopes

Scope	Oper.(#Mutants)	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
			10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
2	insert(99)	36	44	44	44	44	44	44	44	44	44
	delete(184)	124	152	152	152	152	152	152	152	152	152
	find(28)	0	12	12	12	12	12	12	12	12	12
3	insert(99)	25	26	34	34	26	34	34	26	34	34
	delete(184)	81	106	149	149	106	149	149	106	149	149
	find(28)	0	8	12	12	8	12	12	8	12	12
4	insert(99)	25	25	34	34	25	34	34	25	34	34
	delete(184)	77	99	149	149	101	149	149	101	149	149
	find(28)	0	6	12	12	6	12	12	6	12	12
5	insert(99)	25	25	25	34	25	25	34	25	25	34
	delete(184)	61	80	101	149	65	85	149	65	85	149
	find(28)	0	2	6	12	2	6	12	2	6	12
6	insert(99)	25	25	25	34	25	25	34	25	25	34
	delete(184)	33	65	99	149	48	53	115	48	53	115
	find(28)	0	2	6	12	0	3	12	0	3	12
7	insert(99)	25	25	25	34	25	25	34	25	25	34
	delete(184)	31	37	82	149	35	49	115	35	49	115
	find(28)	0	0	5	12	0	0	12	0	0	12
8	insert(99)	25	25	25	34	25	25	34	25	25	34
	delete(184)	31	54	70	149	32	48	115	32	48	115
	find(28)	0	2	5	12	0	0	12	0	0	12

Fig. 4. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant-killing for `insert`, `delete` and `search` for binomial heaps (table reports mutants remaining live)

indicate the maximum number of nodes in the tree, the range for the size field of the tree, and number of keys allowed in the tree. In this case study, paths and sizes were for some scopes too large to enable us to perform the analysis. Thus, we considered in this case study a *bounded* version of path coverage, namely path coverage without taking into account repetitions of edges (known as *simple path coverage* [19]).

Routines `remove`, `add` and `contains` were mutated (the number of mutants obtained were 142, 126 and 36, respectively), and the results of the analysis are reported in Figure 10. Out of the 41, 36 and 6 mutants that remained live with the largest bounded-exhaustive suite for `remove`, `add` and `contains`, respectively, 4, 11 and 4 are equivalent to the respective original program. In this case study, reduced test suites showed better effectiveness for the `contains` routine, matching in many cases the mutant-killing score of the bounded-exhaustive suites. For the other two routines it was not the same case, although they achieved a very good mutant-killing score in many cases (e.g., counting decision coverage for `add` in scope 7,0,7,7 missed only 4 out of 126 compared to the bounded-exhaustive suite).

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
4,0,4,4	156	8	2	2	10	4	4	10	4	4
4,0,4,8	820	42	5	2	50	7	4	50	7	4
5,0,5,5	1555	78	8	2	100	13	5	100	13	5
5,0,5,10	16,105	806	81	2	777	108	5	777	108	5
6,0,6,6	19,608	981	99	2	1035	136	6	1035	136	6
6,0,6,12	402,234	20,112	2,012	2	15,786	2,193	6	15,786	2,193	6
7,0,7,7	299,593	14,980	1,498	2	13,239	1,781	7	13,239	1,781	7
7,0,7,14	12,204,241	610,213	61,022	2	402,933	55,918	7	402,933	55,918	7

Fig. 5. Sizes of bounded-exhaustive suites and suites with `repOK`-based reductions, for testing `isPalindromic` operation for doubly linked lists

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
4,0,4,4	15	15	23	23	15	22	22	15	22	22
4,0,4,8	15	15	23	23	15	15	22	15	15	22
5,0,5,5	13	15	22	23	13	15	22	13	15	22
5,0,5,10	13	15	15	23	13	13	22	13	13	22
6,0,6,6	13	13	15	23	13	13	22	13	13	22
6,0,6,12	13	13	15	23	13	13	22	13	13	22
7,0,7,7	13	13	13	23	13	13	22	13	13	22
7,0,7,14	13	13	13	23	13	13	22	13	13	22

Fig. 6. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant-killing for `isPalindromic` for doubly linked lists (table reports mutants remaining live)

5 Related Work

There exist some approaches that are related to the work presented in this paper. With respect to the reduction of bounded-exhaustive test suites, the work of some of the authors of this paper [1] is strongly related to the work presented in this paper, especially because both approaches are based on the use of coverage criteria. However, the previous approach [1] differs from the work of this paper in two aspects. First, it requires the user to provide the coverage criterion to perform the suite reduction, as opposed to our work here, where the coverage criterion is a standard one applied to the representation invariant. Second, the previous approach targets the improvement in the *test generation process*, whereas our work in this paper concerns the reduction of bounded-exhaustive test suites to reduce the time for testing. Another work related to ours is the one presented in [9]. In [9], the authors present various techniques for reducing the costs of bounded-exhaustive testing. These techniques are sparse test generation, which attempts to reduce the time to the first failing test (but not the

Scope	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
3,0,3,3	45	5	5	5	7	7	7	9	9	9
3,0,3,4	148	10	5	5	14	7	7	9	9	9
3,0,3,6	822	70	5	5	72	7	7	78	9	9
3,0,3,8	2,760	228	25	5	242	21	7	248	27	9
5,0,5,8	29,416	1836	240	5	2634	278	16	2888	260	65
6,0,6,9	167,814	10,158	1095	5	14,430	1605	22	16,665	1576	197

Fig. 7. Sizes of bounded-exhaustive suites and suites with `repOK`-based reductions, for testing `delete`, `insert` and `search` operations of search trees

Scope	Oper.(#Mutants)	BE	Decision Cov.			Count. Decision Cov.			Path Cov.		
			10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
3,0,3,3	delete(24)	2	12	12	12	12	12	12	12	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
3,0,3,4	delete(24)	2	12	12	12	12	12	12	12	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
3,0,3,6	delete(24)	2	12	12	12	12	12	12	12	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
3,0,3,8	delete(24)	2	9	12	12	9	12	12	9	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
5,0,5,8	delete(24)	0	9	16	16	9	12	12	9	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
6,0,6,9	delete(24)	0	9	16	16	2	9	12	0	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0

Fig. 8. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant-killing for `insert`, `delete` and `search` for search trees (table reports mutants remaining live)

suite); oracle-based test clustering, which groups together failing tests to reduce the time for inspection of failing tests; and structural test merging, whose purpose is to generate smaller suites of larger tests by merging together smaller test inputs. Of these three, the latter is related to our work, since it has as a purpose to reduce the size of the test suite. However, the approach is rather different, since bounded exhaustiveness is preserved in structural test merging (although sets of small inputs are encoded as a single large input), whereas in our case we drop bounded exhaustiveness by selecting only some tests. The same differences apply to other works based on test granularity [16].

Scope	BE	Decision Cov.			Count. Decision Cov.			Simple Path Cov.		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
4,0,4,4	164	14	7	7	16	16	16	108	108	108
4,0,4,8	6408	500	62	7	608	64	16	169	157	157
5,0,5,5	575	53	7	7	30	30	30	97	97	97
5,0,5,10	56,790	2732	496	7	5313	532	30	245	165	157
6,0,6,6	1962	174	14	7	184	16	46	113	113	113
6,0,6,12	412,140	10,411	2,652	7	38,579	4,017	46	505	229	157
7,0,7,7	6377	469	61	7	570	66	66	154	142	142
7,0,7,14	3,045,266	89,960	11,654	7	284,408	29,449	66	2211	465	157

Fig. 9. Sizes of bounded-exhaustive suites and suites with `repOK`-based reductions, for testing `remove`, `add` and `contains` operations of red-black trees

Scope	Oper.(#Mutants)	BE	Decision Cov.			Count. Decision Cov.			Simple Path Cov.		
			10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
4,0,4,4	remove(142)	47	82	82	81	70	70	70	47	47	47
	add(126)	38	58	90	90	90	90	90	44	44	44
	contains(36)	6	8	9	9	9	9	9	6	6	6
4,0,4,8	remove(142)	47	66	81	81	65	70	70	82	82	82
	add(126)	38	40	43	90	40	58	90	53	62	62
	contains(36)	6	6	7	9	6	8	9	8	8	8
5,0,5,5	remove(142)	43	81	82	81	68	68	68	68	68	68
	add(126)	38	43	90	90	90	90	90	56	56	56
	contains(36)	6	7	9	9	9	9	9	8	8	8
5,0,5,10	remove(142)	43	55	77	81	52	67	68	82	82	82
	add(126)	36	40	42	90	39	43	90	49	58	78
	contains(36)	6	6	6	9	6	7	9	8	8	8
6,0,6,6	remove(142)	41	66	82	81	46	66	66	71	71	71
	add(126)	36	42	58	90	55	90	90	62	62	62
	contains(36)	6	6	8	9	7	9	9	8	8	8
6,0,6,12	remove(142)	41	53	60	81	50	61	66	82	82	82
	add(126)	36	40	40	90	37	40	90	47	49	78
	contains(36)	6	6	6	9	6	6	9	8	8	8
7,0,7,7	remove(142)	41	60	81	81	41	66	66	76	76	76
	add(126)	36	40	43	90	40	90	90	53	62	62
	contains(36)	6	6	7	9	6	9	9	8	8	8
7,0,7,14	remove(142)	41	50	55	81	44	50	66	76	82	82
	add(126)	36	42	58	90	55	90	90	47	49	79
	contains(36)	6	6	6	9	6	6	9	8	8	8

Fig. 10. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant-killing for `add`, `remove` and `contains` for red-black tree (table reports mutants remaining live)

Other researchers have studied the effects of reducing test suites in finding bugs, e.g., the work in [18]. Our work is related, but we propose a specific approach for

test-suite reduction (as opposed to studying the effects of test-suite reductions in general), and we target specifically bounded-exhaustive test suites.

6 Conclusions and Further Work

Bounded-exhaustive test suites are popular in some testing contexts, such as that of testing complex heap allocated data structures. However, in many cases bounded-exhaustive test suites become too large as the bound for the generated suites increases, thus making their (exhaustive) use impractical. We have presented an approach for reducing bounded-exhaustive test suites, and consequently also the time spent in testing using these suites, for cases in which an imperative representation invariant routine is available for the inputs for which the suites were generated. The approach works by defining black-box criteria for the program under test, based on the definition of equivalence relations of inputs, defined in terms of white-box criteria on the imperative representation invariant; basically, the rationale for this is that, if two inputs exercise the representation invariant code in the same way, according to a white-box criterion, these inputs may be considered similar, i.e., considered to belong to the same equivalence class of inputs. These equivalence classes are then employed in order to filter out of the exhaustive suites some tests that are equivalent to some others already present in the suite.

Although our motivation is the reduction of bounded-exhaustive test suites, the idea of using white-box criteria on the representation invariant is indeed the definition of a new black-box coverage criterion, for programs whose inputs count on a representation invariant. This idea can also be adapted to declarative representation invariants, which are becoming popular, e.g., invariants as specified in Eiffel or via contract languages such as JML and Code Contracts; these invariants are typically involved in run-time contract-checking environments, so they are "executable", and the code corresponding to their run-time evaluation would correspond to what we referred to as `repOK` in this paper.

We presented some case studies showing the performance of suites reduced using the above approach, compared to bounded-exhaustive suites. As the experiments show, for some white-box coverage criteria on the representation invariant, we obtain a performance in mutant killing that is comparable to that of bounded-exhaustive suites. In particular, we used a variant of decision coverage, called *counting decision coverage*, which takes into account the number of times each decision point in the program under test becomes true and false. This criterion, applied to the representation invariant, is useful in our context, since in general we observe that there is a relationship between the size of the structure and the number of times a particular decision point in the corresponding representation invariant evaluates to true or false.

As work in progress, we are currently examining the approach proposed in this paper for several additional case studies, based on more complex data structures. We also plan to assess the approach in the context of testing applications manipulating source code, such as compilers or, more particularly, refactoring engines, as is done using ASTGen [5].

References

1. Aguirre, N., Bengolea, V., Frias, M., Galeotti, J.: Incorporating Coverage Criteria in Bounded Exhaustive Black Box Test Generation of Structural Inputs. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 15–32. Springer, Heidelberg (2011)
2. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated Testing based on Java Predicates. In: Proc. of Intl. Symposium on Software Testing and Analysis ISSTA 2002. ACM Press (2002)
3. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
4. Code Contracts, http://research.microsoft.com/en-us/projects/contracts/
5. Daniel, B., Dig, D., García, K., Marinov, D.: Automated Testing of Refactoring Engines. In: Proc. of European Software Engineering Conference and Intl. Symposium on Foundations of Software Engineering ESEC/FSE 2007. ACM Press (2007)
6. Myers, G.J.: The Art of Sofware Testing, 2nd edn. John Wiley & Sons, Inc. (2004)
7. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test Generation through Programming in UDITA. In: Proc. of Intl. Conference on Software Engineering ICSE 2010. ACM Press (2010)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
9. Jagannath, V., Lee, Y.Y., Daniel, B., Marinov, D.: Reducing the Costs of Bounded-Exhaustive Testing. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 171–185. Springer, Heidelberg (2009)
10. Khurshid, S., Marinov, D.: TestEra: Specification-Based Testing of Java Programs Using SAT. Automated Software Engineering 11(4) (2004)
11. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification and Object-Oriented Design. Addison-Wesley (2000)
12. Ma, Y.-S., Offutt, J., Kwon, Y.-R.: MuJava: An Automated Class Mutation System. Journal of Software Testing, Verification and Reliability 15(2) (2005)
13. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A Tool for Generating Structurally Complex Test Inputs. In: Proc. of Intl. Conference on Software Engineering ICSE 2007. IEEE Press (2007)
14. MuJava, http://www.cs.gmu.edu/~offutt/mujava/
15. Roops, http://code.google.com/p/roops/
16. Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P., Davia, B.: The Impact of Test Suite Granularity on the Cost-Effectiveness of Regression Testing. In: Proc. of Intl. Conference on Software Engineering ICSE 2002. ACM Press (2002)
17. Sullivan, K., Yang, J., Coppit, D., Khurshud, S., Jackson, D.: Software Assurance by Bounded Exhaustive Testing. In: Proc. of Intl. Symposium on Software Testing and Analysis ISSTA 2004. ACM Press (2004)
18. Yu, Y., Jones, J., Harrold, M.: An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization. In: Proc. of Intl. Conference on Software Engineering ICSE 2008. ACM Press (2008)
19. Zhu, H., Hall, P., May, J.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys 29(4) (1997)

A First Step in the Design of a Formally Verified Constraint-Based Testing Tool: FocalTest

Matthieu Carlier[1], Catherine Dubois[1,2], and Arnaud Gotlieb[3]

[1] CEDRIC-ENSIIE, Évry, France
{dubois,carlier}@ensiie.fr
[2] INRIA, Paris, France
[3] Certus V&V Center, SIMULA RESEARCH LAB., Lysaker, Norway
arnaud@simula.no

Abstract. Constraint-based test data generators rely on SMT or constraint solvers to automatically generate test data (e.g., Pex, Sage, Gatel, PathCrawler, Euclide). However, for some test data generation requests corresponding to particular test objectives, these tools may fail to deliver the expected test data because they focus on efficiency rather than soundness and completeness. We adopt an opposite view in the development of FocalTest, a test data generation tool for Focalize programs. The goal of the tool is to generate an MC/DC-compliant set of test data over the precondition of user-defined program properties. The development of such a correct-by-construction test data generator requires 1) to provide a formally verified translation of Focalize programs and properties into constraint systems; 2) to introduce a formally verified constraint solver able to solve those constraint systems. This paper is concerned with the first step only where we formally demonstrate with Coq the soundness of the translation of an intermediate functional language into a constraint system. This objective requires to formally define the operational semantics of the source language that features the manipulation of concrete data types via pattern-matching and function calls, constructions that are mirrored in the constraint language. Although such a semantics-oriented formalization is only a first step of a larger goal which is to provide a formally verified constraint-based testing tool, we argue that it is an important contribution to the building of more robust software testing tools.

1 Introduction

A new trend in software testing consists in using constraint solvers or SMT-solvers to generate test inputs that satisfy a given test objective. The idea of using constraint (logic) programming to capture the concrete semantics of programs written in other languages is not new. In the early 2000s, Podelski proposed using constraint solving procedures to deal with general infinite-state systems [18], while Flanagan more specifically addressed imperative languages [10]. These pioneering works built the foundational layout for opening the door to concrete approaches implemented in tools such as InKa [13] for C or GATEL

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 35–50, 2012.
© Springer-Verlag Berlin Heidelberg 2012

[17] for Lustre, and more recently Euclide [12,7] for C, JAUT [6] and PET [11] for Java Bytecode. However, as far as we know, the question of proving the soundness and completeness of these translations, far from trivial in general, has not yet really been addressed. Soundness means here that a possible execution of the program is a solution of the constraint system and conversely completeness states an assignment, solution of the constraint system is a possible execution of the corresponding program.

This paper tackles this objective in the context of the constraint-based testing tool FocalTest [4,5] developed by the authors, that allows to test programs written with the Focalize language [1]. FocalTest implements property-based testing of programs and thus implies to select test data that satisfy MC/DC (i.e., Modified Condition/Decision Coverage) on the precondition of the user-defined property under test. MC/DC is a well-known structural coverage criterion defined for tackling the complexity of decision in the civil avionics domain [14]. Please consult [4,5] for comparable testing tools or related works. The source language is a functional language that features the definition and manipulation of concrete data types via pattern-matching, higher order and function calls, constructions that are mirrored in the constraint language. In this paper we describe formally the translation of the program and the property under test into a system of constraints and demonstrate its soundness and completeness. In our context, a property is an implication between a precondition and a conclusion (in an algebraic like setting), a precondition being mainly a set of conditions also called decisions. Our objective requires to formally define the operational semantics of both the source language and the constraint language. Both soundness and completeness ensure that when a solution of the constraint system representing the precondition of the property under test is found, then this solution is indeed possible and satisfies the precondition. Conversely, if there exists a possible evaluation that satisfies the precondition the corresponding assignment is necessarily a solution of the constraint system. A machine–checked formalization and proof of soundness of the translation has be realized in Coq [8] and is available as a side part of this paper. A nice result is that a translator written in ML, which preserves semantics, has been extracted from this Coq code. More generally the work presented in this paper prepares the ground for addressing a broader objective, which consists in developing a formally-proved constraint-based testing tool. In the context of FocalTest, this requires 1) to provide a formally verified translation of Focalize programs and properties into constraint systems which is the topic of this paper; 2) to introduce a formally verified constraint solver able to solve those constraint systems.

In the context of testing, few formalizations based on operational semantics (and even fewer machine-checked ones) have been explored. We can cite Bruckner and Wolff' work [2] where the authors formalize and verify in Isabelle some white-box test techniques for a small imperative language. In [19], Wotawa and Nica detail a constraint representation of imperative programs which ends with the proposition of a soundness theorem (theorem 3.3) close to ours. However none of these works proposed a formalization as advanced as the one given in this paper.

Note however that in the general setting of programming languages, a lot of such proof of soundness and completeness exist, e.g., the famous and substancial proof of semantics preservation for Compcert, a compiler for a large subset of C [15].

The paper is organized as follows: Sec. 2 briefly presents Focalize and FocalTest and gives the necessary background to understand our approach. The translation into constraints is done in two steps and uses an intermediate language, a form of monadic language, called FMON. The paper focuses on the second step, however Sec. 3 gives some insight into the first step. Sec. 4 formally describes the second step: syntax and semantics of both FMON and the constraint language and then the translation are detailed. Sec. 5 focuses on soundness and completeness of the translation and we precise the outline of the machine-checked proof in Coq. Finally, Sec. 6 concludes the paper.

2 A Brief Presentation of Focalize and FocalTest

This section briefly presents the technical background necessary to understand what follows.

Focalize (previously named Focal, `http://focalize.inria.fr`) is an environment allowing the development of programs step by step, from specification to implementation. This environment proposes a language also named Focalize and tools to analyze the code -in particular its dependencies-, to compile into various formats -Ocaml executable code, Coq code, HTML representation and UML class diagrams-, to prove properties (e.g., that the implementation satisfies its specification). In our context a specification is a set of algebraic properties describing relations between input and output of the functions implemented in a Focalize program. Focalize is a strongly typed functional language (close to ML) and offers mechanisms inspired by object-oriented programming, e.g., inheritance and late binding to ease modularization and reuse. Besides basic types, Focalize allows the programmer to introduce new datatypes, called in the sequel *algebraic datatypes* or simply *datatypes* for short, defined by their value constructors (with a fixed arity, constant if arity is null). Focalize also offers a convenient mechanism to explore and de-structure values of datatypes by pattern matching, also known as case analysis ($\texttt{match}\ x\ \texttt{with}\ pat_1 \rightarrow e_1 \mid \ldots \mid pat_n \rightarrow e_n$). It also includes mutually recursive functions, local binding ($\texttt{let}\ x = e_1\ \texttt{in}\ e_2$), conditionals ($\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$) and higher-order functions.

As an example, consider the Focalize program of Fig. 1 where *app* (append) and *rev* (reverse) both are user-defined functions. Lists are here built from 2 constructors: the constant constructor *nil* (empty list) and the binary recursive constructor *cons* (adding a new value at the beginning of a list). The property called *rev_prop* simply says that reversing a list can be done by reversing its sub-lists. A more thorough overview of Focalize can be found in [1].

The authors have developed and integrated into the environment Focalize the FocalTest tool that allows the user to test whether his programs meet their specifications or some user-defined properties. FocalTest implements the property-based testing technique, which is a general testing technique that uses property

```
let rec app(l, g) =
      match l with
            nil → g
            | cons(h, t) → cons(h, app(t, g));

let rec rev_aux(l, ll) =
      match l with
            nil → ll
            | cons(h, t) → rev_aux(t, cons(h, ll));
let rev(l) = rev_aux(l, nil);

property rev_prop :
      all l l1 l2 : list(int),
      l = app(l1, l2) → rev(l) = app(rev(l2), rev(l1));
```

Fig. 1. A Focalize program

specifications to select test cases and guide evaluation of test executions [9]. It implies selecting test inputs from the property under test and checking the expected output results in order to evaluate the conformance of programs w.r.t. its property specifications. FocalTest is a constraint-based test data generation tool, meaning that it translates a Focalize program into a set of constraints, and then, by calling a constraint solver, it can generate test data satisfying the precondition part of user-defined properties. This process is the subject of the formalization presented in the paper. Furthermore the constraint-based approach permits one to obtain an MC/DC-compliant test suite that covers the precondition part of any Focalize property. It implies that the tool is able to generate test inputs that satisfy or do not satisfy a decision in the precondition.

Our implementation is based on a systematic translation of Focalize programs into constraint programs (`clpfd` in SICStus Prolog more precisely). It lies on the definition of efficient user-defined constraint combinators to tackle conditionals, pattern-matching and higher-order functions. More details on the random and constraint-based approaches of FocalTest can be found in [4] and [5]. Experimental evaluation results given in [5] demonstrate that the constraint resolution process suits well, in particular the way to solve `ite` and `match` combinators.

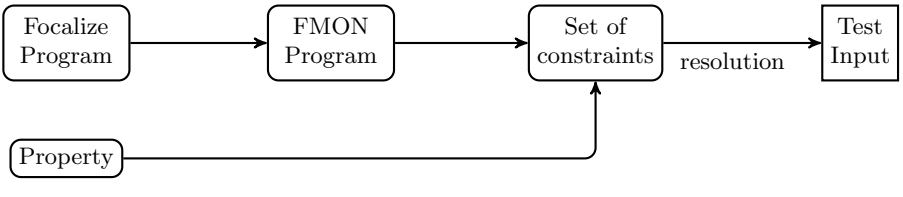


Fig. 2. Test input generation process

In the rest of the paper, we call *test unity* a set of function definitions together with concrete type definitions and the property under test. Let us suppose in the following that everything is defined and contained in the test unity (all the dependencies are present, the inheritance has been solved). It is the task of the compiler not of the testing tool. The functions may be distributed in the different components (called species in the Focalize jargon) of the Focalize program. It may also contain some undefined functions only declared. All we require is that everything needed by the functions involved in the property under test is defined. A property under test is assumed to be of the following form:

$$\forall X_1 \in T_1 \ldots X_m \in T_m,\ \underbrace{A_1 \Rightarrow \cdots \Rightarrow A_n}_{\texttt{Precondition}} \Rightarrow \underbrace{B_1 \vee \cdots \vee B_p}_{\texttt{Conclusion}}$$

where A_is and B_is are function calls. If not, it may be rewritten as a set of such properties (as explained in [4]).

The different steps to produce test data according to a test unity is illustrated by Fig. 2, where FMON (Functional MONadic) is an intermediate language for programs and properties designed to ease the translation into constraints. The first step of the process is the translation of Focalize programs into FMON, while the second step produces a set of constraints for both the FMON program and the precondition part of the property. As said previously, in this paper, we focus on the constraint system generation part, assuming that we have at hand a correct constraint solver. Furthermore, for the sake of simplicity, we consider only simple decisions from the precondition part of properties. It means we assume a correct translation from MC/DC-compliant values on decisions to simple decision true or false values. A more important current restriction concerns higher-order aspects of Focalize programs. Although, we have built a constraint translation that can handle higher-order function calls, formally proving the correction of the translation is far from trivial. We elaborate on this in Section 6.

3 From Focalize to FMON

As Focalize is a real-world language, it includes many additional features that can reasonably be discarded for a correction proof because they are irrelevant and do not introduce any particular problems. So the first step of our process is a normalization one that consists in giving names to all computation steps and to put pattern-matching expressions in a canonical form without any nested patterns. Regarding pattern-matching, the process is a Focalize-dedicated adaptation of the algorithms proposed in [16]. It is detailed in [3] and proved sound and complete w.r.t. both the Focalize semantics and the FMON language semantics. An overview of this normalization step is now given. The naming of all the computation steps is formalized by a function called $\mathcal{N}_e$ that associates an identifier to every expression evaluation (e.g., function call, condition in conditional, etc.) through the help of local bindings. Since Focalize is a pure functional programming language (i.e., no side effects), we do not have to take care about

any evaluation strategy as long as we respect the call-by-value evaluation of Focalize. For example, considering that modular code has already been flattened, any function call ($\mathcal{F}reshF$ is a set of fresh variables) can be tackled with the following:

$$\mathcal{N}_e(f(e_1,\ldots,e_n)) = \begin{array}{l} \texttt{let}\, x_1 = \mathcal{N}_e(e_1)\ \texttt{in} \\ \qquad\vdots \\ \texttt{let}\, x_n = \mathcal{N}_e(e_n)\ \texttt{in} \\ f(x_1,\ldots,x_n) \end{array} \quad (\forall i \in [\![1,n]\!],\ x_i \in \mathcal{F}reshF)$$

For Focalize user-defined properties, a dependency analysis is needed to recover all the functions involved in the precondition part, because some of the function calls may be deeply nested in recursive calls. We do not detail this calculation that may be difficult to implement efficiently, as it is somehow outside the scope of the paper. The interested reader can look at [3] to get more details.

So from here, the *test unity* is built: it is technically a bunch of function definitions with a property whose only interesting part for us is the precondition. The precondition, a set of conditions that can be seen as calls to predicates (Boolean functions), is transformed according to the same principles as Focalize programs. It means that local bindings are used to name the arguments of the called predicates.

4 From FMON to Constraints

This section formally introduces the FMON language (i.e., syntax and semantics), the target language of constraints and the translation from the former to the latter.

4.1 The FMON Language

Syntax. FMON syntax is detailed in Fig. 3. It is close to Focalize's syntax but contains some restrictions. In particular, arguments of function calls, conditions in conditionals, and matched expressions can only be variables (and not expressions), in order to prepare the translation into constraints. Furthermore, any pattern-matching expression cannot have nested patterns. It can only have non-overlapping n-ary constructors applied to variables as patterns. But, note that it can have a *catch-all* pattern (written _) as final clause. As mentioned above, using functional programming terminology, FMON only contains *named abstractions* and *complete applications*. Furthermore functions cannot have functions as arguments or results.

A FMON program is a list of function definitions stored in a function environment which can be considered as a partial function relating a function identifier with its closure $<x_1,\ldots,x_n \rightsquigarrow e>$. In this notation, x_1, ..., x_n are the bound

variables of the function and the expression e its body (where free variables are not authorized). Furthermore functions can call each other and are implicitly mutually recursive.

e	::=	$\texttt{let}\ x = e\ \texttt{in}\ e$	local binding
	|	$\texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e$	conditional
	|	$\texttt{f}(x_1, \ldots, x_i)$	function call
	|	$op\,(x_1, x_2)$	basic bin. operator
		$\texttt{match}\ x\ \texttt{with}$	
		$\mid pat \rightarrow e$	
	|	$\vdots$	pattern matching
		$\mid pat \rightarrow e$	
		$[\mid \_ \rightarrow e]$	
	|	$i \mid b$	integer/boolean
	|	$C(x_1, \ldots, x_n)$	constructors
	|	x	variable
pat	::=	$C(pat\_arg, \ldots, pat\_arg)$	constructor pattern
pat_arg	::=	$x \mid \_$	variable pattern/catch-all

Fig. 3. FMON syntax

Semantics. Operational semantics of FMON expressions is described with inference rules (see Fig. 4). The evaluation judgement is $\mathcal{E}; \mathcal{E}_\texttt{f} \vdash e \triangleright v$ where $\mathcal{E}_\texttt{f}$ is the function environment that associates a closure to each function identifier and $\mathcal{E}$ is the evaluation context that associates free variables of e to their values. Such a judgement tells us that e evaluates to v, provided that the evaluation context is $\mathcal{E}$ and the function environment is $\mathcal{E}_\texttt{f}$. Rule APP evaluates a function call by evaluating the arguments and then evaluating the body of the function in the context that binds each parameter to its corresponding value. Rules MATCH and CATCH concern pattern-matching. The former formalizes the case when the value v of the matched variable x matches the constructor C_i, then the value of the entire expression is the value of expression e_i. The latter corresponds to the situation where v does not match any of the listed constructors but a $catch - all$ branch exists. In both rules, $\mathcal{F}$ checks whether a value matches a given pattern. When it happens, the function returns the evaluation context that binds the variables of the pattern; otherwise it fails (`nok`). The $\oplus$ operator is used to update environments and contexts (and later assignments). In rule OP, $[\![op, v_1, v_2]\!]$ denotes the interpretation of operator op on values v_1 and v_2.

4.2 The Constraint Language

Introduction and Syntax. Firstly, the syntax of our constraint language is given in Fig. 5. A *constraint system* (also called *constraint store*) is mainly composed of integer constants, terms, finite-domain variables, algebraic variables

$$\frac{}{\mathcal{E};\mathcal{E}_\mathsf{f} \vdash b \rhd b}\ \text{BOOL} \qquad \frac{}{\mathcal{E};\mathcal{E}_\mathsf{f} \vdash i \rhd i}\ \text{INT}$$

$$\frac{\mathcal{E}(x) = v}{\mathcal{E};\mathcal{E}_\mathsf{f} \vdash x \rhd v}\ \text{VAR} \qquad \frac{}{\mathcal{E};\mathcal{E}_\mathsf{f} \vdash C(x_1,\ldots,x_n) \rhd C(\mathcal{E}(x_1),\ldots,\mathcal{E}(x_n))}\ \text{NC}$$

$$\frac{\mathcal{E}(x) = \texttt{true} \qquad \mathcal{E};\mathcal{E}_\mathsf{f} \vdash e_1 \rhd v_1}{\mathcal{E};\mathcal{E}_\mathsf{f} \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd v_1}\ \text{IFT} \qquad \frac{\mathcal{E}(x) = \texttt{false} \qquad \mathcal{E};\mathcal{E}_\mathsf{f} \vdash e_2 \rhd v_2}{\mathcal{E};\mathcal{E}_\mathsf{f} \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd v_2}\ \text{IFF}$$

$$\frac{[\![op\,, \mathcal{E}(x_1), \mathcal{E}(x_2)]\!] = v}{\mathcal{E};\mathcal{E}_\mathsf{f} \vdash op\,(x_1,x_2) \rhd v}\ \text{OP} \qquad \frac{\begin{array}{c}\mathcal{E}_\mathsf{f}(\mathsf{f}) = <x_1,\ldots,x_n \rightsquigarrow e> \\ (x_1,\mathcal{E}(x_1)),\ldots,(x_n,\mathcal{E}(x_n));\mathcal{E}_\mathsf{f} \vdash e \rhd v\end{array}}{\mathcal{E};\mathcal{E}_\mathsf{f} \vdash \mathsf{f}(x_1,\ldots,x_n) \rhd v}\ \text{APP}$$

$$\frac{\begin{array}{c}\mathcal{E};\mathcal{E}_\mathsf{f} \vdash e_1 \rhd v_1 \\ \mathcal{E} \oplus (xv_1);\mathcal{E}_\mathsf{f} \vdash e_2 \rhd v_2\end{array}}{\mathcal{E};\mathcal{E}_\mathsf{f} \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rhd v_2}\ \text{LET}$$

$$\frac{\begin{array}{c}\mathcal{E}(x) = v \qquad \mathcal{F}(pat_i, v) = \mathcal{E}' \\ \forall j \neq i,\ \ \mathcal{F}(pat_j, v) = \texttt{nok} \\ \mathcal{E} \oplus \mathcal{E}';\mathcal{E}_\mathsf{f} \vdash e_i \rhd v_i\end{array}}{\begin{array}{l}\texttt{match}\ x\ \texttt{with} \\ \mid pat_1 \rightarrow e_1 \\ \qquad\vdots \\ \mid pat_n \rightarrow e_n \\ [\mid \_ \rightarrow e]\end{array}\ \mathcal{E};\mathcal{E}_\mathsf{f} \vdash \rhd v_i}\ \text{MATCH} \qquad \frac{\begin{array}{c}\mathcal{E}(x) = v \qquad \forall j,\ \ \mathcal{F}(pat_j, v) = \texttt{nok} \\ \mathcal{E};\mathcal{E}_\mathsf{f} \vdash e \rhd v\end{array}}{\begin{array}{l}\texttt{match}\ x\ \texttt{with} \\ \mid pat_1 \rightarrow e_1 \\ \qquad\vdots \\ \mid pat_n \rightarrow e_n \\ \mid \_ \rightarrow e\end{array}\ \mathcal{E};\mathcal{E}_\mathsf{f} \vdash \rhd v}\ \text{CATCH}$$

Fig. 4. FMON expression semantics

which are variables that may be bound to terms built over constructors, equalities and inequalities. In addition, the language contains three constraints introduced to stick to the behavior of functional programs: a constraint capturing function calls, a constraint called `ite` capturing conditionals and a constraint called `match` for pattern-matching. Note that, within a constraint logic programming environment (such as SICStus Prolog for example), the function-call constraint is somehow built-in in the language because it simply corresponds to clause invocation. The last two constraints are basically user-defined constraints. Informally, the constraint $\texttt{ite}(X, \sigma, \sigma')$ where X is an algebraic variable, is satisfied when the constraints of σ (resp. σ') are satisfied if X is valuated with $Ctrue$ (resp. $Cfalse$), $Ctrue$ and $Cfalse$ being two special constants. A similar informal semantics is given to the `match` constraint (this time the discriminant part is the constructor used in the value of X). In order to distinguish FMON variables from constraint variables in the rest of the paper, we write FMON variables with lowercases $x, y, ...$ and both finite domain and algebraic variables with uppercases $X, Y, ...$ However, for the sake of clarity, we use the same spelling for constructors and predefined operators in both contexts, although they are theoretically distinct.

σ	$::= c \mid \sigma, \sigma$		simple constraint/store
c	$::= X =_{fd} a \mid X \neq_{fd} a$		integer equality/inequality
	$\mid \quad X =_h t \mid X \neq_h t$		algebraic equality/inequality
	$\mid \quad \texttt{f}(X_1, \ldots, X_n)$		function call
	$\mid \quad \texttt{match}(X, [\ \texttt{patt}(pat, \sigma),$		
	$\qquad\qquad \ldots,$		matching constraint
	$\qquad\qquad \texttt{patt}(pat, \sigma)], \sigma)$		
	$\mid \quad \texttt{ite}(X, \sigma, \sigma)$		conditional constraint
	$\mid \quad \texttt{fail}$		fail constraint
pat	$::= C(X_1, \ldots, X_n)$		pattern
a	$::= i \mid X \mid op(X_1, X_2)$		integer/finite domain variable/bin. int. operator
t	$::= X \mid C(X_1, \ldots, X_n)$		algebraic variable/term

Fig. 5. Syntax of constraints

Cclosure Environment. The counterpart of function/closure in the constraint system is the notion of *Cclosure* (Constraint closure) that we introduce here. A Cclosure $<X_1, \ldots, X_n, R \rightsquigarrow \sigma>$ is a triple composed of a variable R, a list of constraint variables X_1, ... X_n and a set of constraints $\sigma = \{c_1, ..., c_k\}$. At a first glance, a Cclosure can be seen as a clause definition for a $n+1$-ary predicate $f(X_1, ..., X_n, R)$:- $c_1, ..., c_k$. Note that any FMON function closure is translated into a Cclosure.

Semantics. An assignment (that is a function that associates values to constraint variables) is *consistent* for a given store σ if all the constraints of σ are satisfied by the assignment. The assignment is *total* if all the variables appearing in σ are valued. A *solution* of σ is both a total and consistent assignment.

Defining the semantics of the constraint language means to explicit the cases where an assignment satisfies σ. The hardest part concerns function calls because they require to unfold the constraints σ_b associated to the function body and these constraints may contain new variables, external to the assignment. And these variables also need to be valuated to satisfy the constraints of σ_b. For an assignment valuating only the arguments of a given function call and from the caller point of view (i.e., before any function unfolding), the assignment is total since it is defined for the function arguments. However, from the callee point of view, the assignment is not total because new variables need to be valuated. Intuitively, to tackle this problem it suffices to define a total assignment as *an assignment containing the values for all variables involved in the resolution of the constraint system* or to set up the value of any variable appearing in the unfolded system. The second proposition is however insufficient to allow recursive function calls, because in this case the set of variables may become unbounded. For the first proposition, deciding whether a variable is *involved* depends on

the value of other variables. It explains, at least partially, why we chose to define the semantics as a predicate able to complete a current assignment. Thus, the *satisfaction relation* (also called the *solution predicate*) has the following judgement $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \sigma \mapsto \mathcal{A}'$ which can be read: *the assignment $\mathcal{A}$ is extended into the assignment $\mathcal{A}'$, solution of σ w.r.t. the Cclosure environment $\mathcal{E}_{\mathtt{cl}}$*. A formal definition is given in Fig. 6 and Fig.. 7.

$$\frac{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash c \mapsto \mathcal{A}_1 \quad \mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash \sigma \mapsto \mathcal{A}_2}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash c, \sigma \mapsto \mathcal{A}_2}\ \text{CJ1} \qquad \frac{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \sigma_1 \mapsto \mathcal{A}_1 \quad \mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash \sigma_2 \mapsto \mathcal{A}_2}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \sigma_1, \sigma_2 \mapsto \mathcal{A}_2}\ \text{CJ2}$$

$$\frac{\mathcal{A}(X) = i}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash X =_{fd} i \mapsto \mathcal{A}}\ \text{FDI} \qquad \frac{X \notin \mathcal{A}}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash X =_{fd} i \mapsto \mathcal{A} \oplus (Xi)}\ \text{FDI'}$$

$$\frac{\mathcal{A}(X) = [\![op, \mathcal{A}(X_1), \mathcal{A}(X_2)]\!]}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash X =_{fd} op(X_1, X_2) \mapsto \mathcal{A}}\ \text{OP} \qquad \frac{X \notin \mathcal{A} \quad v = [\![op, \mathcal{A}(X_1), \mathcal{A}(X_2)]\!]}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash X =_{fd} op(X_1, X_2) \mapsto \mathcal{A} \oplus (X, v)}\ \text{OP'}$$

$$\frac{\mathcal{A}(X) = C(\mathcal{A}(X_1), \ldots, \mathcal{A}(X_n))}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash X =_h C(X_1, \ldots, X_n) \mapsto \mathcal{A}}\ \text{C} \qquad \frac{X \notin \mathcal{A} \quad v = C(\mathcal{A}(X_1), \ldots, \mathcal{A}(X_n))}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash X =_h C(X_1, \ldots, X_n) \mapsto \mathcal{A} \oplus (X, v)}\ \text{C'}$$

$$\frac{\mathcal{A}(X) = \mathcal{A}(Y) \qquad \diamond \in \{h, fd\}}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash X =_\diamond Y \mapsto \mathcal{A}}\ \text{X} \qquad \frac{X \notin \mathcal{A} \quad Y \in \mathcal{A} \quad \diamond \in \{h, fd\}}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash X =_\diamond Y \mapsto \mathcal{A} \oplus (X, \mathcal{A}(Y))}\ \text{X'}$$

Fig. 6. Solution predicate for a store of constraints (part1)

The following Cclosure environment illustrates the concepts presented above by defining the store associated to the factorial function:

$$\mathcal{E}_{\mathtt{cl}} = (\mathsf{fact}, <N_1, R_1 \rightsquigarrow \sigma>)$$
$$\sigma = (C =_h (N_1 \leq 1), \mathtt{ite}(C, [R_1 =_{fd} 1], [N_2 =_{fd} N_1 - 1, \mathsf{fact}(N_2, R_2), R_1 =_{fd} R_2 * N_1]))$$

The sequent $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \mathsf{fact}(E, S) \mapsto \mathcal{A}$ (where $\mathcal{A} = (S, 6), (E, 3)$) can be established. Each time the definition of fact is unfolded, any variable from fact can get a value in the assignment. These values are then erased after the verification of the constraints attached to fact (see rule CALL).

The solution predicate specifies that $\mathcal{A}$ has to satisfy all the constraints of the store (see rules CJ1 and CJ2). Equality constraints are satisfied when both sides have the same value (rules FDI, X, OP, C). If a variable is not valued in the assignment, then extending the assignment according to the constraint is possible. In every case, the variable is fully defined by the sole right-hand side of the constraint (rules FDI', X', OP', HC'). The other rules associated to our additional constraints follow a simple operational semantics related to their programming counterpart. Evaluating a call $\mathtt{f}(X_1, \ldots, X_n, R)$ leads to satisfy the constraints in the closure of f, provided that the variables of f are replaced by

$$\frac{\mathcal{E}_{\mathtt{cl}}(\mathtt{f}) = <X'_1, \ldots, X'_n, R' \rightsquigarrow \sigma_b> \quad (X'_1, \mathcal{A}(X_1)), \ldots, (X'_n, \mathcal{A}(X_n), (R', \mathcal{A}(R)); \mathcal{E}_{\mathtt{cl}} \vdash \sigma_b \mapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \mathtt{f}(X_1, \ldots, X_n, R) \mapsto \mathcal{A}} \text{ CALL}$$

$$\frac{\mathcal{A}(X) = Ctrue \qquad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \sigma_1 \mapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \mathtt{ite}(X, \sigma_1, \sigma_2) \mapsto \mathcal{A}'} \text{ ITE\_TRUE}$$

$$\frac{\mathcal{A}(X) = Cfalse \qquad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \sigma_2 \mapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \mathtt{ite}(X, \sigma_1, \sigma_2) \mapsto \mathcal{A}'} \text{ ITE\_FALSE}$$

$$\frac{\mathcal{A}(X) = C_k(\mathcal{A}(X_1^k), \ldots, \mathcal{A}(X_{n_k}^k)) \qquad 1 \leq k \leq i \quad \mathcal{A} \oplus (X_1^k, \mathcal{A}(X_1^k)), \ldots, (X_{n_k}^k, \mathcal{A}(X_{n_k}^k)); \mathcal{E}_{\mathtt{cl}} \vdash \sigma_k \mapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \begin{array}{l}\mathtt{match}(X, [\\ \quad \mathtt{patt}(X =_h C_1(X_1^1, \ldots, X_{n_1}^1), \sigma_1),\\ \quad \vdots \\ \quad \mathtt{patt}(X =_h C_i(X_1^i, \ldots, X_{n_i}^i), \sigma_i)\ ],\\ \sigma)\end{array} \mapsto \mathcal{A}'} \text{ MATCH\_PAT}$$

$$\frac{\sigma \neq \mathtt{fail} \qquad \mathcal{A}(X) \neq C_k(\mathcal{A}(X_1^k), \ldots, \mathcal{A}(X_{n_k}^k)) \qquad \forall k, 1 \leq k \leq i \quad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \sigma \mapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \begin{array}{l}\mathtt{match}(X, [\\ \quad \mathtt{patt}(X =_h C_1(X_1^1, \ldots, X_{n_1}^1), \sigma_1),\\ \quad \vdots \\ \quad \mathtt{patt}(X =_h C_i(X_1^i, \ldots, X_{n_i}^i), \sigma_i)\ ],\\ \sigma)\end{array} \mapsto \mathcal{A}'} \text{ MATCH\_OTHER}$$

Fig. 7. Solution predicate for a store of constraints (part 2)

the values assigned to $X_1, \ldots, X_n$ and R in the assignment $\mathcal{A}$ (see the rule CALL). Rules MATCH_PAT and MATCH_OTHER are used to formalize a `match` constraint, which is satisfied if at most one of the patterns is matched. Recall that patterns are expected to be non-overlapping.

We now turn on the statement of two main theorems on solution predicates, that will be useful in the next section. The first one is proved by induction over $\mathcal{A} \vdash \sigma \mapsto \mathcal{A}'$, the second one is proved by induction on the store σ.

Theorem 1. *Let $\mathcal{A}$, $\mathcal{A}'$ be two assignments and σ be a constraint store. If $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \sigma \mapsto \mathcal{A}'$ then $\mathcal{A}'$ is a solution for σ.*

Theorem 2. *Let $\mathcal{A}$, $\mathcal{A}'$ be two assignments and σ be a constraint store such that $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash \sigma \mapsto \mathcal{A}'$.*
Then, for any variable X in the domain of $\mathcal{A}'$, $\mathcal{A} \oplus (X, \mathcal{A}'(X)); \mathcal{E}_{\mathtt{cl}} \vdash \sigma \mapsto \mathcal{A}'$ is valid.

4.3 Translating FMON Expressions into Constraints

In this section, we explain how to translate any FMON expression e into a set of contraints σ using syntax-directed rules (see Fig. 8). Such rules involve a judgement $\mathcal{T}_x, R \vdash_C e \longmapsto \sigma$ where $\mathcal{T}_x$ is an environment associating a FMON variable to its corresponding constraint variable, and R is a constraint variable associated to the result of the expression. An implicit hypothesis behind this translation states that the test unity is composed of well-typed functions and properties. As a result, choosing between numerical equality or algebraic equality is performed using the typing information only from rules VALUE and VAR. In rule VALUE, $\mathcal{T}_v$ is the identity function except for FMON Boolean values that are translated into $Ctrue$ or $Cfalse$, 2 algebraic constants. Translating a variable is immediate, just look at the translation environment $\mathcal{T}_x$. The translation of a conditional (rule IF) is also straightforward: translate the condition which is a variable, translate both branches separately with the same result variable R. As we said previously, a FMON n-ary function call is translated into a $n+1$-ary constraint, close to a Prolog predicate by adding a constraint variable for the result (see rule FUNCTION).

Rule LET requires to produce a fresh variable associated with the value of the expression e_1. Thus, predicate $\mathcal{F}reshC(X)$ denotes that X is a fresh variable. Rule MATCH (resp. MATCHCATCH) translates any pattern-matching expression without (resp. with) a final *catch-all* clause. Both rules require recursive calls to translate the expressions e_i and the patterns pat_i. The only difference between both rules lies on the default clause: `fail` is generated if there is none. Function $\mathcal{T}_p$ syntactically translates a pattern into a term where FMON variables are translated into constraint variables. The function also returns a translation variable environment associating pattern variables with fresh variables.

Each function definition appearing in $\mathcal{E}_\mathsf{f}$ is translated into a Cclosure. More precisely, if $\mathcal{E}_\mathsf{f}(\mathsf{f}) = <x_1, \ldots, x_n \rightsquigarrow e>$ then the Cclosure obtained by translation is $<X_1, \ldots, X_n, R \rightsquigarrow \sigma>$ (R, X_1, ..., X_n are fresh variables) where σ is the translation of e when R is bound to the result of e or formally,
$(x_1, X_1), \ldots, (x_n, X_n); R \vdash_C e \longmapsto \sigma$.
For example the Cclosure for function `app` in Fig. 1 is

$$<L, G, R \rightsquigarrow \begin{array}{l} \texttt{match}(L, \\ \quad [\texttt{patt}(L =_h nil, R =_h G) \\ \quad \texttt{patt}(L =_h cons(H, T), R =_h cons(H, K), app(T, G, K))>. \\ \quad], \texttt{fail}) \end{array}$$

To end up this presentation, we now explain how to translate the precondition part of a property into a set of constraints. As a fresh variable is associated to each condition, the process just adds an equality constraint to either $true$ or $false$ depending on the expectations. As a simple example, if we want to satisfy the precondition of the property of Fig. 1 with the Cclosure environment containing the Cclosures for `app`, `rev_aux` and `rev`, then the precondition is translated into $L =_h R, \texttt{app}(L_1, L_2, R)$. On the contrary, if we want to falsify it, then the precondition is translated into: $L \neq_h R, \texttt{app}(L_1, L_2, R)$.

$$\frac{\mathcal{T}_x(x_1) = X_1 \qquad \ldots \qquad \mathcal{T}_x(x_n) = X_n}{\mathcal{T}_x; R \vdash_C \mathtt{f}(x_1, \ldots, x_n) \longmapsto \mathtt{f}(R, X_1, \ldots, X_n)} \text{ FUNCTION}$$

$$\frac{\mathcal{F}reshC(X) \qquad \mathcal{T}_x; X \vdash_C e_1 \longmapsto \sigma_1 \qquad \mathcal{T}_x \oplus (xX); R \vdash_C e_2 \longmapsto \sigma_2}{\mathcal{T}_x; R \vdash_C \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \longmapsto \sigma_1 \wedge \sigma_2} \text{ LET}$$

$$\frac{\mathcal{T}_v(v) = v'}{\mathcal{T}_x; R \vdash_C v \longmapsto R =_\diamond v'} \text{ VALUE} \qquad \frac{\mathcal{T}_x(x) = X}{\mathcal{T}_x; R \vdash_C x \longmapsto R =_\diamond X} \text{ VAR}$$

$\diamond \in \{fd, h\}$ wrt the type of the value/variable

$$\frac{\mathcal{T}_x(x) = X \qquad \mathcal{T}_x; R \vdash_C e_1 \longmapsto \sigma_1 \qquad \mathcal{T}_x; R \vdash_C e_2 \longmapsto \sigma_2}{\mathcal{T}_x; R \vdash_C \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \longmapsto \mathtt{ite}(X, \sigma_1, \sigma_2)} \text{ IF}$$

$$\frac{\mathcal{T}_x(x) = X \qquad \mathcal{T}_p(pat_i) = (Cpat_i, \mathcal{T}_i) \qquad \mathcal{T}_x \oplus \mathcal{T}_i; R \vdash_C e_i \longmapsto \sigma_i \qquad \forall i \in [\![1, n]\!]}{\mathcal{T}_x; R \vdash_C \begin{array}{l|l} \mathtt{match}\ x\ \mathtt{with} & \mathtt{match}(X, [\\ \mid pat_1 \to e_1 & \quad \mathtt{patt}(X =_h Cpat_1, \sigma_1) \\ \vdots \quad \longmapsto & \quad \vdots \\ \mid pat_n \to e_n & \quad \mathtt{patt}(X =_h Cpat_n, \sigma_n)], \mathtt{fail}) \end{array}} \text{ MATCH}$$

$$\frac{\mathcal{T}_x(x) = X \qquad \mathcal{T}_p(pat_i) = (Cpat_i, \mathcal{T}_i) \qquad \mathcal{T}_x \oplus \mathcal{T}_i; R \vdash_C e_i \longmapsto \sigma_i \qquad \forall i \in [\![1, n+1]\!]}{\mathcal{T}_x; R \vdash_C \begin{array}{l|l} \mathtt{match}\ x\ \mathtt{with} & \mathtt{match}(X, [\\ \mid pat_1 \to e_1 & \quad \mathtt{patt}(X =_h Cpat_1, \sigma_1) \\ \vdots \quad \longmapsto & \quad \vdots \\ \mid pat_n \to e_n & \quad \mathtt{patt}(X =_h Cpat_n, \sigma_n)\ ], \\ \mid \_ \to e_{n+1} & \quad \sigma_{n+1}) \end{array}} \text{ MATCHCATCH}$$

Fig. 8. Translation of FMON expressions into constraints

5 Soundness and Completeness of the Constraint Generation

In this section, we formalize both theorems and discuss shortly the difficulties of the proofs.

Theorem 3 (Soundness). *if* $\mathcal{E}; \mathcal{E}_f \vdash e \rhd v$ *and* $\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$ *and* $\mathcal{A} \models_{\mathcal{T}_x} \mathcal{E}$ *then there exists* $\mathcal{A}'$ *such that* $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash R = v, \sigma \mapsto \mathcal{A}'$.

This theorem establishes the correction property we want to demonstrate, namely to show that the evaluation result of an expression is necessarily a solution of the constraint system. Formally speaking, if an expression e evaluates to value v w.r.t. the evaluation context $\mathcal{E}, \mathcal{E}_\mathsf{f}$, and if e is translated into a constraint store σ where R is the variable that captures the evaluation result v, then there exists an extension $\mathcal{A}'$ of $\mathcal{A}$ where $R = v$. The notation $\mathcal{A} \models_{\mathcal{T}_x} \mathcal{E}$ means that $\mathcal{A}$ and $\mathcal{E}$ share the same domain and agree on the values of a variable and its constraint counterpart.

This semantics-preserving theorem shows that the FMON semantics is really correctly captured by the semantics of the constraint system. Its proof is done by induction over the evaluation of the expression e (i.e. $\mathcal{E}; \mathcal{E}_\mathsf{f} \vdash e \rhd v$) The hardest part of the proof is concerned with the `let` binding and `match` cases where freshness of variable has to be dealt with.

Theorem 4 (Completeness). *If* $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash R = v, \sigma \mapsto \mathcal{A}'$ *and* $\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$ *and* $\mathcal{A} \models_{\mathcal{T}_x} \mathcal{E}$ *then* $\mathcal{E}; \mathcal{E}_f \vdash e \rhd v$.

This theorem states the completeness of the translation by showing that any solution of the constraint system is also a possible result of the evaluation of the corresponding expression. Formally speaking, if expression e is translated into σ where R is the returned value and if $\mathcal{A}$ is one of the solutions of $R = v, \sigma$ then the context $\mathcal{E}$ leads to the evaluation of e into v. Note that, with this theorem, the set of possible evaluations of e is an over-approximation of the set of solutions of the constraint system. This theorem is proved by induction over the solution predicates; the hardest point being related to rule CJ2 that combines two distinct constraint stores to translate the `let` binding expression.

Both soundness and completeness theorems prove there exists a bijection between the set of functional program evaluations and the solution set of the constraint system. Provided we have at hand a correct constraint solver, it means that a solution of the constraint system is actually a test data solving the testing objective over the functional program.

Paper-written proofs of these theorems are available (in Carlier's Ph.D [3] and [5]) but more interestingly, we also performed a machine-checked proof of the soundness theorem in Coq. This allowed us to find a bug in one of the rules in Fig. 7 and to patch the soundness proof. The Coq implementation available at `www.ensiie.fr/~dubois/Coqfocaltest` follows carefully the specifications given in this paper and also provides a lot of extra details, in particular about fresh names and dedicated data structures. Minimal extra type information has been necessary, it appears as annotations tied to variables, more precisely a Boolean set to true if the variable has type integer, false if it has a concrete type. The Coq development contains around 20 000 lines of code. Thanks to the Coq extraction mechanism a correct translator written in OCaml can be extracted and is operational.

6 Conclusion

We have formally and semantically defined and verified the translation of test unities into constraints. A test unity is composed of both a program and a test objective for which a test data generation is issued. In this paper, provided that we have at hand a correct and complete constraint solver, we proved both manually and with the proof-assistant Coq, that such an issue can be automatically solved in our constraint-based testing tool FocalTest. In other words we have demonstrated the soundness and completeness of the translation process, ensuring so an equivalence between the solutions of the constraint system and the evaluation of the corresponding Focalize program.

However, this work still needs to be extended in two directions. Firstly, we did not deal with higher-order function calls in our proofs, although these aspects are tackled in FocalTest. Secondly, to complete the overall picture, we need to formalize and verify (with machine-checked proofs) the algorithms involved in the constraint solving procedure. This work is currently under process and we are pretty confident in our ability to design a complete correction proof of our constraint-based test data generation tool. This would make a first step towards more formally verified test data generation tools.

References

1. Ayrault, P., Carlier, M., Delahaye, D., Dubois, C., Doligez, D., Habib, L., Hardin, T., Jaume, M., Morisset, C., Pessaux, F., Rioboo, R., Weis, P.: Trusted software within focal. In: C&ESAR 2008, Computer Electronics Security Applications Rendez-vous, pp. 162–179 (2008)
2. Brucker, A.D., Wolff, B.: Interactive Testing with HOL-TestGen. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 87–102. Springer, Heidelberg (2006)
3. Carlier, M.: Test automatique de propriétés dans un atelier de développement de logiciels sûrs. PhD thesis, CEDRIC Laboratory, Paris, France (2009)
4. Carlier, M., Dubois, C.: Functional Testing in the Focal Environment. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 84–98. Springer, Heidelberg (2008)
5. Carlier, M., Dubois, C., Gotlieb, A.: Constraint reasoning in focaltest. In: Int. Conf. on Soft. and Data Tech. (ICSOFT 2010), Athens (July 2010); Also, CNAM Tech. Report CEDRIC-09-1703, 36 pages (2009)
6. Charreteur, F., Gotlieb, A.: Constraint-based test input generation for java bytecode. In: 21st IEEE Int. Symp. on Softw. Reliability Eng. (ISSRE 2010), San Jose, CA, USA (November 2010)
7. Denmat, T., Gotlieb, A., Ducasse, M.: Improving constraint-based testing with dynamic linear relaxations. In: 18th IEEE Int. Symp. on Soft. Reliability Eng. (ISSRE 2007), Trollhttan, Sweden (November 2007)
8. Coq development team. The Coq proof assistant reference manual, Ver. 8.3 (2009)
9. Fink, G., Bishop, M.: Property-based testing: A new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes 22(4), 74–80 (1997)
10. Flanagan, C.: Automatic software model checking via constraint logic. Sci. Comput. Program. 50(1-3), 253–270 (2004)

11. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Test case generation for object-oriented imperative languages in clp. TPLP 10(4-6), 659–674 (2010)
12. Gotlieb, A.: Euclide: A constraint-based testing platform for critical c programs. In: Int. Conf. on Soft. Testing, Valid. and Verif. (ICST 2009), Denver (April 2009)
13. Gotlieb, A., Botella, B., Rueher, M.: A CLP Framework for Computing Structural Test Data. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 399–413. Springer, Heidelberg (2000)
14. Hayhurst, K., Veerhusen, S., Chilenski, J., Rierson, L.K.: A practical tutorial on modified condition/decision coverage, nasa langley. Technical report (2001)
15. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
16. Maranget, L.: Compiling Lazy Pattern Matching. In: Conference on Lisp and Functional Programming. ACM Press (1992)
17. Marre, B., Arnould, A.: Test sequences generation from lustre descriptions: Gatel. In: Proc. of the 15th IEEE Conference on Automated Software Engineering (ASE 2000). IEEE CS Press (September 2000)
18. Podelski, A.: Model Checking as Constraint Solving. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 22–37. Springer, Heidelberg (2000)
19. Wotawa, F., Nica, M.: On the compilation of programs into their equivalent constraint representation. Informatica 32(4), 359–371 (2008)

Testing Library Specifications by Verifying Conformance Tests

Joseph R. Kiniry[1], Daniel M. Zimmerman[2], and Ralph Hyland[3]

[1] IT University of Copenhagen, Denmark
kiniry@acm.org
[2] University of Washington Tacoma, USA
dmz@acm.org
[3] University College Dublin, Ireland
ralph.hyland@gmail.com

Abstract. Formal specifications of standard libraries are necessary when statically verifying software that uses those libraries. Library specifications must be both *correct*, accurately reflecting library behavior, and *useful*, describing library behavior in sufficient detail to allow static verification of client programs. Specification and verification researchers regularly face the question of whether the library specifications we use are correct and useful, and we have collectively provided no good answers. Over the past few years we have created and refined a software engineering process, which we call the *Formal CTD Process* (FCTD), to address this problem. Although FCTD is primarily targeted toward those who write Java libraries (or specifications for existing Java libraries) using the Java Modeling Language (JML), its techniques are broadly applicable. The key to FCTD is its novel usage of library conformance test suites. Rather than executing the conformance tests, FCTD uses them to measure the *correctness* and *utility* of specifications through static verification. FCTD is beginning to see significant use within the JML community and is the cornerstone process of the JML Spec-a-thons, meetings that bring JML researchers and practitioners together for intensive specification writing sessions. This article describes the Formal CTD Process, its use in small case studies, and its broad application to the standard Java class library.

1 Introduction

In an ideal world, all software systems would be 100% reliable and have verifiable formal specifications. These specifications would be both *correct*, accurately reflecting the runtime behavior of the implementations they claim to describe, and *useful*, providing sufficient information to allow developers to use and extend the implementations in safe, behaviorally predictable ways.

Clearly, we do not—and will likely never—live in this ideal world. The vast majority of today's software has defects, and some is completely unreliable. Of the software that is considered robust and reliable, most has not undergone formal verification. A commonly discussed way in which the reliability of such software is determined is through exhaustive automated unit testing, both of

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 51–66, 2012.
© Springer-Verlag Berlin Heidelberg 2012

individual components and of full systems. Very little of today's software has formal specifications of any kind and, in our experience, many of the formal specifications that do exist are either incorrect or not useful.

While it is clearly impossible to achieve an ideal, 100% reliable, fully-specified world, it is certainly possible to improve the current state of software specification and reliability. One approach to doing this for Java applications, taken by the Java Modeling Language (JML) community, has been to retroactively provide formal specifications for the behavior of the standard Java class library. Since the standard library is the foundation for all Java software, this enables Java programs to be formally specified and verified using the standard library specifications as building blocks. A set of specifications for much of the Java 1.4 class library was included as part of the release of the Common JML tool suite [2], and has since been used to formally specify and verify multiple large Java systems [13,14]. In addition, a set of specifications for Java 1.5 through 1.7 is under development as part of the OpenJML project.

We call such retroactive specification of classes *Contract the Design* (CTD). To perform CTD, the specification writer takes an implementation that already exists, such as the Java class library, and writes *contracts* (formal specifications) to describe the existing behavior. This approach is effectively the logical dual of *Design by Contract* (DBC) [17], where contracts are written first and the software is subsequently implemented in a way that fulfills the contracts.

Unfortunately, two critical difficulties arise when using CTD to specify existing systems. The first is that retroactively devising good specifications for existing systems is quite difficult, even with full access to source code, and is especially problematic when documentation is incomplete, absent, or vague (a not infrequent occurrence). The second is that, once a specification has been written for a non-trivial piece of software, it is hard to determine whether the specification is correct and useful. Imagining all the situations in which the software might be used is infeasible, and it is impossible to just "try out" the specification with the multitude of tools that will consume and use it. These difficulties are also faced by developers writing conformance tests for libraries (and, more generally, unit tests for nontrivial software systems), who must devise sets of tests that thoroughly exercise the functionality of their systems.

As a result of these difficulties, the Java class library specifications currently packaged with JML were written over several years in essentially ad hoc fashion. Efforts were made to ensure their correctness, and they are all (to the best of our knowledge) at least type-correct; however, there was no systematic way to measure their utility and, until now, no attempt to devise such. Deficiencies in the specifications have primarily been discovered by application developers attempting to verify their code and by tool developers attempting to make their tools understand the specifications, and the specifications have been patched in various ways over time in response to these discoveries.

This work addresses the difficulties in using CTD for library specification, taking advantage of the substantial body of knowledge related to the creation of high quality unit test suites by combining existing testing techniques with

runtime and static verification techniques. The resulting process, which we call the *Formal CTD Process* (FCTD), allows us to effectively test specifications for correctness and measure their utility.

The *Formal* in *Formal CTD Process* refers to the fact that we use verification tools to determine the *correctness* and *utility* of specifications. We consider the correctness of a specification to be a binary property: a specification is correct if it is never violated by the implementation it claims to specify and incorrect otherwise. This can be determined statically or, in cases where the source code of the system being specified is unavailable, evidence for such can be observed dynamically. We measure the utility of a specification as the percentage of the unit tests for the specified implementation that can be statically verified using the specification.

The "secret sauce" of this process is the realization that unit tests and verification "fit" together well. Unit tests *operationally* express the correct behavior of a system under test (SUT), while specifications *denotationally* express the correct behavior of a SUT. Thus, unit test outcomes *should* be statically verifiable using SUT specifications. Just as *test coverage* criteria tell us something about the quality of our *implementations*, *verification coverage* criteria (correctness and utility) tell us something about the quality of our *specifications*.

FCTD has been used in several case studies, and we have devised a method to apply it to the entire standard Java class library. The resulting ability to test the specifications of the Java class library is playing a critical role in the effort to update JML to support current and future versions of Java. The end result of this effort will be a correct and useful JML specification of the Java class library to complement the (often poor) existing Javadoc "specification", which will both clarify the behavior of the library for application developers and allow them to confidently use the many verification and validation tools that understand JML. In the remainder of this article we provide background information about the components of the FCTD process, describe the principles underlying FCTD, and discuss concrete realizations of FCTD for JML and Java code.

2 Background

FCTD combines unit testing and verification to evaluate specifications. In this section, we provide background information on the tools and techniques used in our concrete realizations of FCTD: JML, unit testing, and static checking. More detailed descriptions are available in the cited works and in the documentation accompanying the tools.

2.1 The Java Modeling Language

The Java Modeling Language [16] is a specification language for Java programs. It allows developers to specify class and method contracts (i.e., preconditions, postconditions, invariants) as well as more sophisticated properties, up to and including mathematical models of program behavior. Many tools are compatible

with JML, including compilers, static checkers, runtime assertion checkers, unit test generators, and automated specification generators [2].

Runtime assertion checking (RAC) is one of the most common applications of JML. A special RAC compiler is used to transform JML-annotated Java code, adding runtime checks for method preconditions and postconditions, class invariants and constraints, and other JML contracts such as loop invariants and variant functions. The transformed code, when run in any Java virtual machine, signals assertion failures at runtime via special JML runtime errors.

JML and its associated tools are used in many contexts. Many formal methods researchers have adopted JML, and many JML-compatible tools have full JML specifications; in some cases, these tools have even been statically verified. JML is also used in educational contexts at several universities [15], as well as in production systems such as the KOA Internet-based remote voting system [13].

Currently, most stable and supported publicly available JML tools only support Java versions prior to 5.0. The OpenJML project [5] and the JMLEclipse project [3] are efforts to build JML tools atop modern compiler code bases (OpenJDK [19] and the Eclipse JDT [22], respectively), so that JML will be able to better keep pace with future changes to Java. This modernization effort is a primary motivator for the work presented here, as a multitude of new specifications must be written for new and changed classes in the standard Java class library before a new set of JML tools can be effectively used.

2.2 Unit Testing

Unit testing has long been an important validation technique in many software development processes. It is essentially the execution of individual components of a system (the *units*) in specific contexts to determine whether the components generate expected results. A single *unit test* has two main parts: *test data* and a *test oracle*. The *test data* are input values—for example, method parameter values—used to establish the state of the unit under test. The *test oracle* is a (typically small) program that determines whether the behavior of the unit is correct when it is executed with the test data.

Testing a non-trivial software system typically requires many unit tests, which are collectively called a *test suite*. The quality, usually called *coverage*, of a test suite is measured in several ways [23]; for example, given a particular SUT, *statement coverage* (sometimes called *code coverage*) is the percentage of the executable code in that system that is executed during testing, while *path coverage* is the percentage of the possible execution paths through that system that is executed during testing.

As will become evident in Section 3, effective use of FCTD requires the availability of a high-coverage test suite for the system being specified, preferably a *conformance test* suite that defines the full functionality of the system. Significant research has been done on techniques for automatically generating test suites, and many of those techniques are quite promising. However, the development of test suites is still predominantly done without automation. Developers sit down with the system to be tested, decide what test data should be used

and how to determine whether each test has passed or failed, and encode this information manually.

Test suites are usually executed within a test framework that runs and records the result of each test and summarizes the results of the suite to the developer. This allows the test suite to be run easily, repeatably, and without constant developer supervision; it also allows test runs to be incorporated into the automatic build processes that exist in many software development environments. Such frameworks exist for nearly every programming language; the predominant ones for Java are JUnit [10] and TestNG [1]. For the purposes of FCTD there is no clear reason to choose one over the other; we have typically used JUnit because it ships as an integrated part of the open-source Eclipse Development Platform [22], which we use for most of our development work.

One particular way of automating the development and execution of test suites that applies specifically to JML-annotated Java programs is embodied in the JMLUnit [4] and JMLUnitNG [24] tools. These tools automatically generate test oracles using JML runtime assertion checks. For each method under test, the tools construct JUnit or TestNG (respectively) tests to call that method with multiple test data values taken from a default or developer-provided set. Each test passes if the method call completes with no assertion failures and fails if the method call completes with an assertion failure other than a violation of the method's precondition. If a test violates the method's precondition it is considered *meaningless*, because the behavior of a method is undefined when its precondition is violated and can therefore not be evaluated.

FCTD does not work with unit tests generated by tools like JMLUnit and JMLUnitNG, because such tools generate operational "specifications" (in the form of unit tests) *directly from* denotational specifications (in the form of JML). Since the resulting tests pass exactly when the specifications are satisfied, regardless of how trivial the specifications are, they can provide no information about specification utility. To guarantee "objective" evaluation of specification utility, unit tests must be written *independently* (i.e., without reference to the specifications being evaluated).

2.3 Static Verification

Static verification is a process whereby a body of formally specified source or object code is analyzed to determine whether it satisfies its specification. This analysis is typically carried out by transforming the code and specification into verification conditions, which are then evaluated using one or more automated theorem provers. An *extended static checker* is a tool that performs static verification as well as checking for and flagging common programming errors.

In our initial experiments with FCTD we have primarily used ESC/Java2 [14], an evolution of the original Digital SRC ESC/Java [8], to perform static verification on JML-annotated Java code. We have also experimented with the prototype ESCs being developed as part of the OpenJML and JMLEclipse projects, but these are not currently robust enough to reason about the rich specifications under discussion here.

ESC/Java2 detects typical Java programming errors such as null pointer dereferences, invalid class casts, and out-of-bounds array indexing. It also performs several kinds of automated verification to attempt to ensure that the code is correct with respect to its associated JML specifications and, in conjunction with a specification consistency checker, to ensure that the specifications themselves are sound. The consistency check is important, especially for specification writers who are just learning CTD/DBC, because it is easy for inexperienced developers to write inconsistent specifications (e.g., invariants that collapse to `true`) that are always satisfied regardless of the system's actual behavior.

The verification performed by ESC/Java2 is *modular*; it verifies each method m by transforming the Java code of only method m into verification conditions and relies on the JML specifications of all the other methods and classes to which m refers to create the remainder of the verification conditions it needs. Thus, ESC/Java2 verifies each method in relative isolation, which is far less resource-intensive (and far more feasible given the state of automated theorem prover technology) than processing an entire system, or even an entire class, at once.

Support for modern Java syntax and constructs in ESC, as in JML itself, is still a work in progress. We expect that the new ESC tools being developed as part of the OpenJML and JMLEclipse projects will address this issue.

3 The Formal CTD Process

The Formal CTD Process is a combination of unit testing techniques and verification techniques. FCTD is general enough to be applied to systems and specifications written in any language and unit tests running in any framework; the only requirement is the availability of at least one static verification tool capable of reasoning about off-the-shelf unit tests.

It is important to note that executing unit tests is a *completely optional* part of the process. Executing unit tests within a RAC environment to determine whether CTD specifications are violated by the tests is certainly feasible; however, there are two serious issues with using runtime checks to evaluate specification quality. First, it is quite easy to write correct, but clearly non-useful, specifications that pass their runtime assertion checks regardless of what the underlying implementation does. Second, it is possible for all the runtime checks to pass when running a test suite that does not thoroughly exercise the implementation, leaving incorrect specifications undiscovered.

A conformance test suite (or other high-coverage test suite) can provide some persuasive evidence for the *correctness* of a set of specifications through runtime checking, because it thoroughly exercises all the intended functionality of the implementation. However, even runtime checks of conformance tests cannot *conclusively* establish correctness because there is always the possibility of undesirable "easter eggs" (e.g., "when a specific, undocumented set of parameters is passed to this method, exit the virtual machine") in any given implementation. To truly establish correctness, the specification must be statically verified against the implementation; if source code is not available, the best we can do is to

perform runtime checking of a conformance test suite and hope (or, when possible, measure with coverage tools) that it completely covers the implementation.

On the other hand, it is impossible to determine the *utility* of a specification through runtime checking regardless of the test suite used. Runtime checking provides no basis for determining whether a specification will enable us to prove the correctness of programs that use the specified system.

In this section, we describe how FCTD allows us to determine the utility of specifications; in the next section, we describe our specific implementations of FCTD for testing JML specifications of Java systems.

3.1 Unit Tests as Operational Behavioral Specifications

As noted above, CTD specifications written for an existing system are *correct* if they can be statically verified. The use of static verification to determine specification correctness is one aspect of FCTD, but the critical hypothesis underlying the process is the following: *if correct CTD specifications for a system are sufficient to allow an existing high-quality and high-coverage test suite for the system to be statically verified, then for all practical purposes the specifications are useful.* By statically verifying a full test suite, we effectively "test" the ability of the specifications to capture the *intended* behavior of the system. The existing test suite is an operational behavioral specification of the system, against which we compare our formal specifications.

To illustrate this idea, consider the method `java.lang.String.getChars` from the Java class library. Its signature and Javadoc documentation (Figure 1) are fairly straightforward. A careful reading of this documentation (plus the knowledge that, in Java, performing array operations on a `null` array causes a `NullPointerException`) yields a 23-line JML specification (Figure 2) with three behavior clauses, one normal and two exceptional. This specification refers to `charArray`, a model field representing the character sequence encapsulated by the `String`, and `equal`, a model method that compares ranges of characters in two arrays for equivalence; both of these are inherited from `java.lang.CharSequence`, an interface implemented by `String`.[1]

Surprisingly, the reference implementation of `getChars` is only 7 statements long. Three `if` statements check the legitimacy of parameters and, if necessary, throw various instances of `StringIndexOutOfBoundsException`. Once the parameters are found to be legitimate, a single call to `System.arraycopy` copies the characters from the string into the destination array.

Verifying the correctness of this specification with respect to the reference implementation is a straightforward proposition—after all, the body of the method is short and, while it has a relatively high cyclomatic complexity given its size, its "shape" matches that of the specification. Moreover, the specification of `System.arraycopy` is strong and well-used.

But how do we measure the utility of our 23-line `getChars` specification? To accomplish this, we turn our attention to the conformance test suite for the

[1] The specification would, of course, be significantly longer if it did not use the inherited model field and model method.

`public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`

Copies characters from this string into the destination character array. The first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1` (thus the total number of characters to be copied is `srcEnd-srcBegin`). The characters are copied into the subarray of `dst` starting at index `dstBegin` and ending at index `dstbegin+(srcEnd-srcBegin)-1`.

(parameter descriptions omitted; they contain no restrictions on parameter values)

Throws `IndexOutOfBoundsException` if any of the following is true: `srcBegin` is negative; `srcBegin` is greater than `srcEnd`; `srcEnd` is greater than the length of this string; `dstBegin` is negative; `dstBegin+(srcEnd-srcBegin)` is larger than `dst.length`.

Fig. 1. Javadoc documentation for library method `java.lang.String.getChars`

```
  public normal_behavior
    requires srcBegin >= 0
          && srcBegin <= srcEnd
          && srcEnd <= charArray.length
          && dst != null
          && dstBegin >= 0
          && dst.length >= dstBegin + (srcEnd - srcBegin);
    modifies dst[dstBegin .. dstBegin+srcEnd-srcBegin-1];
    ensures equal(charArray, srcBegin, dst, dstBegin, srcEnd - srcBegin);
  also
   public exceptional_behavior
    requires srcBegin < 0
          || srcBegin > srcEnd
          || srcEnd > charArray.length
          || dstBegin < 0
          || (dst != null && dst.length < dstBegin + (srcEnd - srcBegin));
    modifies \nothing;
    signals_only IndexOutOfBoundsException;
  also
   public exceptional_behavior
    requires dst == null;
    modifies \nothing;
    signals_only NullPointerException;
```

Fig. 2. JML specification for library method `java.lang.String.getChars`

Java class library, the Java Compatibility Kit (JCK), which includes tests for `getChars` in the class `GetCharsTest`. This class is around 450 lines in length and contains nine comprehensive tests that call `getChars` a total of 36 times.

If we *execute* this test suite against the reference implementation of `getChars`, we exercise the implementation against the idea of correctness that the test writers had in mind when writing the tests. If, on the other hand, we attempt to *prove* that the unit tests always pass, we exercise our *specification* of `getChars` against that same idea of correctness.

One potential issue with this technique is *overspecification*—generating specifications detailed enough to prove that all the unit tests pass but too cumbersome or too highly specific to the particular tested situations for developers to use in general applications. We attempt to avoid overspecification in two ways: first, we develop specifications only from publicly-available documentation (including

test suites when necessary to clarify ambiguous documentation) and not directly from source code; second, we use conformance tests or other high-coverage, high-quality test suites in an effort to cover enough functionality that "specifying to the tests", if it is done at all, actually results in useful specifications.

We return to the JCK and `getChars` in Section 4.2; first, however, we explain how we combine unit testing and static verification to prove that the unit tests always pass.

3.2 Unit Test Specifications

In order to statically verify a test T, we must have specifications for both T itself and the test framework F against which T is written. Otherwise, T would be trivially verifiable as it need not maintain or establish any particular properties.

The specification for each test T is simple, and happens to match the default specification that ESC/Java2 assigns to any method that has no specification. Its precondition is `true`, as there are no constraints on when the unit test can be run;[2] its postcondition is also `true`, as we expect the test to terminate normally. Additionally, no exceptions should be thrown, so its *exceptional* postcondition is `false`. In essence, such a specification says that all unit tests should pass, no unit tests should fail and the test suite should not halt unexpectedly.

While test frameworks typically have a very rich set of methods to assert test conditions, we presume that test framework F contains only two methods—`Assert(boolean P)` and `Fail()`—and that all other assertion methods in F are defined in terms of these two methods. The `Assert` method does nothing if `P` is `true` and reports a test failure if `P` is `false`, while the `Fail` method unconditionally reports a test failure.

Framework F records a passing result for test T if T terminates normally. T may call the `Assert` method an arbitrary number of times to check the validity of an arbitrary number of predicates; if any of these assertions fail, F records a failing result for T. F also, of course, records a failing result for T if T calls the `Fail` method.

The specification of `Assert(P)` is {`P`} `Assert(P)` {`P`}. This specification forces the verifier to check that `P` holds, as it is the precondition of `Assert`. This is logically equivalent to simply asserting `P`. In case the test calls other methods of the framework API in the same method body, we also require that `P` holds as the postcondition.

The specification of `Fail` is {`false`} `Fail` {`true`}. This specification forces the verifier to attempt to prove an unprovable verification condition, and thus will always fail verification. The idea here is that the verification systems we use have the ability to reason about unreachable code, and one way of checking for such is to attempt to assert `false` in unreachable blocks [12]. Since unit tests should never fail, all calls to `Fail` should be unreachable during static analysis.

[2] Unit tests that depend on the results of other unit tests, which can be written in certain testing frameworks, have somewhat more complex preconditions; however, such frameworks typically handle the ordering of such tests automatically.

With this combination of automatic default specification for the unit test and novel specification of test framework assertions, successful verification of a given unit test means that the unit test always passes: neither `Assert(false)` nor `Fail` is ever called. Therefore, it demonstrates that the CTD specification is sufficient to guarantee the success of the unit test. We measure specification utility as the percentage of unit tests in a test suite that can be statically verified; clearly, the correspondence between this measure of utility and actual "real-world" utility for developers is highly dependent on the quality of the test suite we verify. In particular, successful verification of a library conformance test—effectively, a complete definition of the required library behavior—against a library specification means that generalized client programs of the library should also be verifiable against the library specification (though they may still, of course, be unverifiable for reasons having nothing to do with the library specification).

Now that we have described the general idea of using verification to test the correctness and utility of specifications, we will discuss our concrete applications of this general idea using Java, JML, ESC/Java2, and Java unit testing frameworks.

4 The Concrete Process

We have been using the FCTD process for several years when writing new, or reworking existing, specifications of the Java API. An early variant of the concrete process was proposed by David Cok during the development of ESC/Java2. That process, which only used hand-written unit tests, the ESC/Java2 static checker and the JUnit framework, was used to write specifications of several classes. More recently one of the authors focused on refining the process, incorporating support for both static and runtime checking and using tests from the Java Compatibility Kit (JCK) [11].

In this section, we describe the modifications we have made to use JUnit and the JCK with the FCTD process, as well as giving background information on the organization of the JCK and its companion JavaTest framework.

4.1 JUnit

The FCTD process was originally designed for use with JUnit-like frameworks; therefore, the JML specifications added to the JUnit testing framework in order to successfully carry out FCTD are essentially those discussed in Section 3.2.

The JUnit API's key class, `org.junit.Assert`, has many methods that allow the test writer to assert various conditions. The most important two such methods are `assertTrue(String message, boolean condition)`, which asserts that `condition` is `true`, and `fail(String message)`, which causes an unconditional test failure. Figure 3 shows the JML specifications added to those methods for FCTD. A number of other methods, including `assertFalse` and `assertNull`, are implemented in terms of `assertTrue`; still others, such as `assertEquals` (for object equivalence) have their own implementations and we do not show the JML specifications for these here.

```
/** Asserts that a condition is true. */
//@ public normal_behavior
//@   requires condition;
//@   ensures condition;
public static void assertTrue(/*@ nullable @*/ String message,
                              boolean condition);

/** Fails a test with no message. */
//@ public normal_behavior
//@   requires false;
//@   ensures true;
public static void fail(/*@ nullable @*/ String message);
```

Fig. 3. The specification of key methods in JUnit's `Assert` class

With these modifications to the JUnit API, we can perform FCTD with any existing set of JUnit tests for a JML-annotated SUT. We have sets of hand-written JUnit tests for a small selection of classes in the Java class library, as described later in Section 5; however, the conformance tests in the Java Compatibility Kit are preferable to our JUnit tests for testing the Java class library.

4.2 The Java Compatibility Kit

An implementation of the Java class library typically consists of a combination of Java code and native libraries. There are many such implementations, even on the same hardware and OS platform; Sun themselves implemented the class library for three platforms (Solaris, Windows, Linux), Apple implemented it for Mac OS and Mac OS X, and other implementations have been developed by companies such as IBM and Hewlett-Packard and open-source groups such as the Apache Harmony [21] and GNU Classpath [9] projects. In addition, OpenJDK [19], an open-source (GNU GPLv2+Classpath) version of Java first made available by Sun in 2007 and currently maintained by Oracle, runs on many different hardware and operating system platforms.

To ensure that the multitude of Java class library implementations would be mutually compatible and conform to the Java standard, Sun developed an extensive conformance test suite called the Java Compatibility Kit (JCK). Java licensees are required to ensure that their implementations pass the JCK tests before they can use the Java trademark.

Initially, Sun released the JCK to the public with a *read-only* license; the source code for the JCK was publicly available, but developers were explicitly permitted only to read the source code and not to compile or execute it.[3] The read-only license was one of the inspirations for the FCTD process, as it led us to consider ways in which we could make use of this conformance test suite while neither compiling nor executing it. Since the release of OpenJDK, however, Sun (now Oracle) has licensed the JCK for use by developers who are running it in

[3] Sun's (now Oracle's) licensees, of course, have always had full access to the JCK under more liberal license terms.

```
//@ public normal_behavior
//@   ensures \fresh(\result);
//@   ensures \result.isPassed();
//@   ensures stringEquals(\result.getReason(), reason);
public static /*@ pure non_null @*/ Status passed
(/*@ nullable @*/ String reason);

//@ public normal_behavior
//@   requires false;
//@   ensures \fresh(\result);
//@   ensures \result.isFailed();
//@   ensures stringEquals(\result.getReason(), reason);
public static /*@ pure non_null @*/ Status failed
(/*@ nullable @*/ String reason);
```

Fig. 4. The specification of key methods in JavaTest's `Status` class

conjunction with OpenJDK development or with projects that derive substantially from OpenJDK (such as the OpenJML project) [20].

JavaTest. The JCK test suite is designed to be executed within a framework called JavaTest, which was also developed by Sun. The source for the JavaTest framework is available separately from the JCK distribution [18] and is (primarily) released under the same open-source license as OpenJDK. Thus, we are able to modify the JavaTest framework in a way that allows us to attempt static verification of all the unit tests in the JCK (and, for that matter, any other test suites written for the JavaTest framework).

The JavaTest API differs significantly from most standard test APIs in that it does not enable test code to directly assert predicates. Instead, "status" objects are constructed to indicate whether a given test branch passed or failed in some fashion. Thus, there is no direct analogue to the `Assert` method in the JavaTest API. When a test branch determines that it has succeeded, it always constructs and returns a status object representing a "success". Similarly, when a test branch determines that it has failed, it always constructs and returns a status object representing a "failure".

Our modifications to the framework are primarily to the class `com.sun.javatest.Status`, which implements the "status" object and therefore encapsulates the result of a single test. `Status` contains factory methods for creating objects that indicate that a test has passed, failed, or caused an error, and these methods are used by the JCK tests to report their outcomes.

To use FCTD with the JCK, we added JML specifications to these factory methods in the manner described in Section 3.2; these appear in Figure 4.[4] Note that the `failed` factory has a precondition of `false`, while the `passed` factory need not have pre- or postconditions relating to an asserted predicate because it always indicates a passed test. We also added minimal specifications to some related classes to ensure that we did not cause any `NullPointerException`s or similar runtime issues.

[4] `stringEquals` is shorthand for "the two strings are equivalent or are both `null`."

```
public Status String0061() {
  String testCaseID = "String0061";
  String s = "getChar Test"; //step Create a String
  char[] dst = null; //step Create a null reference
  try {
    s.getChars(1,3,dst,0); //step Try to get chars
  }
  catch (NullPointerException e) { //step Catch an exception
    return Status.passed( "OKAY" );
  }
  return Status.failed( testCaseID + " getChars failed" );
}
```

Fig. 5. The first JCK test method for `String.getChars`

JCK Example. An example will help to clarify the usage of the JavaTest framework for running automated tests. Recall that we discussed the method `java.lang.String.getChars` in Section 3.1. The first JCK test method for `getChars` (also the 61st JCK test method for the `String` class), `String0061`, is replicated in Figure 5. In this test, the success case is the `return` in the `catch` block where a "passed" instance of `Status` is constructed; this particular test is checking to see that passing a `null` array to `getChars` correctly causes a `NullPointerException`. The failure case is the final `return` where a "failed" `Status` is constructed.

The JML specification of `getChars`, therefore, needs to be strong enough to guarantee that a `NullPointerException` is always thrown when `getChars` is called on a `String` and given a `null` destination array. If the specification is sufficient, `String0061` will pass its static verification. The JML specification in Figure 2 correctly guarantees a `NullPointerException` in this instance and allows `String0061` to be statically verified. Interestingly, the specification for `getChars` shipped with some versions of the JML tools (including the most recent version of the Common JML tools released in 2009) does *not* include the exceptional behavior clause with the `NullPointerException`; FCTD would have detected that deficiency by failing to statically verify `String0061` and reporting less than 100% utility for the `String` specification.

5 Case Studies

For our first case study, we wrote or rewrote specifications for 26 commonly-used classes in the Java class library: `AbstractList`, `ArrayList`, `Arrays`, `BitSet`, `Boolean`, `ByteArrayInputStream`, `Character`, `Class`, `Collection`, `Comparable`, `Exception`, `File`, `InputStream`, `Integer`, `List`, `Long`, `Map`, `Math`, `Object`, `Properties`, `Set`, `String`, `StringBuffer`, `System`, `Throwable`, and `Vector`.[5] We used FCTD with hand-written JUnit tests to help ensure that the specifications were both correct and useful.

[5] See `http://kindsoftware.com/trac/mobius/browser/src/mobius.esc/trunk/ESCTools/Escjava/test/jdktests` for details.

The second case study was conducted as a part of Hyland's MSc work [11]. Specifications for three classes that had no existing JML specifications were written and verified using the JCK tests. These classes—`ResourceBundle`, `PrintStream`, and `Stack`—were chosen because they are the most frequently used classes in the Java API (as measured by two static analysis tools) that were missing specifications.

In both case studies, JML specifications for a class C were written using Javadoc documentation for C and any classes of which C is a client. Reading publicly available unit tests was permitted, but not encouraged, and served only to resolve ambiguity in the documentation. In an effort to avoid overspecification, viewing C's source code was *not permitted* under any circumstances.

The struggles and outcomes of this process highlight the fact that natural language specifications, as seen in Javadoc documentation, are often imprecise and incomplete. This was witnessed both during attempts to verify the classes in question against their newly-written specifications and during attempts to test the specifications by verifying the JCK tests and other tests against them. Hyland's MSc report discusses in detail some of the challenges posed and solutions found while writing usable and correct specifications for these three classes.

During both case studies, a specification was deemed *correct* only if (a) a manual review by multiple parties of the informal documentation and the formal specification had a positive outcome, (b) hand-written unit tests all passed when executed with runtime assertion checking of the JML specifications turned on, and (c) ESC/Java2 verified all unit tests successfully. If unit tests could not be executed, as in the case of the JCK, then part (b) was not performed. The utility of a class specification was measured in these case studies as the ratio of the number of verified unit tests for the class to the total number of unit tests run for the class, and no class specification was declared "finished" until its utility was measured at 100%. All class specifications were declared "finished" by the end of the case studies.

As a result of these case studies, we have much higher confidence in the correctness and utility of the tested specifications than we did for specifications that were written in the past using more ad hoc techniques. Consequently, the JML community is continuing to use this process to write (and rewrite) specifications for Java 1.7 as development on the new JML tool suite, OpenJML, continues.

6 Conclusion

We have described a new process, the Formal CTD Process (FCTD), for determining whether formal specifications written for an existing software system are correct and useful. FCTD effectively uses the existing unit test suite for the software system as a behavioral specification and validates the formal specifications against the unit test suite by performing modular static verification. By doing so it ensures that the formal specifications capture enough of the system's behavior to pass the unit tests, demonstrating the utility of the specifications.

FCTD is best suited to testing library specifications using conformance tests, since conformance tests by definition describe exactly the required functionality

of a library. We have described a method for applying FCTD to the standard Java class library using the Java Compatibility Kit and a version of its accompanying JavaTest infrastructure augmented with formal specifications for key classes. This will allow us to evaluate new specifications written for Java library classes against the library conformance tests used by all Java licensees, and thus to ensure that our library class specifications are of high quality.

While the concrete processes we have described here are specific to Java, JML, and their associated tools and tests, the general technique of using static verification of unit tests to validate specifications written for existing software systems is widely applicable. Our process can be easily adapted to any programming language and specification language for which both unit testing frameworks and static verification tools are available. We believe, therefore, that our process has the potential to significantly improve the quality of specifications written for existing software systems, and thereby also to significantly increase the utility of formal verification techniques that rely on such specifications.

There are several open research challenges and opportunities associated with this work. We speculate that unit tests generated by tools that are unaware of specifications (e.g., those that perform symbolic execution, shape analysis, etc.) may have some utility in terms of testing specifications, but we have not yet explored this avenue. We also speculate that the output of *specification inference* tools such as Daikon [6] and Houdini [7], which attempt to infer preconditions, postconditions and invariants from a body of code, may provide good starting points for the generation of correct and useful specifications. Finally, we believe it is possible to provide a more precise measurement of code and specification coverage than the "number of tests" ratio we currently use as a measure of utility.

References

1. Beust, C., Suleiman, H.: Next Generation Java Testing. Addison–Wesley Publishing Company (2007)
2. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (February 2005)
3. Chalin, P., Robby, et al.: JMLEclipse: An Eclipse-based JML specification and verification environment (2011), http://jmleclipse.projects.cis.ksu.edu/
4. Cheon, Y., Leavens, G.T.: A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002)
5. Cok, D.R., et al.: OpenJML (2011), http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml
6. 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. Science of Computer Programming 69(1-3), 35–45 (2007)
7. Flanagan, C., Leino, K.R.M.: Houdini, an Annotation Assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)

8. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. ACM SIGPLAN Notices 37(5), 234–245 (2002)
9. Free Software Foundation, Inc.: GNU Classpath (2011), http://www.gnu.org/software/classpath/
10. Gamma, E., Beck, K.: JUnit: A regression testing framework (2011), http://www.junit.org/
11. Hyland, R.: A Process for the Specification of Core JDK Classes. Master's thesis, University College Dublin (April 2010)
12. Janota, M., Grigore, R., Moskal, M.: Reachability analysis for annotated code. In: 6th International Workshop on the Specification and Verification of Component-based Systems (SAVCBS 2007), Dubrovnik, Croatia (September 2007)
13. Kiniry, J.R., Cochran, D., Tierney, P.: A verification-centric realization of e-voting. In: International Workshop on Electronic Voting Technologies (EVT 2007), Boston, Massachusetts (2007)
14. Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting eSC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
15. Kiniry, J.R., Zimmerman, D.M.: Secret Ninja Formal Methods. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 214–228. Springer, Heidelberg (2008)
16. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 262–284. Springer, Heidelberg (2003)
17. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Inc. (1988)
18. Oracle Corporation: JT Harness project (2011), http://jtharness.java.net/
19. Oracle Corporation: OpenJDK (2011), http://openjdk.java.net/
20. Oracle Corporation: OpenJDK community TCK license agreement (2011), http://openjdk.java.net/legal/openjdk-tck-license.pdf
21. The Apache Software Foundation: Apache Harmony - Open Source Java SE (2011), http://harmony.apache.org/
22. The Eclipse Foundation: The Eclipse project (2011), http://www.eclipse.org/
23. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys 29(4), 366–427 (1997)
24. Zimmerman, D.M., Nagmoti, R.: JMLUnit: The Next Generation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 183–197. Springer, Heidelberg (2011)

Incremental Model-Based Testing of Delta-oriented Software Product Lines

Malte Lochau[1], Ina Schaefer[2], Jochen Kamischke[1], and Sascha Lity[1]

[1] TU Braunschweig, Institute for Programming and Reactive Systems, Germany
{m.lochau,j.kamischke,s.lity}@tu-bs.de
[2] TU Braunschweig, Institute for Software Systems Engineering, Germany
i.schaefer@tu-bs.de

Abstract. Software product line (SPL) engineering provides a promising approach for developing variant-rich software systems. But, testing of every product variant in isolation to ensure its correctness is in general not feasible due to the large number of product variants. Hence, a systematic approach that applies SPL reuse principles also to testing of SPLs in a safe and efficient way is essential. To address this issue, we propose a novel, model-based SPL testing framework that is based on a delta-oriented SPL test model and regression-based test artifact derivations. Test artifacts are incrementally constructed for every product variant by explicitly considering commonality and variability between two consecutive products under test. The resulting SPL testing process is proven to guarantee stable test coverage for every product variant and allows the derivation of redundancy-reduced, yet reliable retesting obligations. We compare our approach with an alternative SPL testing strategy by means of a case study from the automotive domain.

Keywords: Delta-oriented Software Product Lines, Model-based Testing, Regression Testing.

1 Introduction

Diversity is prevalent in modern software systems in order to meet different customer requirements and application contexts [25]. Software product line (SPL) engineering [8] provides a promising approach to develop variant-rich software systems by managed reuse. Since these software systems increasingly control safety- or business-critical applications, it is essential to ensure that they meet their requirements. Recently, there has been considerable progress in applying model checking [24,7,2] and theorem proving [3] to SPLs. However, those techniques are still far from being used in industrial engineering, mainly because of scalability issues, even for single products. Testing is much more established for practical applications in order to ensure that software systems meet their requirements. Testing is indispensable to reveal faults coming from different sources, such as erroneous feature interactions arising from obscured interplays between software and hardware devices [4].

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 67–82, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Testing SPLs product by product is, in general, infeasible due to the high number of products to be tested. Recent SPL testing approaches focus on redundancy reduction by considering *representative* product subsets under test. The subset selection is based on (1) combinatorial criteria on feature models [9,20,22], (2) coverage criteria on SPL test models [5], and (3) coverage criteria on feature interactions [16]. But, until now, few attention in SPL research is paid to the problem how to actually conduct an efficient testing process on those subsets that avoids a traditional product by product process, again, contradicting SPL reuse principles. The new ISO 26262 standard for automotive systems even requires comprehensive testing strategies coping with existing system variants [28].

In this paper, we propose a novel approach for incremental model-based testing (MBT) [30] of SPLs based on principles of regression testing [1]. MBT is well-suited for planing reuse potentials in SPL testing [18]. Variable test models are used to explicitly capture behavioral commonality and variability between product variants. On this basis, a concise approach for incrementally assembling and reusing test artifacts for sets of products under test is built.

Our framework comprises state machines as test models extended with delta modeling concepts [6,23] to express variability. When testing a set of products, for each step from a product p to the next product p', an automated adaptation of the test model is performed by applying a regression delta. The regression delta contains the modifications to obtain the test model of product p' from test model of product p. It is computed automatically from the delta modeling structure of the test models. From the regression delta, the test goals for product p', as well as of the set of test cases and retest obligations are derived. Additionally, it can be determined which existing test cases are applicable to product p' and which test results still hold. This framework has two major potentials in SPL testing: (1) test cases can be reused for different product variants while guaranteeing the validity of test cases and the confidential test coverage for every product variant, and (2) test results can be reused according to change impacts between product variants, thus guaranteeing appropriate fault detection efficiency. Our approach is evaluated by means of a case study from the automotive domain w.r.t. previous results obtained from an existing SPL testing approach. It is, to the best of our knowledge, the first SPL MBT framework that captures reuse potentials between different product variants.

The paper is organized as follows. In Sect. 2, foundations of model-based testing are introduced. In Sect. 3, delta modeling for test models is presented. The incremental SPL testing approach is described in Sect. 4 and evaluated in Sect. 5. Sect. 6 discusses related work, and Sect. 7 concludes.

2 Foundations

We briefly introduce the main principles of MBT and regression testing underlying the incremental SPL testing framework developed in the remainder of this paper.

2.1 Model-Based Testing

Model-based testing aims at the automation of black-box testing processes [30]. A *test model* serves as a behavioral specification capturing the functional requirements of a software *product under test* to be verified.

Due to their wide acceptance in industrial control systems engineering, state-machine-like modeling approaches are commonly used as test models. State machine test models define input/output relations by means of sequences of *controllable input* and expected *observable output events*. We focus on basic, i.e., flat basic state machines as test models to keep our illustrations graspable, where the major results are enhanceable to, e.g., UML-like state machine variants providing hierarchy, concurrency, variables etc.

Definition 1. *(State Machine Test Model)*
A state machine test model *is a 4-tuple* $tm = (S, s_0, L, T)$*, where* S *is a finite set of* states*,* $s_0 \in S$ *is the* initial state*,* L *is a set of* transition labels*, and* $T \subseteq S \times L \times S$ *is a* transition relation.

A transition label $l = (\pi_I, \pi_O) \in L = \Pi_I \times \Pi_O$ is a pair of a *controllable* input event $\pi_I \in \Pi_I$ triggering the transition, and an *observable* output event $\pi_O \in \Pi_O$ specifying a system reaction released by the transition, where Π_I and Π_O are disjoint input/output alphabets. We assume state machine test models to be deterministic, and to obey well-formedness properties as usual, i.e., the transition graph has to be *connected* and every state has to be *reachable* from the initial state. By $TM(L)$ we refer to the set of well-formed state machine test models over a label set L.

Test models specify all intended behaviors a product under test is to be verified against by means of *test runs*, i.e., representative executions. Test runs refer to *test cases* derived from a test model $tm \in TM(L)$.

Definition 2. *(State Machine Test Case)*
A test case $tc = (t_0, t_1, \ldots, t_k) \in T^*$ *of a state machine test model* $tm \in TM(L)$ *is a* finite *sequence of* k *transitions of* tm*.*

Test case tc is *valid* for test model $tm \in TM(L)$, written $valid(tc, tm)$, if its transition sequence corresponds to an alternating sequence $s_0, t_0, s_1, \ldots, s_{k-1}, t_{k-1}, s_k$ of states and transitions conforming tm, i.e., (1) it starts in the initial state s_0, and (2) for all segments $(s_i, t_i, s_{i+1}) \in T$, $0 \leqslant i \leqslant k-1$, holds. For a test case $tc = (t_0, t_1, \ldots, , t_{k-1})$, we define a corresponding *test run*:

$$exec(tm, tc) = (l_0, l_1, \ldots, l_{k-1}) \in L^*$$

to be given as the *trace* traversed by tc in tm, i.e., a sequence of labels l_i of transitions $t_i, 0 \leqslant i \leqslant k-1$. We limit our considerations to deterministic behaviors, i.e., a one-to-one correspondence between test runs and test cases. By $TC(tm)$, we denote the set of test cases, i.e., all valid paths of a test model tm.

In MBT, the behaviors of an implementation of product p are verified for test cases tc to *conform* those specified in its test model tm [29]. By $\approx_{te}$, we denote

the *testing equivalence* under consideration in the following, usually some kind of trace equivalence [10]. The equivalence notion applied for the purposes of this paper is discussed in more detail in Sect. 5. According to deterministic behaviors specified in a test model, we assume product variants to also behave deterministically when reasoning about equivalence of test case executions. A product under test p *passes* a test run of a test case $tc \in TC(tm)$, if its observable behavior under the sequence of inputs conforms to the expected output behavior specified in test model tm:

$$p \textbf{ passes } tc \;:\Leftrightarrow\; exec(p, tc) \;\approx_{te}\; exec(tm, tc)$$

A test suite $ts \subseteq TC(tm)$ is a collection of test cases, where:

$$p \textbf{ passes } ts \;:\Leftrightarrow\; \forall tc \in ts : p \textbf{ passes } tc$$

A test model tm (and thus $TC(tm)$) potentially contains (1) an infinite number of paths, as well as (2) paths of infinite length. For the test conformance to be decidable, test suites $ts \subseteq TC(tm)$ are restricted to those with (1) a finite number of test cases, and (2) each test case to be of finite length. Adequacy criteria for selecting appropriate test suites from test models tm usually require structural elements in tm to be *traversed* at least once. For state machines, such *coverage criteria* are *all-states*, *all-transitions*, etc. [30]. Formally, a coverage criterion C applied to a test model tm selects a finite set of *test goals*:

$$tg = C(tm) = \{g_1, g_2, \ldots, g_n\}$$

for instance, $tg = T$, i.e., the set of all transitions in tm. We write $covers(tc, g)$ for a test case $tc \in ts \subseteq TC(tm)$, if the test goal g is traversed in test model tm via tc. A test suite ts *satisfies* coverage criterion C, if:

$$\forall g \in C(tm) : \exists tc \in ts : covers(tc, g)$$

Summarizing, the set of test artifacts for a product p is given as follows.

Definition 3. *(Product Test Artifacts)*
The collection of test artifacts *for product p is a 4-tuple $ta_p = (tm_p, tg_p, ts_p, tp_p)$ consisting of a* test model *tm_p, a finite set tg_p of* test goals *in tm_p for criterion C, a* test suite *ts_p, and a* test plan *tp_p.*

A *test plan* organizes the test suite application by further (de-)selecting, prioritizing, etc. test cases from ts. We assume test plans simply to be subsets $tp_p \subseteq ts_p$ containing those test cases actually to be (re-)tested on product p. In case of single product testing, we assume $tp_p = ts_p$.

Example 1. Consider the state machine test model in Fig. 1 consisting of states $S = \{s0, s1, s2\}$ and transitions $T = \{t0, t1, t2, t3\}$. Assuming C is the *all-transitions* coverage criterion, the set of test goals is given as $tg = \{t0, t1, t2, t3\}$. A sample test suite $ts = \{tc1, tc2\}$ that satisfies C consists, e.g., of two test cases $tc1 = (t0, t1)$ and $tc2 = (t0, t2, t3)$, where $tc1$ covers $t0$ and $t1$ and $tc2$ covers $t0$, $t2$, and $t3$. A test run of $tc2$ corresponds to the sequence $exec(tc2, tm) = ((\pi_1, \pi_2), (\pi_3, \pi_5), (\pi_3, \pi_2))$.

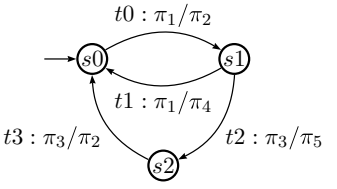


Fig. 1. Sample State Machine Test Model

For the following discussions, we assume the existence of some (black-box) *test case generator* (cf. e.g. [12]) and write $tc = gen(tm, g)$ to generate a test case that covers test goal g on test model tm, and $ts = gen(tm, C)$ for the generation of entire test suites satisfying coverage criterion C on test model tm.

2.2 Regression Testing

The purpose of regression testing is to efficiently verify that changes between different *versions* of a product are as intended [1,11]. For a software product implementation version p evolving to version p' over time, regression testing strategies aim at verifying that (1) the changes are correctly implemented, and (2) the changes do not erroneously influence parts of p reused in p' other than intended. When stepping to the next version p', the test suite evolves accordingly such that $ts' = gen(tm', C)$ is executed on p'. For non-naive regression testing approaches, reuse potentials between p and p' arise, namely (1) the reuse of *test cases* in ts generated from tm in ts' for reducing test generation efforts, and (2) reuse of *test execution results* for test cases in ts applied to p for p', i.e., reducing test execution efforts. For the reuse of test cases $tc \in ts$ in ts', we require:

$$exec(tc, tm) \approx_{te} exec(tc, tm')$$

Thus, tc concerns system reactions equivalently specified in tm and tm', i.e., addressing behaviors similar to p and p'. For the reuse of test execution results obtained from $exec(tc, p)$ for reusable test cases $tc \in ts'$, we further require:

$$exec(tc, tm) \approx_{te} exec(tc, tm') \Rightarrow exec(tc, p) \approx_{te} exec(tc, p')$$

This second reuse problem refers to the well-known retest selection problem of regression testing [1]: select from the set ts_R of reusable test cases a minimum retest subset $ts_{RT} \subseteq ts_R$ such that ts_{RT} is capable to cover all potentially erroneous impacts of changes between the product implementations p and p':

$$exec(ts_{RT}, p') \approx_{te} exec(ts_{RT}, p) \Rightarrow exec(ts_R, p') \approx_{te} exec(ts_R, p)$$

Regression testing approaches categorize test cases into sets of *reusable* $ts_R \subseteq ts$, *obsolete* $ts_O = ts \backslash ts_R$, and *new* $ts_N = ts' \backslash ts_R$ test cases. The set of test cases to be (re-)executed on p' contains the *retest* set $ts_{RT} \subseteq ts_R$, as well as all new test cases in ts_N.

3 Delta-Oriented SPL Test Modeling

In order to apply an incremental MBT approach to SPLs, we need a reusable test model to capture the commonality and variability in a closed form instead of storing each test model variant separately. We base our approach on the concept of delta modeling [6,23], a modular and flexible variability modeling approach that is well suited as basis for regression-based SPL testing by incrementally evolving test artifacts for product variants. In delta modeling, a family of similar products is captured by a designated core product and a set of deltas encapsulating changes to the core model. A delta adds and removes elements from the core product. If there are hierarchically structured elements, a delta operation can be used to change the internal structure of these elements. A product variant is obtained by selecting a subset of the available deltas, determining a suitable ordering and applying the operations of the deltas one by one to the core product.

We apply the principles of delta modeling to state machine test models. A delta over a state machine test model, as defined in Def. 1, can add and remove states and transitions. Changing the label of a transition can be encoded by removing a transition and adding a transition between the same states with a different label. The following definition introduces the syntax of state machine deltas.

Definition 4. *(State Machine Delta)*
A state machine delta *is a set of delta operations $\delta \subseteq Op$, where Op contains (1) for every $s \in \mathcal{S}$, $\{add\ s\}$ and $\{rem\ s\}$, and (2) for every $t \in \mathcal{T}$, $\{add\ t\}$ and $\{rem\ t\}$ for finite sets of possible states $\mathcal{S}$ and transitions $\mathcal{T}$.*

The application of a set of delta operations transforms one state machine into another. A delta is *applicable* to a state machine if the states and transitions to be removed exist and if the states and transitions to be added do not yet exist. A delta is *consistent* if it only adds or removes each state or transition once.

Definition 5. *(State Machine Delta Application)*
The application *of an applicable and consistent delta $\delta \subseteq Op$ to a state machine $tm = (S, s_0, L, T)$ defines a function $apply : TM(L) \times \mathcal{P}(Op) \rightarrow TM(L)$ such that $apply(tm, \delta) = tm' = (S', s_0, L, T')$ where*

- *if $\delta = \emptyset$, $tm' = tm$*
- *if $\delta = \{op\} \cup \delta'$, then $tm' = apply(apply(tm, op), \delta')$*
- *for $\delta = \{add\ s\}$, we have $S' = S \cup \{s\}$ and $T' = T$*
- *for $\delta = \{rem\ s\}$, we have $S' = S \backslash \{s\}$ and $T' = T$*
- *for $\delta = \{add\ t\}$, we have $T' = T \cup \{t\}$ and $S' = S$*
- *for $\delta = \{rem\ t\}$, we have $T' = T \backslash \{t\}$ and $S' = S$*

In order to describe the set of possible test models for an SPL, we connect the deltas to the product variants. Each product test model is defined by a set of deltas to be applied to a given core test model tm_{core} in order to generate the test model of the variant. Instead of specifying sets of deltas for each product

test model, the connection can also be made by associating deltas to product features [6]. A suitable ordering of delta application has to be defined such that each delta is applicable to the respective model when it is used. The test model of the product variant is obtained by applying the given deltas in the specified ordering to the core test model tm_{core}. During the generation process, it is possible that an intermediate model is constructed that is not well-formed. However, after applying all deltas, it has to be guaranteed that the resulting test model is well-formed. A more detailed description of the product generation process in delta modeling can be found in [23].

To allow for a flexible delta-oriented SPL test modeling, any potential test model should be usable as core model. This means that a state machine delta has to exist to derive every valid test model variant from that arbitrary core model.

Proposition 1. *(Existence of State Machine Delta)*
For each core state machine test model $tm_{core} \in TM(L)$ *and each test model variant* $tm \in TM(L)$*, there exists a state machine delta* $\delta \subseteq Op$ *such that* $tm = apply(tm_{core}, \delta)$ *holds.*

Proof: For any potential $tm_{core} = (S, s_0, L, T)$ and test model variant $tm = (S', s_0, L, T')$, we have to show that there exist delta operations $\delta \subseteq Op$ that are sufficient to transform the sets S and T to S' and T', respectively. For each $s \in S'$, three cases arise: (1) for states $s \in S \cap S'$ no delta operation is required, (2) for states $s \in S \backslash S'$, s can be removed from S to build S' via $\{rem\ s\}$, and (3) for states $s \in S' \backslash S$, s can be added to S to built S' via $\{add\ s\}$. For transitions, the same cases hold.

Example 2. Consider Fig. 2. The state machine introduced in Fig. 1 now serves as the *core model*. By applying the delta operations of δ_{tm}, we obtain the left test model variant *tm*. By applying the delta operations of $\delta_{tm'}$, we obtain the right test model variant tm'.

4 Delta-Oriented SPL Regression Testing

When applying MBT to SPLs, i.e., a family of similar product variants $P = \{p_1, p_2, \ldots p_n\}$ with explicit commonality and variability, a corresponding collection of *test artifacts* $ta_i = (tm_i, tg_i, ts_i, tp_i)$ is to be provided for every product variant $p_i \in P$. The artifact construction and application of test suites ts_i to implementations of product variants $p_i \in P$ is usually done in some ordering. The result is a chain of product testing campaigns continuously stepping from test artifacts *ta* of variant *p* to the next product test artifacts ta' of variant p'. Reuse potentials between *ta* and ta' arise by incrementally promoting previous test artifacts to subsequent products under test. In contrast to classical regression scenarios, differences between product variants are explicitly specified beforehand in an SPL, e.g., on the basis of a reusable test model.

Based on delta-oriented state machines as reusable SPL test models, we define a model-based SPL regression testing approach that assembles product-specific

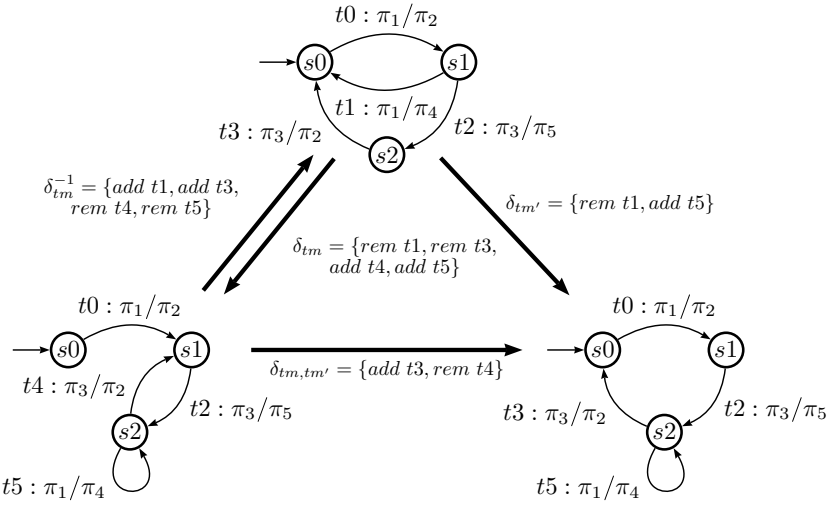


Fig. 2. Sample Delta-oriented SPL Test Model and Regression Delta Derivation

test artifacts by incrementally reusing test artifacts of previous products. In particular, we incrementally evolve product test artifacts for a sequence $p_1, p_2, \ldots, p_n$ of products under test as follows:

1. Generate an initial collection of product test artifacts ta_1 using MBT techniques for single products as usual and apply the resulting test suite ts_1 to the implementation of p_1.
2. Incrementally evolve ta_i to ta_{i+1}, for $1 \leqslant i < n$, and apply the new (re-)test plan $tp_{i+1} \subseteq ts_{i+1}$ to p_{i+1}.

Although p_1 might be chosen arbitrarily, we suggest to start the incremental SPL testing campaign with the core product p_{core} as it usually comprises most of the commonalities among product variants.

As illustrated in Fig. 3, the incrementation of product test artifacts ta for product p to ta' of a subsequent variant p' decomposes into four levels. For each incrementation from p_i to p_{i+1}, (1) the reuse of product test artifacts from ta_i in ta_{i+1}, as well as (2) the generation of new artifacts required for ta_{i+1} is to be performed. Both steps are to be conducted in a way that ensures the different components of ta_{i+1} to meet the requirements according to their relationships (cf. Sect. 2), namely *validity* of test cases $tc \in ts_{i+1}$ w.r.t. tm_{i+1}, *coverage* of test goals $g \in tg_{i+1}$ for criterion C by test suite ts_{i+1}, and appropriate *(re-)test* selections for test plans $tp_{i+1} \subseteq ts_{i+1}$.

Accordingly, we apply the delta approach to also reason about the incremental changes on test artifacts from ta to ta', where $\delta_{ta,ta'}$ is decomposed into sub deltas for the different components of test artifact collections:

$$\delta_{ta,ta'} = (\delta_{tm,tm'}, \delta_{tg,tg'}, \delta_{ts,ts'}, \delta_{tp,tp'})$$

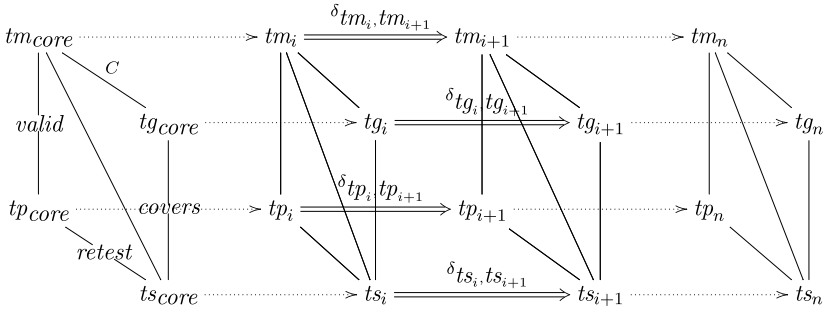


Fig. 3. Incremental Evolution of SPL Test Artifacts

The test model is the central part for deriving any kind of test artifacts in MBT. As a consequence, the sub deltas on the remaining *product test artifacts* are directly deducible from changes on test model specifications.

Test Model Delta. One of the main benefits of a delta-oriented SPL test model is its ability to comprehensively encapsulate the differences of every product variant w.r.t. some core model. However, for the incremental evolution of product test artifacts, we rather have to make explicit the differences of a product variant p' to the previous product p under test. Therefore, we introduce the concept of *regression deltas* to aggregate all changes when evolving from tm to tm'.

Definition 6. *(State Machine Regression Delta)*
A state machine regression delta $\delta_{tm,tm'} \subseteq Op$ *for state machine pair* $(tm, tm') \in TM(L) \times TM(L)$ *is a state machine delta such that* $tm' = apply(tm, \delta_{tm,tm'})$.

The application of a state machine regression delta on a test model yields the subsequent test model variant. As illustrated in Fig. 2, by intuition, a regression delta $\delta_{tm,tm'}$ results from composing the inverted delta δ_{tm}^{-1} of tm and the delta $\delta_{tm'}$ of tm'. The *inverse* $\delta^{-1} \subseteq Op$ of a state machine delta $\delta \subseteq Op$ is built component-wise, i.e., by inverting each delta operation $op \in \delta$ to op^{-1} in δ^{-1} such that $\{add\ e\}^{-1} = \{rem\ e\}$ and $\{rem\ e\}^{-1} = \{add\ e\}$ for $e \in \mathcal{S} \cup \mathcal{T}$.

However, using set union to compose δ_{tm}^{-1} and δ'_{tm} into $\delta_{tm,tm'}$ produces unsound results in case of equal delta operations. For instance, in Fig. 2, $\{add\ t5\} \subseteq \delta_{tm} \cap \delta_{tm'}$ holds, thus set union would yield $\{rem\ t5,\ add\ t5\} \subseteq \delta_{tm,tm'}$, i.e., conflicting operations in the regression delta. Instead, for the correct derivation of regression delta $\delta_{tm,tm'}$ from state machine deltas $\delta_{tm} \subseteq Op$ and $\delta_{tm'} \subseteq Op$, we have to apply an alternative composition operator that takes common delta operations into account. The *symmetric difference* $A \,\Delta\, B = (A \backslash B) \cup (B \backslash A)$ of two sets A and B solely contains those elements being either exclusive to set A or B. In addition, to build the regression delta, we require the first operand of the symmetric difference to be inverted.

Proposition 2. *(State Machine Regression Delta Construction)*
For two state machine deltas $\delta_{tm} \subseteq Op$ *and* $\delta_{tm'} \subseteq Op$, *the regression delta is given as* $\delta_{tm,tm'} = (\delta_{tm} \backslash \delta_{tm'})^{-1} \cup (\delta_{tm'} \backslash \delta_{tm})$.

Proof: For $tm' = (S', s_0, L, T')$ to result from applying $\delta_{tm,tm'} = (\delta_{tm} \backslash \delta_{tm'})^{-1} \cup (\delta_{tm'} \backslash \delta_{tm})$ to $tm = (S, s_0, L, T)$, we have to show the sets S' and T' to be built correctly from S and T. For states $s \in S'$, we have two cases: (1) $s \in S$, and (2) $s \notin S$. For case (1), we have to show that $\{rem\ s\} \notin \delta_{tm,tm'}$, where we have two further cases: (1a) $s \in S_{core}$, thus $\{add\ s\} \notin \delta_{tm}$ which implies $\{rem\ s\} \notin (\delta_{tm} \backslash \delta_{tm'})^{-1}$, and (1b) $s \notin S_{core}$, thus $\{add\ s\} \subseteq \delta_{tm}$, but $\{rem\ s\} \notin (\delta_{tm} \backslash \delta_{tm'})^{-1}$, because $\{add\ s\} \notin (\delta_{tm} \backslash \delta_{tm'})$. For case (2), we have to show that $\{add\ s\} \subseteq \delta_{tm,tm'}$, where, again, two further cases arise: (2a) $s \in S_{core}$, thus $\{rem\ s\} \subseteq \delta_{tm}$ which implies $\{add\ s\} \subseteq (\delta_{tm} \backslash \delta_{tm'})^{-1}$, and (2b) $s \notin S_{core}$, thus $\{add\ s\} \notin (\delta_{tm} \backslash \delta_{tm'})^{-1}$, but $\{add\ s\} \subseteq (\delta_{tm'} \backslash \delta_{tm})$. Symmetric cases arise for ensuring states $s \notin S'$ are either removed if $s \in S$, or not added if $s \notin S$ via $\delta_{tm,tm'}$. Further note, that these cases also hold for the set of transitions. Finally, the existence of a regression delta for arbitrary pairs of state machines follows directly from Prop. 1: as any test model variant is derivable from an arbitrary core model by a set of delta operations, any test model tm can be assumed as core model to derive the test model of tm'.

Example 3. The regression delta between the test model tm and tm' in Fig. 2 results in $\delta_{tm,tm'} = (\delta_{tm} \backslash \delta_{tm'})^{-1} \cup (\delta_{tm'} \backslash \delta_{tm}) = \{add\ t3,\ rem\ t4\}$. As both products share the delta operations concerning $t1$ and $t5$, those transitions are not affected by the regression delta.

Please note, that regression deltas constitute a generalization of state machine deltas, i.e., δ_{tm} can be represented as $\delta_{tm_{core},tm}$.

We now describe the derivation of the deltas concerning the incrementation of the three remaining test artifacts from the state machine regression delta. Those deltas are similar to those for state machines (cf. Sect. 3), but are to be adapted to artifact types considered in the particular components of *ta*.

Test Goal Delta. The construction of the delta $\delta_{tg,tg'}$ for the incrementation of the set of test goals depends on the coverage criterion C considered. For simple structural criteria such as *all-states* and *all-transitions*, i.e., criteria with $C(tm) \subseteq \mathcal{S} \cup \mathcal{T}$, $\delta_{tm,tm'}$ is directly adaptable to evolve the test goals via the following rules:

- $\forall \{rem\ e\} \subseteq \delta_{tm,tm'} : e \in tg \Rightarrow \{rem\ e\} \subseteq \delta_{tg,tg'}$
- $\forall \{add\ e\} \subseteq \delta_{tm,tm'} : e \in C(tm') \Rightarrow \{add\ e\} \subseteq \delta_{tg,tg'}$

Otherwise, for more complex criteria, e.g., path-oriented criteria like MC/DC coverage [30], a (partial) regeneration of test goals via $C(tm')$ is required, where $\delta_{tm,tm'}$ indicates model parts in tm' potentially affected.

Test Suite Delta. As described in Sect. 2.2, regression testing approaches partition an existing test suite ts of product p into subsets of reusable tests ts_R and obsolete tests ts_O when evolving to product p'. For our incremental SPL testing approach it seems promising not to discard obsolete test cases in the next test suite ts', but rather to collect them for potential reuse for subsequent products under test. Therefore, we partition product test suites $ts = ts_V \cup ts_O$ into sets

of *valid* and *obsolete* test cases. When evolving $ts = ts_V \cup ts_O$ to $ts' = ts'_V \cup ts'_O$ via $\delta_{ts,ts'}$, changes in $\delta_{tm,tm'}$ have effects on the incrementation of both sets. Accordingly, we also partition the test suite delta into δ_{ts_V,ts'_V} and δ_{ts_O,ts'_O}.

By $T_{tc} \subseteq \mathcal{T}$, we refer to the subset of transitions from $\mathcal{T}$ such that (1) $tc \in T^*_{tc}$, and (2) T_{tc} is *minimal*. Thus, a test case tc is valid for test model $tm = (S, s_o, L, T)$, if $T_{tc} \subseteq T$, whereas $T_{tc} \nsubseteq T$ holds for obsolete test cases. A test case $tc \in ts_O$ being obsolete for p becomes valid for p' as follows:

$$\forall t \in T_{tc} \backslash T : \exists \{add\ t\} \subseteq \delta_{tm,tm'} \Rightarrow \{add\ tc\} \subseteq \delta_{ts_V,ts'_V} \wedge \{rem\ tc\} \subseteq \delta_{ts_O,ts'_O}$$

i.e., the set of transitions of tc missing in the set T of the test model of p is added to p' via the regression delta. Correspondingly, valid test cases $tc \in ts_V$ become obsolete by the rule:

$$\exists t \in T_{tc} : \{rem\ t\} \subseteq \delta_{tm,tm'} \Rightarrow \{add\ tc\} \subseteq \delta_{ts_O,ts'_O} \wedge \{rem\ tc\} \subseteq \delta_{ts_V,ts'_V}$$

The set of reusable test cases $ts'_R = ts_V \cap ts'_V$ therefore contains those test cases valid for p, as well as for p'. In addition to ts'_R, further test cases may be required in ts to cover all test goals in tg'. A test goal $g \in tg'$ is uncovered by ts'_R if either

- $\{add\ g\} \subseteq \delta_{tg,tg'}$, i.e., the test goal is new in p', or
- $\forall tc \in ts_V : covers(tc, g) \Rightarrow tc \in ts'_O$, i.e., all test cases of p covering g are obsolete for p'.

For covering those test goals, further previously obsolete test cases $tc \in ts_O \cap ts'_V$ with $covers(tc, g)$ may be found and added to ts'_R. Otherwise, a new test case $tc_g = gen(tm', g)$ is required, where $\{add\ tc_g\} \subseteq \delta_{ts_V,ts'_V}$. The set of all new test cases generated for p' thus gives the set ts'_N in terms of regression testing.

Example 4. Consider the test cases $tc1 = (t0, t1)$ and $tc2 = (t0, t2, t3)$ of Example 1 for *all-transition* coverage. When stepping from the core model to tm (cf. Fig. 2), $tc1$ and $tc2$ both become obsolete, thus new test cases, e.g., $tc3 = (t0, t2, t5)$ and $tc4 = (t0, t2, t4)$ are generated. For tm', again, $tc1$ is obsolete, whereas $tc2$ as well as $tc3$ are reusable and cover all test goals.

Test Plan Delta. Test plans $tp \subseteq ts_V$ are used to define which valid test cases from a test suite are actually executed on the product under test, where $tp = ts_N \cup ts_{RT}$. New test cases $tc \in ts_N$ are applied in any case to verify that new, i.e., varying behaviors are correctly implemented. In addition, from the set of reusable test cases ts_R, a retest set $ts_{RT} \subseteq ts_R$ is selected to verify that the changes do not erroneously affect common behaviors covered by ts_R. For the selection of ts_{RT}, different strategies appear in the literature [11], e.g., *retest-all* $ts_{RT} = ts_R$, *retest-non* $ts_{RT} = \varnothing$, and *retest-random*, where some $ts_{RT} \subseteq ts_R$ is chosen. In addition, techniques for change impact analyses such as *program slicing* [13] support the retest selection decision by the following criterion:

$$tc \in ts_{RT} :\Leftrightarrow exec(tc, tm) \approx_{te} exec(tc, tm') \not\Rightarrow exec(tc, p) \approx_{te} exec(tc, p')$$

Summarizing, the test plan delta $\delta_{tp,tp'}$ is defined by the rules:

- $\forall tc \in tp \backslash ts'_{RT} : \{rem\ tc\} \subseteq \delta_{tp,tp'}$
- $\forall tc \in ts'_{RT} \backslash tp : \{add\ tc\} \subseteq \delta_{tp,tp'}$
- $\forall tc \in ts'_{N} : \{add\ tc\} \subseteq \delta_{tp,tp'}$

For further enhancements, additional information about previous test plans can be used for retest selections, e.g., how often a test case has been already executed (and failed).

Soundness of the Approach. For the soundness of the presented approach, we require the resulting test artifacts to be (1) *valid*, i.e., every test suite solely contains valid test cases, and (2) *complete*, i.e., guaranteeing complete test coverage of every product test model w.r.t. criterion C. Let $ta_1, ta_2, \ldots, ta_n$ be a collection of test artifacts incrementally built for a sequence of products $p_1, p_2, \ldots, p_n$ via deltas on test artifact as defined above.

Theorem 1. *(Validity of Product Test Suites)*
For product test suites ts_i of each ta_i, $1 \leqslant i \leqslant n$, $ts_{V_i} \subseteq TC(tm_i)$ holds.

Proof: By induction over the chain of regression delta applications. For $i = 1$, we assume soundness of the test case generator, i.e., $gen(tm_1, C) \subseteq TC(tm_1)$. For induction steps from i to $i+1$, (1) validity of ts_{V_i} follows from the induction hypothesis, and (2) validity of ts_{V_i} holds as obsolete and reusable test cases from ts_i are confirmed via the regression delta, and new test cases in ts_{i+1} are, again, delivered by the test case generator, i.e., $gen(tm_{i+1}, tg) \in TC(tm_{i+1})$.

Theorem 2. *(Completeness of Product Test Suites)*
For product test suites ts_i and test goals tg_i of each ta_i, $1 \leqslant i \leqslant n$, (1) $tg_i = C(tm_i)$ holds, and (2) ts_i satisfies C.

Proof Idea: Again, by induction over the chain of regression delta applications. For the correct incrementation (1) of test goals for more complex criteria, we rely on a sound implementation of the test goal selection function C, and (2) of test suites, we, again, assume soundness of the test case generator.

Moreover, the approach ensures every test case generated during the incremental testing process to be executed at least once as the set ts_N is always selected for the test plan. Our approach implicitly fulfills the *complete SPL test suite coverage* requirement proposed in [5]. In addition, it supports reasoning about the *reliability* of test plans: the impact on the fault detection efficiency of retesting selections between SPL products in comparison to complete product by product SPL testing is parameterizable via the change impact criterion under consideration.

5 Implementation and Evaluation

We developed a tool chain for the sample implementation of our incremental SPL testing approach. For the delta-oriented state machine SPL test modeling, we developed an Eclipse plug-in incorporating the Eclipse Modeling Framework. The tool supports the configuration of product variants based on a domain feature

model and the automated derivation of product test models. Those test models are imported into IBM Rational Rhapsody to apply the add-on ATG for automated test case generation and execution.

To evaluate our approach, we considered an SPL case study from the automotive domain, a simplified Body Comfort System (BCS) including numerous features like automatic power windows, human machine interface, alarm system, etc., comprising 11,616 valid product variants. We already obtained evaluation results from testing the BCS SPL in previous work for an SPL subset testing approach covering all valid feature pairs [19,16]. This allowed us to compare the results to those of our incremental testing technique w.r.t. gain in efficiency arising from test artifact reuse potentials. The original BCS SPL 150% state machine test model created for the MoSo-PoLiTe approach contains 105 states and 107 transitions comprising 26 input and 33 output events. We remodeled this test model to build a delta-oriented SPL test model including one core model and 40 delta modules.

For our experiments, we considered the *Model Element Coverage* criterion as supported by ATG. For covering every single product variant, an estimated amount of 743,424 test cases is required including multitudes of redundancies due to similarities among product variants. After applying MoSo-PoLiTe [19] we obtained 17 representative products ($P1 - P17$), thus reducing the number of test cases to 1,093 for testing this set product by product. To evaluate our incremental approach, we considered the same product subset and further added a core product ($P0$) as the starting point of the incremental SPL testing process. The results of the case study are shown in Fig. 4. Triangles denote the number of test cases generated and applied per product in the MoSo-PoLiTe approach [19]. In contrast, for the incremental SPL testing approach, diamonds denote the number of test cases to be newly generated for a product, and squares denote the number of test cases to be (re-)tested on that product. We focused our experiments on the reuse of test cases, whereat for the reuse of test results, we applied change impact analyses based on test model slicing [13]. Comparing our results to those of MoSo-PoLiTe, a significant reduction of the testing efforts concerning test case generation and execution was achieved, however ensuring the same degree of test model coverage. In particular, the average number of test

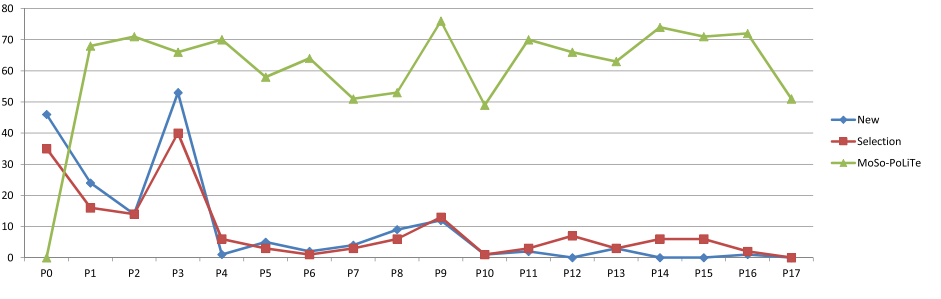


Fig. 4. Evaluation Results for the BCS SPL Case Study

cases generated and executed per product for MoSo-PoLiTe amounts 64, whereas our incremental approach solely requires an average number of 10 new test cases and 9 test cases selected for execution per product. In cases where the number of test case executions exceeds the number of test cases generated, existing test cases are selected for retesting. Most test cases are generated and selected for the first four products. As the number of existing test cases covering commonality between product variants continuously increases, a decreasing number of test cases is generated and executed for the remaining products.

Threats to Validity. The efficiency of the approach depends on the test case generator applied. The quality of the test suite of the initial product under test is particularly crucial for the subsequent iterations. However, this drawback is adherent to model-based testing in general, rather than an inconvenience of our approach. For the reuse of test cases, our current approach uses global repositories $\mathcal{S}$ and $\mathcal{T}$ to identify equality of traces by means of syntactical identity for testing equivalence $\approx_{te}$ and is restricted to deterministic behaviors. This is a rather strict requirement, but weakening this notion to more realistic testing equivalences [10] is far less efficiently decidable. Providing sound criteria for retest selection is, due to the black-box assumption of model-based testing, an open problem as common change impact analysis techniques are usually based on source code investigations [13]. For evaluating the impact of those criteria w.r.t. decreasing fault detection efficiency compared to complete product by product testing, further experiments, e.g., considering mutations, have to be performed.

6 Related Work

Various applications of behavioral models with variabilities to model-based SPL testing were proposed [18]. Cichos et al. propose a coverage-driven SPL test suite generation approach that is based on an annotative 150% test model [5]. Lochau et al. also use an annotative statechart test model for the detection and test coverage of interactions among feature artifacts [15,16]. Weissleder et al. define variabilities in state machines via annotations [32], whereas Szasz et al. add variable parts in Statecharts using composition operators [26].

Two research directions for reducing redundancies in product by product testing of SPLs currently exist: regression-based SPL testing and SPL subset selection heuristics. In [27,11], surveys on regression-based SPL testing approaches are presented mainly concentrating on empirical evaluations of different strategies. A first conceptional approach for regression-based SPL testing was, e.g., proposed by Batory et al. [31]. The authors propose an incremental refinement of test suites for a particular product variant under test w.r.t. the features composed into the product. Neto et al. [17] introduce an SPL testing framework, where regression testing decisions are performed on the basis of architectural similarities between product variants. Subset selection heuristics mainly use combinatorial testing heuristics to select representative products under test, e.g., considering features as combinatorial parameters [14]. For instance, Oster et al. cover pairwise feature combinations [20,21,19], whereas Perrouin et al. consider T-wise

combinations [22]. However, no strategies for test artifact reuse between products in those sub sets are mentioned. The notion of SPL test suites introduced in [5] is the closest related to our framework, but no application strategies of those test suites are provided. Furthermore, as our approach incrementally generates test cases on demand rather than symbolically in one pass, it is assumed to obey better scalability properties.

7 Conclusion

In this paper, we presented a novel MBT framework for incrementally deriving test suites for SPL product variants by applying principles of regression testing. As future work, we plan to further optimize the SPL testing process by (1) local minimizations of product test suites as well as global reductions on complete SPL test suites, and (2) delta-oriented, i.e., compositional test suite generation. For reliable fault detection efficiency, further theoretical considerations concerning appropriate test case reuse and retest selection criteria are to be considered.

References

1. Agrawal, H., Horgan, J.R., Krauser, E.W., London, S.A.: Incremental Regression Testing (1993)
2. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Model-Checking Tool for Families of Services. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 44–58. Springer, Heidelberg (2011)
3. Bruns, D., Klebanov, V., Schaefer, I.: Verification of Software Product Lines with Delta-Oriented Slicing. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 61–75. Springer, Heidelberg (2011)
4. Calder, M., Kolberg, M., Magill, E., Reiff-Marganiec, S.: Feature Interaction: A Critical Review and Considered Forecast. Computer Networks 41(1), 115–141 (2003)
5. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Model-Based Coverage-Driven Test Suite Generation for Software Product Lines. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 425–439. Springer, Heidelberg (2011)
6. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract Delta Modeling. Mathematical Structures in Computer Science (2011) (to appear)
7. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In: ICSE 2010 (2010)
8. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc. (2001)
9. Cohen, M., Dwyer, M., Shi, J.: Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In: ISSTA, pp. 129–139 (2007)
10. De Nicola, R.: Extensional Equivalence for Transition Systems. Acta Inf. 24, 211–237 (1987)
11. Engström, E., Skoglund, M., Runeson, P.: Empirical Evaluations of Regression Rest Selection Techniques. In: Proc. of ESEM 2008, pp. 22–31 (2008)

12. Fraser, G., Wotawa, F., Ammann, P.: Testing with Model Checkers: A Survey. Software Testing, Verification and Reliability 19(3), 215–261 (2009)
13. Gupta, R., Jean, M., Mary, H., Soffa, L.: An Approach to Regression Testing using Slicing. In: Proceedings of the Conference on Software Maintenance. pp. 299–308. IEEE Computer Society Press (1992)
14. Kim, C.H.P., Batory, D.S., Khurshid, S.: Reducing Combinatorics in Testing Product Lines. In: AOSD 2011, pp. 57–68. ACM (2011)
15. Lochau, M., Goltz, U.: Feature Interaction Aware Test Case Generation for Embedded Control Systems. ENTCS 264, 37–52 (2010)
16. Lochau, M., Oster, S., Goltz, U., Schürr, A.: Model-based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering. Software Quality Journal, 1–38 (2011)
17. da Mota Silveira Neto, P.A., do Carmo Machado, I., Cavalcanti, Y.C., de Almeida, E.S., Garcia, V.C., de Lemos Meira, S.R.: A regression testing approach for software product lines architectures. In: SBCARS 2010, pp. 41–50 (2010)
18. Olimpiew, E.M.: Model-Based Testing for Software Product Lines. Ph.D. thesis, George Mason University (2008)
19. Oster, S., Lochau, M., Zink, M., Grechanik, M.: Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations. In: FOSD 2011 (2011)
20. Oster, S., Markert, F., Ritter, P.: Automated Incremental Pairwise Testing of Software Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 196–210. Springer, Heidelberg (2010)
21. Oster, S., Zorcic, I., Markert, F., Lochau, M.: MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing. In: VAMOS 2011 (2011)
22. Perrouin, G., Sen, S., Klein, J., Le Traon, B.: Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In: ICST 2010, pp. 459–468 (2010)
23. Schaefer, I., Bettini, L., Damiani, F.: Compositional Type-Checking for Delta-oriented Programming. In: AOSD 2011. ACM Press (2011)
24. Schaefer, I., Gurov, D., Soleimanifard, S.: Compositional Algorithmic Verification of Software Product Lines. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 184–203. Springer, Heidelberg (2011)
25. Schaefer, I., Hähnle, R.: Formal Methods in Software Product Line Engineering. IEEE Computer 44(2), 82–85 (2011)
26. Szasz, N., Vilanova, P.: Statecharts and Variabilities. In: VAMOS 2008, pp. 131–140 (2008)
27. Tevanlinna, A., Taina, J., Kauppinen, R.: Product Family Testing: A Survey. ACM SIGSOFT Software Engineering Notes 29, 12–18 (2004)
28. Thiel, S., Hein, A.: Modeling and Using Product Line Variability in Automotive Systems. IEEE Software 19(4), 66–72 (2002)
29. Tretmans, J.: Testing Concurrent Systems: A Formal Approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 46–65. Springer, Heidelberg (1999)
30. Utting, M., Legeard, B.: Practical Model-Based Testing. A Tools Approach. M. Kaufmann (2007)
31. Uzuncaova, E., Khurshid, S., Batory, D.S.: Incremental test generation for software product lines. IEEE Trans. Software Eng. 36(3), 309–322 (2010)
32. Weißleder, S., Sokenou, D., Schlingloff, H.: Reusing State Machines for Automatic Test Generation in ProductLines. In: MoTiP 2008 (2008)

Conformance Relations for Labeled Event Structures

Hernán Ponce de León[1], Stefan Haar[1], and Delphine Longuet[2]

[1] INRIA and LSV, École Normale Supérieure de Cachan and CNRS, France
ponce@lsv.ens-cachan.fr, stefan.haar@inria.fr
[2] Univ. Paris-Sud, LRI UMR8623, Orsay, F-91405
longuet@lri.fr

Abstract. We propose a theoretical framework for testing concurrent systems from true concurrency models like Petri nets or networks of automata. The underlying model of computation of such formalisms are labeled event structures, which allow to represent concurrency explicitly. The activity of testing relies on the definition of a conformance relation that depends on the observable behaviors on the system under test, which is given for sequential systems by ioco type relations. However, these relations are not capable of capturing and exploiting concurrency of non sequential behavior. We study different conformance relations for labeled event structures, relying on different notions of observation, and investigate their properties and connections.

1 Introduction

This paper aims at laying the foundations of a systematic study of conformance relations for specifications that integrate features of concurrent behavior. Our ultimate goal is to lift conformance testing and its formal tools to the level of *event structure semantics*, where it currently focusses on *sequential* actions.

The Present State of the Art: A Sequential Picture. In fact, one of the most popular formalisms studied in conformance testing is that of *labeled transition systems* (LTS). A labeled transition system is a structure consisting of states and transitions labeled with actions from one state to another. This formalism is usually used for modeling the behavior of processes and as a semantical model for various formal languages such as CCS [1], CSP [2], SDL [3] and LOTOS [4]. Depending on the nature of the possible observations, different conformance relations have been defined for labeled transitions systems [5,6,7,8,9,10,11]; we will study how these lift to the "concurrent world". Several developments were built on the relation of *trace preorder* (trace inclusion). Firstly, it was refined into the *testing preorder*, that requires not only the inclusion of the implementation traces in those of the specification, but also that any action refused by the implementation should be refused by the specification [5,12]. A practical modification of the testing preorder was presented in [7], which proposed to base the observations on the traces of the specification only, leading to a weaker relation called **conf**. A further refinement concerns the inclusion of quiescent traces as

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 83–98, 2012.
© Springer-Verlag Berlin Heidelberg 2012

a conformance relation (e.g. Segala [10]). Moreover, Tretmans [11] proposed the **ioco** relation: each output produced by the implementation on specified stimuli should corresponde to the specified ones, and the implementation is authorised to reach a state where it cannot produce any output only if this is the case in the specification too.

Shifting to Concurrent Specifications. However, this framework does not yet very well support the testing from *concurrent* specifications, in which some pairs of events can be specified to occur in arbitrary order, or jointly. The exhaustive testing of all interleavings for a set of concurrent transitions is prohibitively slow, and is also conceptually inadequate; for both reasons, our aim is to provide a generalized framework that handles *true concurrency* in partially ordered models. The first major steps in this direction had been made in [13,14]: partially ordered patterns of input/output events were admitted as transition labels in a generalized I/O-automaton model, leading to a generalization of the basic notions and techniques of I/O-sequence based conformance testing. An important *practical* benefit of true-concurrency models here is an overall *complexity reduction*, despite the fact that checking partial orders requires in general multiple passes through the same labelled transition, so as to check for presence/absence of specified order relations between input and output events. In fact, if the system has n parallel and interacting processes, the length of checking sequences increases by a factor that is polynomial in n. At the same time, the overall size of the automaton model (in terms of the number of its states and transitions) shrinks *exponentially* if the concurrency between the processes is explicitly modeled. This feature indicates that with increasing size and distribution of SUTs in practice, it is computationally wise to seek alternatives for the direct sequential modeling approach. We add that true concurrency models are not only promising for practical reasons, but also are more adequate in reflecting the actual structure of distributed systems, and tend to be more accessible for designers and implementers, in particular if modularity can be exploited.

As indicated above, the work presented in [13,14] presents a first step towards a concurrency-based conformance theory. The partial-order I/O automata models developed there progress with respect to global state models such as multiport I/O-Automata by specifying dependence relations *across* processes explicitly, and allow to specify natural conditions that avoid e.g. controllability violations. However, the models of [13,14] still force us to maintain a sequential automaton as the system's skeleton, and to include synchronization constraints (typically: that all events specified in the pattern of a transition must be completed before any other transition can start), which limit both the application domain and the benefits from concurrency modeling. In other work in progress, we abandon automata altogether and focus on *Petri nets* as system models, which allows to completely discard any *global* synchronizations, and to exploit existing theory of concurrent behavior for devising testing strategies.

The present article provides the *semantic* viewpoint which accompanies and complements that shift in *systems* modeling. We use throughout a canonical semantic model for concurrent behavior, *labeled event structures*, providing a

unifying semantic framework for system models such as Petri nets, communicating automata, or process algebras; we abstract away from the particularities of system specification models, to focus entirely on behavioral relations.

The underlying mathematical structure for the system semantics is given by *event structures* in the sense of Winskel et al [15]. Mathematically speaking, they are particular partially ordered sets, in which order between events e and e' indicates precedence, and where any two events e and e' that are *not* ordered maybe either

- in *conflict*, meaning that in any evolution of the system in which e occurs, e' *cannot* occur; or
- *concurrent*, in which case they may occur in the same system run, without a temporal ordering, i.e. e may occur before e', after e', or simultaneously.

The state reached after some execution is represented by a *configuration* of the event structure, that is a conflict-free, history-closed set. The use of partial order semantics provides richer information and finer system comparisons than the interleaved view.

Overview. The paper is organized as follows: Section 2 gives the fundamental definitions of the semantic model of labeled event structures and Sect. 3 gives two definitions of observation of processes. Then, Sect. 4 introduces and studies conformance relations for *general* labeled event structures, and Sect. 5 specializes to *I/O systems* in which the label set is split into *input* and *output* labels, and introduces a new, true-concurrency-enabled **ioco** relation. Section 6 discusses the advantages and drawbacks of the conformance relations presented, and concludes.

2 Labeled Event Structures

We shall be using event structures following Winskel et al [15] to describe the dynamic behavior of a system. In this paper we will consider only prime event structures [16], a subset of the original model which is sufficient to describe concurrent models (therefore we will simply call them event structures), and we label their events with actions over a fixed alphabet L.

Definition 1 (Labeled event structure). *A* labeled event structure *over an alphabet L is a 4-tuple $\mathcal{E} = (E, \leq, \#, \lambda)$ such that*

- *E is a set of events,*
- *$\leq \subseteq E \times E$ is a partial order (called* causality*) satisfying the property of* finite causes*, i.e. $\forall e \in E : |\{e' \in E \mid e' \leq e\}| < \infty$,*
- *$\# \subseteq E \times E$ is an irreflexive symmetric relation (called* conflict*) satisfying the property of* conflict heredity*, i.e. $\forall e, e', e'' \in E : e \# e' \wedge e' \leq e'' \Rightarrow e \# e''$,*
- *$\lambda : E \to L$ is a labeling mapping.*

We denote the class of all labeled event structures over L by $\mathcal{LES}(L)$.

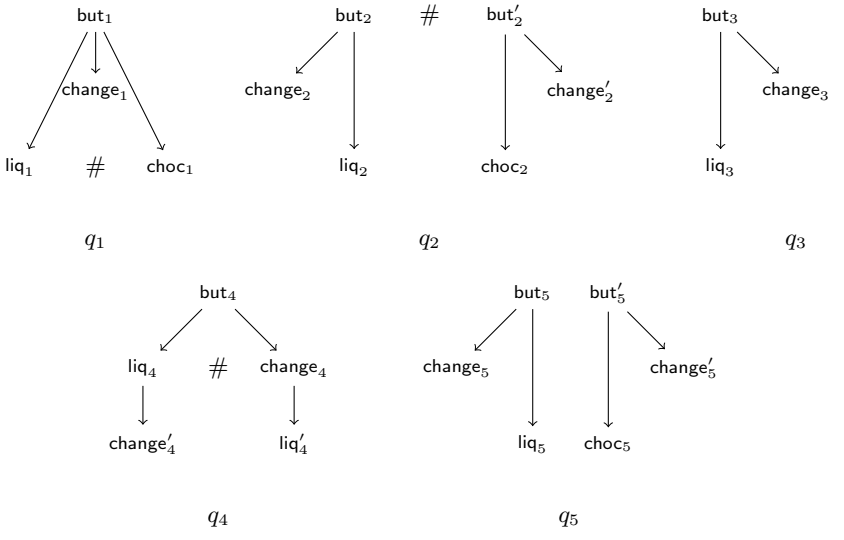


Fig. 1. Labeled event structures

Given a labeled event structure $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{LES}(L)$, two events $e, e' \in E$ are said to be *concurrent*, written e **co** e', iff neither $e \leq e'$ nor $e' \leq e$ nor $e \# e'$ hold.

Example 1. Fig. 1 presents different LES specifications of vending machines. The requirements are the following: when one pushes a button, the machine delivers chocolate bars or liquorices, and supplies change. We represent causality between events by the Hasse diagram of $\leq$, and direct conflict by $\#$. The labeling λ is such that $\lambda(e_i) = \lambda(e'_i) = e$.

Machine q_1 to q_4 have only one button while machine q_5 has two of them. In machines q_1 and q_2, a choice is made between supplying liquorice or chocolate after pressing the button, and concurrently, the machines supply change. The choice is made when the button is pushed in machine q_2 but internally after the pressing of the button in machine q_1. Machine q_3 only supplies liquorice and change concurrently while q_4 do both, but in a sequential way. We can press concurrently two different buttons in q_5, each of them producing liquorice or chocolate and supplying change.

A computation state of an event structure is called a configuration and is represented by the set of events that have occurred in the computation. If an event is present in a configuration, then so are all the events on which this event causally depends. Moreover, a configuration obviously does not contain conflicting events.

Definition 2 (Configuration). *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{LES}(L)$, *a* configuration *of* $\mathcal{E}$ *is a set of events* $C \subseteq E$ *such that*

- C *is causally closed:* $e \in C \Rightarrow \forall e' \leq e : e' \in C$, *and*
- C *is conflict-free:* $\forall e, e' \in C : \neg(e \# e')$.

The initial configuration of $\mathcal{E}$, denoted by $\bot_{\mathcal{E}}$, is the empty set of events. We denote the set of all the configurations of $\mathcal{E}$ by $\mathcal{C}(\mathcal{E})$.

Example 2. The configurations of the labeled event structure q_1 of Fig. 1 are $\bot_{q_1}$, $\{\mathsf{but}_1\}$, $\{\mathsf{but}_1, \mathsf{liq}_1\}$, $\{\mathsf{but}_1, \mathsf{change}_1\}$, $\{\mathsf{but}_1, \mathsf{choc}_1\}$, $\{\mathsf{but}_1, \mathsf{liq}_1, \mathsf{change}_1\}$, and $\{\mathsf{but}_1, \mathsf{choc}_1, \mathsf{change}_1\}$. It is worth noting that the configurations of q_1 and q_2 are different but their λ-images are the same.

A particular kind of event structures are those representing only sequential behaviors, i.e. without concurrency. A labeled event structure is called *sequential* iff there are no pairs of concurrent events in it: $\leq \cup \# = E \times E$. Sequential event structures can be seen as the computation trees obtained by unfolding labeled transition systems [17]. In Fig. 1, q_4 is a sequential labeled event structure.

3 Observing Event Structures

The next sections will present several conformance relations over labeled event structures. These relations are based on the chosen notion of observation of the system behavior in response to stimuli. The observations most studied in the literature for defining conformance relations are (execution) traces and refusals.

The definition of the notion of trace for a labeled event structure is not straightforward since it relies on the chosen semantics for concurrency [18]. The presence of explicit concurrency in a specification may be interpreted in several ways. In an early stage of specification, concurrency between events may be used as underspecification, leaving the choice of the actual order between events to the developper. The events specified as concurrent may then occur in any order in the implementation (maybe always the same one). In the specification of a distributed system however, concurrent events in the specification may be meant to remain concurrent in the implementation, because they are destined to occur in different components executed in parallel for instance.

We follow here two established semantics for concurrency, namely *interleaving* semantics where concurrent events may be executed in any order, and *partial order* semantics where no order is wanted or can be observed between concurrent events. In the first case, observing the behavior of the system action by action is sufficient since concurrent events will be observed sequentially. In the second case, several concurrent events may be observed together in one step, since they are not ordered. This leads to two definitions of traces for labeled event structures.

3.1 Single Action Observations

In this first setup, one considers *atomic* experiments on a system as single actions, and obtains an interleaving semantics for concurrency.

Definition 3. *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{LES}(L)$, $a \in L$, $\sigma = \sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_n \in L^+$ *and* $C, C' \in \mathcal{C}(\mathcal{E})$, *we define*

$$
\begin{aligned}
C \overset{a}{\Longrightarrow} C' &\triangleq \exists e \in E \backslash C : C' = C \cup \{e\} \text{ and } \lambda(e) = a \\
C \overset{a}{\Longrightarrow} &\triangleq \exists C' : C \overset{a}{\Longrightarrow} C' \\
C \overset{\sigma}{\Longrightarrow} C' &\triangleq \exists C_0, \ldots C_n : C = C_0 \overset{\sigma_1}{\Longrightarrow} C_1 \overset{\sigma_2}{\Longrightarrow} \ldots \overset{\sigma_n}{\Longrightarrow} C_n = C' \\
C \overset{\sigma}{\Longrightarrow} &\triangleq \exists C' : C \overset{\sigma}{\Longrightarrow} C'
\end{aligned}
$$

One goes from a configuration to another by performing only one action at a time, thus leading to a trace semantics where an execution is a sequence of single actions (obviously, the empty sequence leads to the same configuration, i.e. $C \overset{\epsilon}{\Longrightarrow} C$).[1] Possible observations of the system behavior are captured by the following definition.

Definition 4. *Let* $\mathcal{E} \in \mathcal{LES}(L)$, $A \subseteq L$, $\sigma \in L^*$, $S \subseteq \mathcal{C}(\mathcal{E})$ *and* $C, C' \in \mathcal{C}(\mathcal{E})$, *we define*

- $\mathit{traces}(\mathcal{E}) \triangleq \{\sigma \in L^* \mid \bot_{\mathcal{E}} \overset{\sigma}{\Longrightarrow}\}$
- C ***after*** $\sigma \triangleq \{C' \mid C \overset{\sigma}{\Longrightarrow} C'\}$
- C ***refuses*** $A \triangleq \forall a \in A : C \overset{a}{\not\Longrightarrow}$
- S ***refuses*** $A \triangleq \exists C \in S : C$ ***refuses*** A

The set traces($\mathcal{E}$) contains the full action sequences of $\mathcal{E}$, while C **after** σ contains the possible configurations reached from C when σ was observed. Refusal of an action set A means the impossibility of executing any transition with a label in A. In the next section we will use **refuses** together with **after**, and as the system can reach several configurations after σ, we extend **refuses** to sets of configurations.

Example 3. With this interleaving semantics, the traces of machine q_3 in Fig. 1 are $\{\epsilon, \mathsf{but}, \mathsf{but} \cdot \mathsf{liq}, \mathsf{but} \cdot \mathsf{change}, \mathsf{but} \cdot \mathsf{liq} \cdot \mathsf{change}, \mathsf{but} \cdot \mathsf{change} \cdot \mathsf{liq}\}$, since concurrent events may be seen in any order. Therefore, machines q_3 and q_4 have the same traces. Due to the inheritance of conflict, machines q_1 and q_2 also have the same traces since after but, one can perform liq and change in any order, or choc and change in any order.

Concerning refusals, one can see that machine q_1 cannot produce chocolate after producing liquorice, i.e. ($\bot_{q_1}$ **after** $\mathsf{but} \cdot \mathsf{liq}$) **refuses** $\{\mathsf{choc}\}$. Note that S **refuses** A is false when S is empty, therefore ($\bot_{q_3}$ **after** $\mathsf{but} \cdot \mathsf{choc}$) **refuses** $\emptyset$ is false since $\mathsf{but} \cdot \mathsf{choc}$ is not a trace of q_3.

3.2 Partially Ordered Observations

Since the event structure model is capable of explicitly distinguishing the causal structure of the model, it is natural to expect the observations of machines q_3 and q_4 of Fig. 1 to be different; in fact, actions liq and change are independent in q_3, but they are causally ordered in q_4.

[1] We denote by ϵ the empty word and by $\_ \cdot \_$ the concatenation of words in L^*.

In order to distinguish such behaviors, the notion of trace must keep concurrency explicit, i.e. must preserve the partial order of the events of an execution. We first recall the notion of labeled partial order, then we lift Def. 3 and Def. 4 to the partial order setting.

Definition 5 (Labeled partial order). *A* labeled partial order *over L is a tuple $\omega = (E_\omega, \leq_\omega, \lambda_\omega)$, where*

- *$(E_\omega, \leq_\omega)$ is a partial order, and*
- *$\lambda_\omega : E_\omega \to L$ is a labeling mapping.*

We denote the class of all labeled partial orders over L by $\mathcal{LPO}(L)$.

Labeled partial orders will be used to represent observations of executions containing concurrent events. Moreover, we will need the notion of *labeled concurrent set* to represent a set of concurrent events: we say that $\alpha \in \mathcal{LPO}(L)$ is a labeled concurrent set over L iff $<_\alpha = \emptyset$, and denote the class of such objects by $\mathcal{CO}(L)$.

In partial order semantics, a step of an execution from a given configuration may be a single action or a set of actions performed concurrently. This leads to the following definitions. We indicate by a subscript or superscript π the relations and sets to be interpreted in the partial order semantics.

Definition 6. *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{LES}(L)$, $\alpha = (E_\alpha, \leq_\alpha, \lambda_\alpha) \in \mathcal{CO}(L)$, $\omega = (E_\omega, \leq_\omega, \lambda_\omega) \in \mathcal{LPO}(L)$ and $C, C' \in \mathcal{C}(\mathcal{E})$, we define*

$$C \overset{\alpha}{\Longrightarrow}_\pi C' \triangleq \exists A \subseteq E \backslash C : C' = C \cup A, A = E_\alpha, <_{|A \times A} = \emptyset \text{ and } \lambda_{|A} = \lambda_\alpha$$

$$C \overset{\alpha}{\Longrightarrow}_\pi \triangleq \exists C' : C \overset{\alpha}{\Longrightarrow}_\pi C'$$

$$C \overset{\omega}{\Longrightarrow}_\pi C' \triangleq \exists A \subseteq E \backslash C : C' = C \cup A, A = E_\omega, \leq_{|A \times A} = \leq_\omega \text{ and } \lambda_{|A} = \lambda_\omega$$

$$C \overset{\omega}{\Longrightarrow}_\pi \triangleq \exists C' : C \overset{\omega}{\Longrightarrow}_\pi C'$$

The ability of making concurrent execution explicit is the key advantage in using partial order semantics.

Definition 7. *Let $\mathcal{E} \in \mathcal{LES}(L)$, $A \subseteq \mathcal{CO}(L)$, $\omega \in \mathcal{LPO}(L)$, $S \subseteq \mathcal{C}(\mathcal{E})$ and $C, C' \in \mathcal{C}(\mathcal{E})$, we define*

- *$traces_\pi(\mathcal{E}) \triangleq \{\omega \in \mathcal{LPO}(L) \mid \perp_\mathcal{E} \overset{\omega}{\Longrightarrow}_\pi\}$*
- *C **after**$_\pi$ $\omega \triangleq \{C' \mid C \overset{\omega}{\Longrightarrow}_\pi C'\}$*
- *C **refuses**$_\pi$ $A \triangleq \forall \alpha \in A : C \overset{\alpha}{\not\Longrightarrow}_\pi$*
- *S **refuses**$_\pi$ $A \triangleq \exists C \in S : C$ **refuses**$_\pi$ A*

Example 4. We consider labeled partial orders of Fig 2. We can observe liq and change concurrently after but in machine q_1 of Fig. 1, so we have $\omega_1 \in \text{traces}_\pi(q_1)$, but we cannot observe them concurrently in q_4 because the system only allows to see them ordered, thus $\omega_1 \notin \text{traces}_\pi(q_4)$.

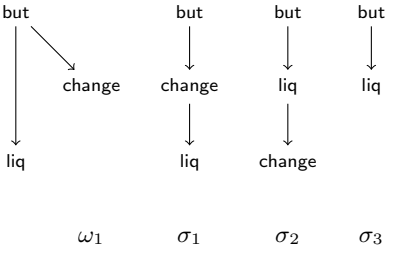


Fig. 2. Partially ordered observations vs. sequential observations

In the other way round, causality between liq and change is desired in q_4 but cannot be observed or is not wanted in q_1, so $\sigma_1 \in \text{traces}_\pi(q_4)$ and $\sigma_1 \notin \text{traces}_\pi(q_1)$.

Prefixes are also allowed, we have for instance $\sigma_3 \in \text{traces}_\pi(q_1)$.

Note that in a sequential labeled event structure $\mathcal{E}$, since there is no concurrency, we have $\text{traces}_\pi(\mathcal{E}) = \text{traces}(\mathcal{E})$. Since $L \subseteq \mathcal{CO}(L)$ and $L^* \subseteq \mathcal{LPO}(L)$, Def. 6 and Def. 7 are strict generalizations of Def. 3 and Def. 4.

Remark 1. By abuse of notation, we will use indistinctly σ_1 and $\mathsf{but} \cdot \mathsf{change} \cdot \mathsf{liq}$ or ω_1 and $\mathsf{but} \cdot (\mathsf{liq}\ \mathbf{co}\ \mathsf{change})$.

4 Conformance Relations for Concurrent Systems

The objective of this paper is to propose a generalization of the **ioco** relation [11]. We first propose generalizations of the conformance relations defined in the literature for systems with symmetric interactions, i.e. where inputs and outputs are not differentiated. We follow the presentation and notations adopted in [11].

The first relation proposed in the literature, called *trace preorder*, is based on the inclusion of the executions of the system under test in those allowed by the specification. The intuition is that an implementation i should not exhibit any unspecified sequence of actions, i.e. not present in the specification s.

Definition 8 (Trace preorder for single action observation). *Let* $i, s \in \mathcal{LES}(L)$*, then*

$$i \leq_{tr} s \Leftrightarrow traces(i) \subseteq traces(s)$$

Example 5. With the interleaving semantics, the traces of q_1 and q_2 are the same, and we have $q_1 \leq_{\text{tr}} q_2$ and $q_2 \leq_{\text{tr}} q_1$. Analogously, $q_3 \leq_{\text{tr}} q_4$ and $q_4 \leq_{\text{tr}} q_3$. The systems q_3 and q_4 implement part of what is specified in q_1 and q_2, therefore $q_3, q_4 \leq_{\text{tr}} q_1, q_2$.

This relation is very similar to the trace preorder for labeled transition systems since it is based on the observation of sequences of actions. On the contrary, the adaptation of the trace preorder to labeled event structures with the partial order semantics leads to a new conformance relation.

Definition 9 (Trace preorder for partially ordered observation). *Let* $i, s \in \mathcal{LES}(L)$*, then*

$$i \leq^{\pi}_{tr} s \Leftrightarrow traces_{\pi}(i) \subseteq traces_{\pi}(s)$$

Example 6. Since $\text{traces}_{\pi}(q_1) = \text{traces}_{\pi}(q_2)$, we have $q_1 \leq^{\pi}_{\text{tr}} q_2$ and $q_2 \leq^{\pi}_{\text{tr}} q_1$. We also have $q_3 \leq^{\pi}_{\text{tr}} q_1$ and $q_3 \leq^{\pi}_{\text{tr}} q_2$, but $q_4 \not\leq^{\pi}_{\text{tr}} q_3$ because $\leq^{\pi}_{\text{tr}}$ requires concurrent events in the specification to remain truly concurrent in the implementation and does not accept any order between them as it is the case of $\leq_{\text{tr}}$. The traces of q_2 are observable in q_5, but q_5 accepts more behaviors since the second button can still be pushed after the first one was, so $q_2 \leq^{\pi}_{\text{tr}} q_5$ but $q_5 \not\leq^{\pi}_{\text{tr}} q_2$.

With both relations, we have that q_2 correctly implements q_1, but q_1 specifies that after pressing a button the user has a choice between liquorice and chocolate, while q_2 may refuse one of these. The reason of this is that both $\leq_{\text{tr}}$ and $\leq^{\pi}_{\text{tr}}$ only consider sequences (resp. partial order) of actions as observations, and not whether conflicts are resolved internally by the machine, or externally by the environment.

Therefore, we propose a stronger relation to refine $\leq^{\pi}_{\text{tr}}$. In addition to requiring that any execution of the implementation is allowed in the specification, we require that any time the implementation refuses to perform a new action, that action cannot be performed in the specification either. This new conformance relation generalizes the testing preorder of [5].[2]

Definition 10 (Testing preorder for partially ordered observation). *Let* $i, s \in \mathcal{LES}(L)$*, then*

$$i \leq^{\pi}_{te} s \Leftrightarrow \forall \omega \in \mathcal{LPO}(L), A \subseteq \mathcal{CO}(L) : \\ \perp_i \textbf{\textit{after}}_{\pi}\ \omega\ \textbf{\textit{refuses}}_{\pi}\ A \Rightarrow \perp_s \textbf{\textit{after}}_{\pi}\ \omega\ \textbf{\textit{refuses}}_{\pi}\ A$$

Example 7. Consider again Fig. 1, we have $q_1 \leq^{\pi}_{\text{te}} q_2$: there are no trace ω and no set of events A such that $\perp_{q_1}$ **after**$_{\pi}$ ω **refuses**$_{\pi}$ A and $\neg(\perp_{q_2}$ **after**$_{\pi}$ ω **refuses**$_{\pi}$ $A)$. However, $q_2, q_3 \not\leq^{\pi}_{\text{te}} q_1$, since for instance $\perp_{q_2}$ **after**$_{\pi}$ but **refuses**$_{\pi}$ $\{\mathsf{choc}\}$ and $\neg(\perp_{q_1}$ **after**$_{\pi}$ but **refuses**$_{\pi}$ $\{\mathsf{choc}\})$. The button can be pressed twice concurrently in q_5, but not in q_2, so $(\perp_{q_2}$ **after** but) **refuses** $\{\mathsf{but}\}$ and $q_2 \not\leq^{\pi}_{\text{te}} q_5$.

Note that the relation $\leq^{\pi}_{\text{te}}$ does not allow extra traces in the implementation. In fact, $q_1 \not\leq^{\pi}_{\text{te}} q_3$ since $\perp_{q_1}$ **after**$_{\pi}$ but $\cdot$ choc **refuses**$_{\pi}$ $\emptyset$, yet $\perp_{q_3}$ **after**$_{\pi}$ but $\cdot$ choc $= \emptyset$, hence $\neg(\perp_{q_3}$ **after**$_{\pi}$ but $\cdot$ choc **refuses**$_{\pi}$ $\emptyset)$. As but **co** but is a trace of q_5 yet not one of q_2, we have $q_5 \not\leq^{\pi}_{\text{te}} q_2$. As this relation checks for trace inclusion, it still differentiates between q_3 and q_4, i.e. $q_4 \not\leq^{\pi}_{\text{te}} q_3$

We propose a weaker relation **conf**$_{\pi}$ restricting all the traces to only the ones contained in the specification. This relation requires that the implementation

[2] From now on, we will present the conformance relations for the partial order semantics only. The corresponding conformance relations for the interleaving semantics can be straightforwardly deduced.

does what it has to do, not that it does not what it is not allowed to do. It allows underspecification, i.e. that only a subset of the functionalities of the actual system are specified.

Definition 11 (Relation conf for partially ordered observation). *Let $i, s \in \mathcal{LES}(L)$, then*

$$i \ \boldsymbol{conf}_\pi \ s \Leftrightarrow \forall \omega \in traces_\pi(s), A \subseteq \mathcal{CO}(L) : \\ \bot_i \ \boldsymbol{after}_\pi \ \omega \ \boldsymbol{refuses}_\pi \ A \Rightarrow \bot_s \ \boldsymbol{after}_\pi \ \omega \ \boldsymbol{refuses}_\pi \ A$$

Example 8. We saw in Ex. 7 that $q_1 \leq^\pi_{\text{te}} q_2$, and since $\mathbf{conf}_\pi$ considers the traces of q_2 only, we have $q_1 \ \mathbf{conf}_\pi \ q_2$.

Since the relation $\mathbf{conf}_\pi$ is based on the traces of the specification only, it allows extra traces in the implementation. So even if $q_1 \not\leq^\pi_{\text{te}} q_3$, we have $q_1 \ \mathbf{conf}_\pi \ q_3$. In the same way, $q_5 \not\leq^\pi_{\text{te}} q_2$ but $q_5 \ \mathbf{conf}_\pi \ q_2$.

However, if all traces of the implementation are also traces of the specification, then the testing preorder is equivalent to $\mathbf{conf}_\pi$. We have $\neg(q_2 \ \mathbf{conf}_\pi \ q_1)$ since $q_2 \not\leq^\pi_{\text{te}} q_1$ and $\text{traces}_\pi(q_1) = \text{traces}_\pi(q_2)$. Moreover, we have $\neg(q_3 \ \mathbf{conf}_\pi \ q_1)$ since $q_3 \not\leq^\pi_{\text{te}} q_1$ and $\text{traces}_\pi(q_3) \subseteq \text{traces}_\pi(q_1)$, and also $\neg(q_2 \ \mathbf{conf}_\pi \ q_5)$ since $q_2 \not\leq^\pi_{\text{te}} q_5$ and $\text{traces}_\pi(q_2) \subseteq \text{traces}_\pi(q_5)$.

The following result relates the different implementation relations in the partial order semantics.

Proposition 1

1. *$\leq^\pi_{tr}$ and $\leq^\pi_{te}$ are preorders; $\boldsymbol{conf}_\pi$ is reflexive.*
2. *$\leq^\pi_{te} = \leq^\pi_{tr} \cap \boldsymbol{conf}_\pi$*

Proof. Point 1 being obvious, we only show point 2, by proving that the inclusion holds in both directions. Suppose $i \not\leq^\pi_{\text{tr}} s$, then there exists $\omega \in \mathcal{LPO}(L)$ such that $\bot_i \stackrel{\omega}{\Longrightarrow}_\pi$, but $\bot_s \not\stackrel{\omega}{\Longrightarrow}_\pi$, thus $\bot_s \ \mathbf{after}_\pi \ \omega = \emptyset$ and $\neg(\bot_s \ \mathbf{after}_\pi \ \omega \ \mathbf{refuses}_\pi \ \emptyset)$ while $\bot_i \ \mathbf{after}_\pi \ \omega \ \mathbf{refuses}_\pi \ \emptyset$, $i \not\leq^\pi_{\text{te}} s$ and finally $\leq^\pi_{\text{te}} \subseteq \leq^\pi_{\text{tr}}$. As $\mathbf{conf}_\pi$ is a restriction of $\leq^\pi_{\text{te}}$ to the traces of s, it is easy to prove that $\leq^\pi_{\text{te}} \subseteq \mathbf{conf}_\pi$.

Suppose $i \not\leq^\pi_{\text{te}} s$, then there exist $\omega \in \mathcal{LPO}(L), A \subseteq \mathcal{CO}(L)$ such that $\bot_i \ \mathbf{after}_\pi \ \omega \ \mathbf{refuses}_\pi \ A$ and $\neg(\bot_s \ \mathbf{after}_\pi \ \omega \ \mathbf{refuses}_\pi \ A)$. If $\omega \in \text{traces}_\pi(s)$ we have that $\neg(i \ \mathbf{conf}_\pi \ s)$. If $\omega \notin \text{traces}_\pi(s)$, we know by $\bot_i \ \mathbf{after}_\pi \ \omega \ \mathbf{refuses}_\pi \ A$ that $\omega \in \text{traces}_\pi(i)$ and therefore $i \not\leq^\pi_{\text{tr}} s$.

Example 9. In Fig. 3 we can see that $p_2 \ \mathbf{conf}_\pi \ p_1$. If we denote the set $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ by L, we have $\bot_{p_2} \ \mathbf{after}_\pi \ \mathsf{a} \ \mathbf{refuses}_\pi \ \{\mathsf{a}, \mathsf{b}\}$ and $\bot_{p_1} \ \mathbf{after}_\pi \ \mathsf{a} \ \mathbf{refuses}_\pi \ \{\mathsf{a}, \mathsf{b}\}$; we also have $\bot_{p_2} \ \mathbf{after}_\pi \ \mathsf{a} \cdot \mathsf{c} \ \mathbf{refuses}_\pi \ L$ and $\bot_{p_1} \ \mathbf{after}_\pi \ \mathsf{a} \cdot \mathsf{c} \ \mathbf{refuses}_\pi \ L$; finally $\bot_{p_2} \ \mathbf{after}_\pi \ \mathsf{a} \cdot \mathsf{b} \ \mathbf{refuses}_\pi \ S$ is false for any set S. We can see that $p_3 \ \mathbf{conf}_\pi \ p_2$ since p_3 is p_2 with an additional branch. Nevertheless we do not have $p_3 \ \mathbf{conf}_\pi \ p_1$: we have $\bot_{p_3} \ \mathbf{after}_\pi \ \mathsf{a} \cdot \mathsf{b} \ \mathbf{refuses}_\pi \ \{\mathsf{c}\}$ but $\neg(\bot_{p_1} \ \mathbf{after}_\pi \ \mathsf{a} \cdot \mathsf{b} \ \mathbf{refuses}_\pi \ \{\mathsf{c}\})$. This shows that $\mathbf{conf}_\pi$ is not transitive.

The interested reader may verify that the relations presented in Ex. 6, Ex. 7 and Ex. 8 satisfy Prop. 1.

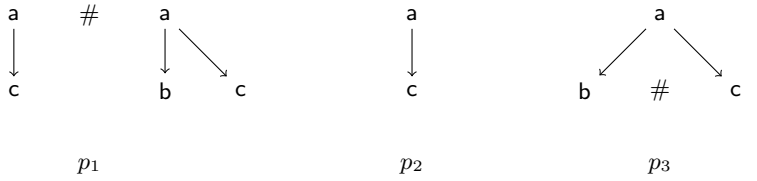


Fig. 3. The $\mathbf{conf}_\pi$ relation is not transitive

5 Conformance Relations for Input/Output Concurrent Systems

As usual when testing reactive systems, we want to distinguish between the controllable and observable actions of the system under test. We extend the model of labeled event structures to make a distinction between input actions (proposed by the environment) and output actions (produced by the system) of the system, leading input-output labeled event structures (IOLES).

Definition 12 (Input-output labeled event structure). *An* input-output labeled event structure *is a labeled event structure over the alphabet* $L = L_i \uplus L_o$*. The class of input-output labeled event structures over* L *is denoted by* $\mathcal{IOLES}(L)$*.*

As usual, let $?a$ denote an input action in L_i and $!a$ an output action in L_o. Examples of IOLES are given in Fig. 4.

5.1 The ioco Relation for the Interleaving Semantics

We first present the definition of the **ioco** conformance relation for labeled event structures with the interleaving semantics.

The **ioco** relation requires that, after a trace of the specification, the outputs produced by the implementation are authorized by the specification, but also the absence of outputs. A state where the system cannot produce outputs is called quiescent in the labeled transition system framework. Similarly, in the labeled event structure framework, a configuration where the system cannot produce outputs will be called quiescent.

Definition 13 (Quiescent configuration for single action observation). *Let* $\mathcal{E} \in \mathcal{IOLES}(L)$*. A configuration* $C \in \mathcal{C}(\mathcal{E})$ *is* quiescent *iff* $\forall a \in L_o : C \stackrel{a}{\Longrightarrow}$*.*

In our framework, a system is in a quiescent configuration if it is waiting for an input from the environment or it deadlocks.

As it is now standard in the LTS framework, we assume that quiescent configurations are observable by a special output action $\delta \in L_o$. The event corresponding to a δ action should be unique in the given configuration (1), and it should be in conflict with all the other possible events from the same configuration (2).

Additionally, since the δ action captures the notion of not observing anything but the absence of reaction of the system, observing δ should not change the behavior of the system (3). We denote by $\Delta_{\mathcal{E}}$ an IOLES $\mathcal{E}$ enriched by δ such that these properties hold. Formally, we assume that every quiescent configuration $C \in \mathcal{C}(\Delta_{\mathcal{E}})$ has the following properties:

$$(\exists!\ e_\delta \in E \setminus C : \lambda(e_\delta) = \delta \wedge C \xRightarrow{\delta}) \tag{1}$$

$$\wedge\ (\forall e' \in E \setminus (C \cup \{e_\delta\}), C \xRightarrow{\lambda(e')}\ \Rightarrow\ e_\delta \# e') \tag{2}$$

$$\wedge\ (\forall \sigma \in L^* : C \xRightarrow{\sigma}\ \Rightarrow\ C \cup \{e_\delta\} \xRightarrow{\sigma}) \tag{3}$$

In the interleaving semantics, the way to observe the outputs of the system in response to stimuli is the same as in the LTS framework: the set of possible outputs from a given configuration is the set of every single possible output.

Definition 14 (Expected outputs for single action observation). *Let $\mathcal{E} \in \mathcal{IOLES}(L)$ and $S \subseteq \mathcal{C}(\mathcal{E})$, then*

$$out(S) \triangleq \bigcup_{C \in S} \{a \in L_o \mid C \xRightarrow{a}\}$$

We obtain the following adaptation of the **ioco** conformance relation to the labeled event structure framework with the interleaving semantics: for any trace of the specification enriched with δ actions, every single output produced by the implementation after this trace (including δ) is authorised by the specification.

Definition 15 (ioco for single action observation). *Let $i, s \in \mathcal{IOLES}(L)$, then*

$$i\ \textbf{\textit{ioco}}\ s \Leftrightarrow \forall \sigma \in traces(\Delta_s) : out(\bot_{\Delta_i}\ \textbf{\textit{after}}\ \sigma) \subseteq out(\bot_{\Delta_s}\ \textbf{\textit{after}}\ \sigma)$$

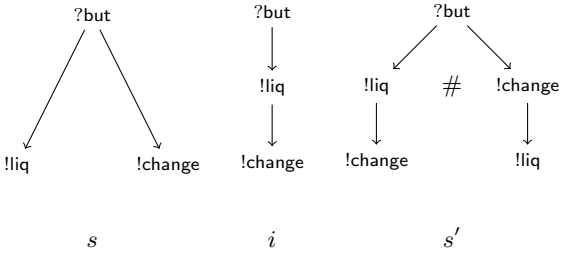


Fig. 4. Difference between **ioco** and **co-ioco**

Example 10. We consider the labeled event structures of Fig. 4, s and s' being two specifications and i a possible implementation.

As we saw in previous examples, with the interleaving semantics, s and s' have the same traces: $traces(s) = traces(s') = \{\epsilon, ?but, ?but \cdot !liq, ?but \cdot !change, ?but \cdot !liq \cdot !change, ?but \cdot !change \cdot !liq\}$ and we obtain:[3]

σ	$out(\bot_{\Delta_s}$ **after** $\sigma)$	$out(\bot_{\Delta'_s}$ **after** $\sigma)$	$out(\bot_{\Delta_i}$ **after** $\sigma)$
ϵ	$\{\delta\}$	$\{\delta\}$	$\{\delta\}$
?but	{!liq, !change}	{!liq, !change}	{!liq}
?but · !liq	{!change}	{!change}	{!change}
?but · !change	{!liq}	{!liq}	$\emptyset$
?but · !liq · !change	$\{\delta\}$	$\{\delta\}$	$\{\delta\}$
?but · !change · !liq	$\{\delta\}$	$\{\delta\}$	$\emptyset$

We conclude that both i **ioco** s and i **ioco** s' hold. Note that the fact of observing the empty set is different from observing δ. Observing δ after executing a trace σ means that the system performed σ and reached a quiescent configuration, while observing the empty set formally denotes the fact of not being able to execute the experiment σ as in the case of i for the trace ?but · !change.

5.2 The ioco Relation for the Partial Order Semantics: co-ioco

We define a new conformance relation **co-ioco** for labeled event structures with the partial order semantics.

We first need to define the notion of quiescent configuration in this semantics: here, the possible actions in a given configuration are not only single actions but also sets of concurrent events.

Definition 16 (Quiescent configuration for partially ordered observation). *Let* $\mathcal{E} \in \mathcal{IOLES}(L)$. *A* quiescent *configuration* $C \in \mathcal{C}(\mathcal{E})$ *is such that* $\forall \alpha \in \mathcal{CO}(L_o) : C \stackrel{\alpha}{\not\Longrightarrow}_\pi$.

We also need to redefine the properties that the enhancement of an IOLES by δ actions must verify. The conflict with other possible events in the given configuration expressed by property (2) extends to sets of concurrent events. Property (3) naturally extends to partial order semantics, considering partially ordered trace instead of sequential ones. Therefore, denoting by $\Delta_\mathcal{E}$ the enhancement by δ actions of an IOLES $\mathcal{E}$, we assume that every quiescent configuration $C \in \mathcal{C}(\Delta_\mathcal{E})$ has the following properties:

$$
\begin{aligned}
&(\exists!\ e_\delta \in E \setminus C : \lambda(e_\delta) = \delta \wedge C \stackrel{\delta}{\Longrightarrow}_\pi) \\
\wedge\ &(\forall \alpha = (E_\alpha, \leq_\alpha, \lambda_\alpha) \in \mathcal{CO}(L) : C \stackrel{\alpha}{\Longrightarrow}_\pi \ \Rightarrow\ \forall e' \in E_\alpha : e_\delta \# e') \\
\wedge\ &(\forall \omega \in \mathcal{LPO}(L) : C \stackrel{\omega}{\Longrightarrow}_\pi \ \Rightarrow\ C \cup \{e_\delta\} \stackrel{\omega}{\Longrightarrow}_\pi)
\end{aligned}
$$

In the partial order semantics, the outputs of the system under test in response to stimuli may be single outputs as well as sets of concurrent outputs. We need

[3] To lighten the examples, we consider the traces of s and i only, and not all the traces of Δ_s and Δ_i, since it makes no difference in these cases.

any set of concurrent outputs to be entirely produced by the system under test, so we define the set of expected outputs from a set of configurations S as the set of maximal sets of concurrent outputs.

Definition 17 (Expected outputs for partially ordered observation). *Let $\mathcal{E} \in \mathcal{IOLES}(L)$ and $S \subseteq \mathcal{C}(\mathcal{E})$, then*

$$out_\pi(S) \triangleq \bigcup_{C \in S} \{\alpha \in \mathcal{CO}(L_o) \mid C \stackrel{\alpha}{\Longrightarrow}_\pi \text{ and } E_\alpha \text{ is maximal w.r.t} \subseteq\}$$

We obtain the following formulation of the **co-ioco** conformance relation, in the case of the partial order semantics: for any partially ordered trace of the specification enriched with δ actions, the sets of concurrent outputs produced by the implementation after this trace (including δ) are among those authorised by the specification.

Definition 18 (co-ioco). *Let $i, s \in \mathcal{IOLES}(L)$, then*

$$i \ \textbf{\textit{co-ioco}} \ s \Leftrightarrow \forall \omega \in traces_\pi(\Delta_s) : out_\pi(\perp_{\Delta_i} \textbf{\textit{after}}_\pi \ \omega) \subseteq out_\pi(\perp_{\Delta_s} \textbf{\textit{after}}_\pi \ \omega)$$

Example 11. We have $\text{traces}_\pi(s) = \{\epsilon, ?\text{but}, ?\text{but} \cdot !\text{liq}, ?\text{but} \cdot !\text{change}, ?\text{but} \cdot (!\text{liq } \textbf{co } !\text{change})\}$ and $\text{traces}_\pi(s') = \{\epsilon, ?\text{but}, ?\text{but}\cdot!\text{liq}, ?\text{but} \cdot !\text{change}, ?\text{but} \cdot !\text{liq} \cdot !\text{change}, ?\text{but} \cdot !\text{change} \cdot !\text{liq}\}$ and we can observe:

ω	$\text{out}_\pi(\perp_{\Delta_s} \textbf{after}_\pi \ \omega)$	$\text{out}_\pi(\perp_{\Delta_{s'}} \textbf{after}_\pi \ \omega)$	$\text{out}_\pi(\perp_{\Delta_i} \textbf{after}_\pi \ \omega)$
ϵ	$\{\delta\}$	$\{\delta\}$	$\{\delta\}$
?but	{!liq **co** !change}	{!liq, !change}	{!liq}
?but · !liq	{!change}	{!change}	{!change}
?but · !change	{!liq}	{!liq}	$\emptyset$
?but · (!liq **co** !change)	$\{\delta\}$	$\emptyset$	$\emptyset$
?but · !liq · !change	$\emptyset$	$\{\delta\}$	$\{\delta\}$
?but · !change · !liq	$\emptyset$	$\{\delta\}$	$\emptyset$

We conclude that i **co-ioco** s' but $\neg(i$ **co-ioco** $s)$. This is due to the fact that the **co-ioco** relation requires any concurrent set of outputs depending on the same input to remain concurrent.

6 Conclusion, Discussion and Future Work

We have laid several cornerstones for conformance testing under true concurrency. Four well-established conformance relations over labeled transition systems [11] (trace preorder, testing preorder, **conf** and **ioco**) have been extended to concurrency-enabled relations over labeled event structures, and illustrated in several examples. The next steps will encompass test case generation and the formalization of adequate centralized and distributed test architectures.

With the interleaving semantics, the relations we obtain boil down to the same relations defined for LTS, since they focus on sequences of actions. The only advantage of using labeled event structures as a specification formalism

for testing remains in the conciseness of the concurrent model with respect to a sequential one. As far as testing is concerned, the benefit is low since every interleaving has to be tested.

By contrast, under the partial order semantics, the relations we obtain allow to distinguish explicitly implementations where concurrent actions are implemented concurrently, from those where they are interleaved, i.e. implemented sequentially. Therefore, these relations will be of interest when designing distributed systems, since the natural concurrency between actions that are performed in parallel by different processes can be taken into account. In particular, the fact of being unable to control or observe the order between actions taking place on different processes will not be considered as an impediment for testing. This opens interesting perspectives for a distributed test architecture where testers would scarcely have to synchronize. If the specification allows it, one could even think of a local test architecture where testers are completely independent from each other.

It should be noted that the **co-ioco** relation allows for further refinement that we do not discuss here. As defined in Def. 7, this relation deals with concurrent inputs and concurrent outputs in the same way: it requires concurrency to be preserved by the implementation. This is exactly what should be the case when the specification requires these concurrent inputs to be processed by different entities, and the concurrent outputs to be issued from different processes. That is, the *distribution* or attribution of events assigned to concurrent processes is then part of the specification, and conformance requires the implementation to have exactly this distribution. An analogous idea is captured in the concurrency-preserving bisimulation relation developed in [19]. It is natural to look also for means of dealing with concurrency in specifications in a different way, namely with a *"don't care"*-type approach: that is, for some events a and b that are specified as concurrent, one may accept implementations that order a before b OR that order b before a (provided of course they conform otherwise to the specification). Care must then be taken to distinguish different types of dependency (output on output? input on output? output on input?) and to study which kind of dependency may be added without compromising required properties. A discussion of some of these aspects, plus the question when dependencies might be *dropped* in the implementation, can be found in [14]. In the present context, such generalization requires the modification of Def. 6: the requirement $\leq_{|A\times A} = \leq_\omega$ is then replaced by an inclusion relation, with additional constraints such as preservation of immediate input-output-orders etc. The discussion and development of these points, which are at the heart of work in progress, would have taken us too far afield here.

Acknowledgment. The research related here was funded by the DIGITEO/ DIM-LSC project TECSTES, convention DIGITEO Number 2011-052D - TECSTES.

References

1. Milner, R.: Communication and concurrency. PHI Series in computer science. Prentice Hall (1989)
2. Hoare, T.: Communicating Sequential Processes. Prentice-Hall (1985)
3. ITU-TS: Recommendation Z.100: Specification and Description Language (2002)
4. Brinksma, E., Scollo, G., Steenbergen, C.: Lotos specifications, their implementations and their tests. In: Linn, R.J., Uyar, M.U. (eds.) Conformance testing methodologies and architectures for OSI protocols, pp. 468–479. IEEE Computer Society Press (1995)
5. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34, 83–133 (1984)
6. Abramsky, S.: Observation equivalence as a testing equivalence. Theoretical Computer Science 53, 225–241 (1987)
7. Brinksma, E.: A theory for the derivation of tests. In: Protocol Specification Testing and Verification VIII, pp. 63–74. North-Holland (1988)
8. Phillips, I.: Refusal testing. Theoretical Computer Science 50, 241–284 (1987)
9. Langerak, R.: A testing theory for LOTOS using deadlock detection. In: Protocol Specification, Testing and Verification IX, pp. 87–98. North-Holland (1990)
10. Segala, R.: Quiescence, fairness, testing, and the notion of implementation. Information and Computation 138(2), 194–210 (1997)
11. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
12. De Nicola, R.: Extensional equivalences for transition systems. Acta Informatica 24(2), 211–237 (1987)
13. von Bochmann, G., Haar, S., Jard, C., Jourdan, G.-V.: Testing Systems Specified as Partial Order Input/Output Automata. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 169–183. Springer, Heidelberg (2008)
14. Haar, S., Jard, C., Jourdan, G.-V.: Testing Input/Output Partial Order Automata. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 171–185. Springer, Heidelberg (2007)
15. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science 13, 85–108 (1981)
16. Winskel, G.: Event Structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)
17. Nielsen, M., Sassone, V., Winskel, G.: Relationships Between Models of Concurrency. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1993. LNCS, vol. 803, pp. 425–476. Springer, Heidelberg (1994)
18. Aceto, L., De Nicola, R., Fantechi, A.: Testing Equivalences for Event Structures. In: Venturini Zilli, M. (ed.) Mathematical Models for the Semantics of Parallelism. LNCS, vol. 280, pp. 1–20. Springer, Heidelberg (1987)
19. Balaguer, S., Chatain, T., Haar, S.: A concurrency-preserving translation from time Petri nets to networks of timed automata. In: International Symposium on Temporal Representation and Reasoning, pp. 77–84. IEEE Computer Society Press (2010)

Test Generation from Recursive Tiles Systems

Sébastien Chédor[1], Thierry Jéron[2], and Christophe Morvan[3]

[1] Université Rennes I
[2] INRIA Rennes - Bretagne Atlantique
[3] Université Paris-Est, Marne-La-Vallée, France
{sebastien.chedor,thierry.jeron}@inria.fr,
christophe.morvan@univ-paris-est.fr

Abstract. In this paper we explore test generation for *Recursive Tiles Systems* (RTS) in the framework of the classical **ioco** testing theory. The RTS model allows the description of reactive systems with recursion, and is very similar to other models like Pushdown Automata, Hyperedge Replacement Grammars or Recursive State Machines. We first present an off-line test generation algorithm for *Weighted* RTS, a determinizable sub-class of RTS, and second, an on-line test generation algorithm for the full RTS model. Both algorithms use test purposes to guide test selection through targeted behaviours.

1 Introduction and Motivation

Conformance testing is the problem of checking by test experiments that a black-box implementation behaves correctly with respect to its specification. It is well known that testing is the most used validation technique to assess the quality of software systems, and represents the largest part in the cost of software development. Automatising is thus required in order to improve the cost and quality of the testing process. In particular, it is undoubtedly interesting to automate the test generation phase from specifications of the system. Formal model-based testing aims at resolving this problem by the formal description of testing artefacts (specifications, possible implementations, test cases) by mathematical models, formal definitions of conformance, the execution of tests and their verdicts, and the proof of some essential properties of test cases relating verdicts produced by test executions on implementations and conformance of these implementations with respect to their specifications. The **ioco** conformance theory introduced in [13] is a well established framework for the formal modelling of conformance testing for Input/Output Transition Systems (IOLTSs). Test generation algorithms and tools have been designed for this model [9,12] and for more general models whose semantics can be expressed in the form of infinite state IOLTSs [10,8].

In this paper, we are interested in test generation for reactive recursive programs, like the one in Fig 1. There already exist several ways to define recursive behaviours: pushdown automata (PDA), recursive state machines [1], regulars graphs, defined by functional (or deterministic) hyperedge replacement grammars (HR-grammars), [7,3]. Each of these models has its merits and flaws: PDA are classical, and well understood; recursive state machines are equally expressive and more visual as a model; HR-grammars are a visual model which characterizes the same languages but enables to model systems having states of infinite degree. Furthermore, recent results define classes of such

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 99–114, 2012.
© Springer-Verlag Berlin Heidelberg 2012

```
static void main(String [] args){
   try{
      // Block 1 (input)
      int k =in.readInt();
      comp(k);
      // Block 2 (output)
      System.out.println("Done");
   }
   catch (Exception e){
      // Block 3 (output)
      System.out.println(e.getMessage());
   }
}
void comp (int x){
   // Block 4 (input)
   int res =1;
   boolean cont=in.readBoolean();
   if (cont){
      if (x==0)throw new Exception("An error occurred");
      // Block 5 (internal)
      res=x*comp(x-1);
      // Block 6 (output)
      System.out.println("Some text");
      return res;
   }
   else {
      // Block 7 (output)
      system.out.println("You stopped");
      return res;
   }
}
```

Fig. 1. A recursive program

systems which may be determinized [5], which is of interest for test generation. The HR-grammars, on the other hand, are very technical to define. Here we try to get the best of both worlds: we use HR-grammars presented as tiling systems, called RTS (RTS). Such systems are mostly finite sets of finite LTS with frontiers, crossing the frontier corresponds to entering a new copy of one of the finite LTS. The semantics of an RTS is then an infinite state LTS. Hopefully for such models (co)-reachability which is essential for test generation using test purposes is decidable. Also determinization is possible for the class of *Weighted* RTS, which permits to design off-line test generation algorithms for this sub-class. For the whole class of RTS however determinization is impossible, but on-line test generation is still possible as subset construction is performed along finite executions.

To the best of our knowledge test generation for recursive programs has been seldom considered in the literature. The only work we are aware of is [6] which considers a model of deterministic PDA with inputs/outputs (IOPDS) and generate test cases in the same model. The present work can be seen as an extension of this, where non-determinism is taken into account.

Contribution and Outline: The contribution of the paper is as follows. Section 2 recalls the main ingredients of the **ioco** testing theory for IOLTSs. In Section 3, we define the model of RTS for the description of recursive reactive programs, give its semantics in terms of an infinite state IOLTS obtained by recursive expansion of tiles. In Section 4, in the **ioco** framework, we propose an off-line test selection algorithm guided by test purposes for *Weighted* RTSs, a determinizable sub-class of RTSs, and prove essential properties of generated test cases. Furthermore in Section 5, we design an on-line test generation algorithm for the full RTS model, also using test purposes for test selection.

2 Conformance Testing Theory for IOLTS

This section recalls the **ioco** testing theory for the model of Input/Output Labelled Transition Systems that will serve as a basis for test generation from RTS. We first give a non-standard definition of IOLTS and introduce notations and basic operations, then review the **ioco** testing theory.

Definition 1. *An IOLTS (Input Output Labelled Transition System) is a tuple* $\mathcal{M} = (Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \Lambda_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \mathcal{C}_{\mathcal{M}}, \mathtt{init}_{\mathcal{M}})$ *where* $Q_{\mathcal{M}}$ *is a set of* states*;* $\Sigma_{\mathcal{M}}$ *is the* alphabet of actions *partitioned into a set of inputs* $\Sigma^{?}_{\mathcal{M}}$*, a set of outputs* $\Sigma^{!}_{\mathcal{M}}$ *and a set of internal actions* $\Sigma^{\tau}_{\mathcal{M}}$ *and we denote by* $\Sigma^{\circ}_{\mathcal{M}} \triangleq \Sigma^{?}_{\mathcal{M}} \cup \Sigma^{!}_{\mathcal{M}}$ *the set of* visible actions [1]*;* $\Lambda_{\mathcal{M}}$ *is a set of* colours *with* $\mathtt{init}_{\mathcal{M}} \in \Lambda_{\mathcal{M}}$ *a colour for initial states;* $\rightarrow_{\mathcal{M}} \subseteq Q_{\mathcal{M}} \times \Sigma_{\mathcal{M}} \times Q_{\mathcal{M}}$ *is the* transition relation*;* $\mathcal{C}_{\mathcal{M}} \subseteq Q_{\mathcal{M}} \times \Lambda_{\mathcal{M}}$ *is a relation between colours and states.*

In this non-standard definition of IOLTSs, colours are used to mark states by the relation $\mathcal{C}_{\mathcal{M}}$. For a colour $\lambda \in \Lambda_{\mathcal{M}}$, $\mathcal{C}_{\mathcal{M}}(\lambda) \triangleq \{q \in Q_{\mathcal{M}} \mid (q,\lambda) \in \mathcal{C}_{\mathcal{M}}\}$ and $\overline{\mathcal{C}_{\mathcal{M}}}(\lambda) \triangleq \{q \in Q_{\mathcal{M}} \mid (q,\lambda) \notin \mathcal{C}_{\mathcal{M}}\}$ denote respectively the sets of states coloured and not coloured by λ. In particular, $\mathcal{C}_{\mathcal{M}}(\mathtt{init}_{\mathcal{M}})$ defines the set of initial states.

We write $q \xrightarrow{a}_{\mathcal{M}} q'$ for $(q,a,q') \in \rightarrow_{\mathcal{M}}$ and $q \xrightarrow{a}_{\mathcal{M}}$ for $\exists q' : q \xrightarrow{a}_{\mathcal{M}} q'$. This notation is generalized to sequences of actions, and for $w = \mu_1 \dots \mu_n \in (\Sigma_{\mathcal{M}})^*$, we note $q \xrightarrow{w}_{\mathcal{M}} q'$ for $\exists q_0, \dots, q_n : q = q_0 \xrightarrow{\mu_1}_{\mathcal{M}} q_1 \xrightarrow{\mu_2}_{\mathcal{M}} \cdots \xrightarrow{\mu_n}_{\mathcal{M}} q_n = q'$.

For $X \subseteq Q_{\mathcal{M}}$ a subset of states and $\Sigma' \subseteq \Sigma$ a sub-alphabet, we denote by $post_{\mathcal{M}}(\Sigma', X) = \{q' \in Q_{\mathcal{M}} \mid \exists q \in X, \exists \mu \in \Sigma' : q \xrightarrow{\mu}_{\mathcal{M}} q'\}$ the set of direct successors of a state in X by an action in Σ', and $pre_{\mathcal{M}}(\Sigma', X) = \{q \in Q_{\mathcal{M}} \mid \exists q' \in X, \exists \mu \in \Sigma' : q \xrightarrow{\mu}_{\mathcal{M}} q'\}$ the set of direct predecessors of X by a transition in Σ'. The set of states *reachable* from $P \subseteq Q_{\mathcal{M}}$ by actions in Σ' is $reach_{\mathcal{M}}(\Sigma', P) \triangleq \mathrm{lfp}(\lambda X.P \cup post_{\mathcal{M}}(\Sigma', X))$ where lfp is the least fixed point operator. Similarly, the set of states *coreachable* from $P \subseteq Q_{\mathcal{M}}$ (*i.e.* the set of states from which P is reachable) is $coreach_{\mathcal{M}}(\Sigma', P) \triangleq \mathrm{lfp}(\lambda X.P \cup pre_{\mathcal{M}}(\Sigma', X))$. We will also write $reach_{\mathcal{M}}(\Sigma', \lambda)$ for $reach_{\mathcal{M}}(\Sigma', \mathcal{C}_{\mathcal{M}}(\lambda))$ and $coreach_{\mathcal{M}}(\Sigma', \lambda)$ for $coreach_{\mathcal{M}}(\Sigma', \mathcal{C}_{\mathcal{M}}(\lambda))$.

$\Gamma_{\mathcal{M}}(q) \triangleq \{\mu \in \Sigma_{\mathcal{M}} \mid q \xrightarrow{\mu}_{\mathcal{M}}\}$ denotes the subset of actions enabled in q and respectively, $Out_{\mathcal{M}}(q) \triangleq \Gamma_{\mathcal{M}}(q) \cap \Sigma^{!}_{\mathcal{M}}$ and $In_{\mathcal{M}}(q) \triangleq \Gamma_{\mathcal{M}}(q) \cap \Sigma^{?}_{\mathcal{M}}$ denote the set of outputs (resp. inputs) enabled in q. For $P \subseteq Q_{\mathcal{M}}$, $Out_{\mathcal{M}}(P) \triangleq \bigcup_{q \in P} Out_{\mathcal{M}}(q)$ and $In_{\mathcal{M}}(P) \triangleq \bigcup_{q \in P} In_{\mathcal{M}}(q)$.

[1] In the examples, for readability reasons, we write $?a$ for an input $a \in \Sigma^{?}_{\mathcal{M}}$, $!x$ for an output $x \in \Sigma^{!}_{\mathcal{M}}$ and internal actions have no sign.

Visible behaviours of $\mathcal{M}$ are defined by the relation $\Longrightarrow_{\mathcal{M}} \in Q_{\mathcal{M}} \times (\{\epsilon\} \cup \Sigma^o_{\mathcal{M}}) \times Q_{\mathcal{M}}$ as follows: $q \overset{\varepsilon}{\Longrightarrow}_{\mathcal{M}} q' \triangleq q = q'$ or $q \overset{\tau_1.\tau_2 \cdots \tau_n}{\longrightarrow}_{\mathcal{M}}{}^{*} q'$ and for $a \in \Sigma^o_{\mathcal{M}}$, $q \overset{a}{\Longrightarrow}_{\mathcal{M}} q' \triangleq \exists q_1, q_2 : q \overset{\varepsilon}{\Longrightarrow}_{\mathcal{M}} q_1 \overset{a}{\longrightarrow}_{\mathcal{M}} q_2 \overset{\varepsilon}{\Longrightarrow}_{\mathcal{M}} q'$. For $\sigma = a_1 \cdots a_n \in (\Sigma^o_{\mathcal{M}})^*$ a sequence of visible actions, $q \overset{\sigma}{\Longrightarrow}_{\mathcal{M}} q'$ stands for $\exists q_0, \ldots, q_n : q = q_0 \overset{a_1}{\Longrightarrow}_{\mathcal{M}} q_1 \cdots \overset{a_n}{\Longrightarrow}_{\mathcal{M}} q_n = q'$ and $q \overset{\sigma}{\Longrightarrow}_{\mathcal{M}}$ for $\exists\, q' : q \overset{\sigma}{\Longrightarrow}_{\mathcal{M}} q'$. We denote $q \ \texttt{after}\ \sigma \triangleq \{q' \in Q \mid q \overset{\sigma}{\Longrightarrow}_{\mathcal{M}} q'\}$ for the set of states in which one can be after observing σ starting from q and for $P \subseteq Q_{\mathcal{M}}$, $P \ \texttt{after}\ \sigma \triangleq \bigcup_{q \in P} q \ \texttt{after}\ \sigma$. $\mathsf{Traces}(q) \triangleq \{\sigma \in (\Sigma^o_{\mathcal{M}})^* \mid q \overset{\sigma}{\Longrightarrow}_{\mathcal{M}}\}$ denotes the set of sequences of visible actions that may be observed from q and $\mathsf{Traces}(\mathcal{M}) \triangleq \bigcup_{q_0 \in \mathcal{C}(\texttt{init}_{\mathcal{M}})} \mathsf{Traces}(q_0)$. $\mathsf{Traces}_P(\mathcal{M}) = \{\sigma \in (\Sigma^o_{\mathcal{M}})^* \mid (\mathcal{C}_{\mathcal{M}}(\texttt{init}_{\mathcal{M}})\ \texttt{after}\, \sigma) \cap P \neq \emptyset\}$ denotes the set of traces of sequences accepted in P.

$\mathcal{M}$ is *input-complete* if in each state all inputs are enabled, possibly after internal actions, i.e. $\forall q \in Q_{\mathcal{M}}, \forall a \in \Sigma^?_{\mathcal{M}}, q \overset{a}{\Longrightarrow}_{\mathcal{M}}$. $\mathcal{M}$ is *complete in a state* q if any action is enabled in q: $\forall q \in Q_{\mathcal{M}}, \Gamma(q) = \Sigma_{\mathcal{M}}$. $\mathcal{M}$ is *complete* if it is complete in all states.

An IOLTS $\mathcal{M}$ is deterministic if $|\mathcal{C}(\texttt{init}_{\mathcal{M}})| = 1$ (*i.e.* there is a unique initial state) and $\forall q \in Q_{\mathcal{M}}, \forall a \in \Sigma^o_{\mathcal{M}}, |q \ \texttt{after}\, a| \leq 1$, where $|.|$ is the cardinal of a set.

From an IOLTS $\mathcal{M}$, one can define a deterministic IOLTS $\mathcal{D}(\mathcal{M})$ with same traces as $\mathcal{M}$ as follows: $\mathcal{D}(\mathcal{M}) = (2^{Q_{\mathcal{M}}}, \Sigma^o_{\mathcal{M}}, \Lambda_{\mathcal{D}}, \rightarrow_{\mathcal{D}}, \mathcal{C}_{\mathcal{D}}, \texttt{init}_{\mathcal{D}})$ where for $P, P' \in 2^{Q_{\mathcal{M}}}$, $a \in \Sigma^o_{\mathcal{M}}$, $P \overset{a}{\longrightarrow}_{\mathcal{D}} P' \iff P' = P \ \texttt{after}\, a$, and $init_{\mathcal{D}} \in \Lambda_{\mathcal{D}}$ is the colour for the singleton state $\mathcal{C}_{\mathcal{D}}(init_{\mathcal{D}}) = \mathcal{C}_{\mathcal{M}}(\texttt{init}_{\mathcal{M}})\ \texttt{after}\, \varepsilon \in 2^{Q_{\mathcal{M}}}$. One can define other colours in $\Lambda_{\mathcal{D}}$ and, depending on the objective, the colouring $\mathcal{C}_{\mathcal{D}}$ may be defined according to $\Lambda_{\mathcal{M}}$ and $\mathcal{C}_{\mathcal{M}}$. For example, if $f \in \Lambda_{\mathcal{M}}$ defines marked states in $\mathcal{M}$, one may define a colour $F \in \Lambda_{\mathcal{D}}$ for $\mathcal{D}(\mathcal{M})$ such that $\mathsf{Traces}_{\mathcal{C}_{\mathcal{M}}(f)}(\mathcal{M}) = \mathsf{Traces}_{\mathcal{C}_{\mathcal{D}}(F)}(\mathcal{D}(M))$ simply by colouring by F the states in $s \in 2^{Q_{\mathcal{M}}}$ such that $\mathcal{C}(f)$ intersects s, *i.e.* at least one state in s is marked by f. Observe that the definition of $\mathcal{D}(\mathcal{M})$ is not always effective. However, it is the case whenever $\mathcal{M}$ is a finite state IOLTS. Even when it is effective, such a transformation may lead to an exponential blow-up. Often, for efficiency reasons, the full construction of $\mathcal{D}(\mathcal{M})$ is avoided, and on-the-fly paths are computed (visiting only a limited part of the powerset).

Synchronous product of IOLTS: One may define a product of two IOLTS such that sequences of actions in the product are the sequences of actions of both IOLTS:

Definition 2. *Let $\mathcal{M}_i = (Q_{\mathcal{M}_i}, \Sigma, \Lambda_{\mathcal{M}_i}, \rightarrow_{\mathcal{M}_i}, \mathcal{C}_{\mathcal{M}_i}, \texttt{init}_{\mathcal{M}_i})$, $i = 1, 2$ be two IOLTSs with same alphabet Σ. Their synchronous product $\mathcal{M}_1 \times M_2$ is the IOLTS $\mathcal{P} = (Q_{\mathcal{P}}, \Sigma_{\mathcal{P}}, \Lambda_{\mathcal{P}}, \rightarrow_{\mathcal{P}}, \mathcal{C}_{\mathcal{P}}, \texttt{init}_{\mathcal{P}})$ such that $Q_{\mathcal{P}} \triangleq Q_{\mathcal{M}_1} \times Q_{\mathcal{M}_2}$, and $\forall (q_1, q_2), (q'_1, q'_2) \in Q_{\mathcal{P}}$, $(q_1, q_2) \overset{a}{\longrightarrow}_{\mathcal{P}} (q'_1, q'_2) \triangleq q_1 \overset{a}{\longrightarrow}_{\mathcal{M}_1} q'_1 \wedge q_2 \overset{a}{\longrightarrow}_{\mathcal{M}_2} q'_2$. We define $\Lambda_{\mathcal{P}} \triangleq \Lambda_{\mathcal{M}} \times \Lambda_{\mathcal{M}'}$, in particular $\texttt{init}_{\mathcal{P}} \triangleq (\texttt{init}_{\mathcal{M}_1}, \texttt{init}_{\mathcal{M}_2})$, and for any $(\lambda_1, \lambda_2) \in \Lambda_{\mathcal{P}}$ the colouring relation is defined by $\mathcal{C}_{\mathcal{P}}((\lambda_1, \lambda_2)) \triangleq \mathcal{C}_{\mathcal{M}_1}(\lambda_1) \times \mathcal{C}_{\mathcal{M}_2}(\lambda_2)$.*

Specification and Implementation: In the **ioco** testing framework, we assume that the behaviour of the specification is modelled by IOLTS $\mathcal{S} = (Q_{\mathcal{S}}, \Sigma_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}}, \texttt{init}_{\mathcal{S}})$. The implementation under test is a black box system with same observable interface as the specification. In order to formalize conformance, it is usually assumed that the implementation behaviour can be modelled by an (unknown) input-complete IOLTS $\mathcal{I} = (Q_{\mathcal{I}}, \Sigma_{\mathcal{I}}, \Lambda_{\mathcal{I}}, \rightarrow_{\mathcal{I}}, \texttt{init}_{\mathcal{I}})$ with $\Sigma_{\mathcal{I}} = \Sigma^?_{\mathcal{I}} \cup \Sigma^!_{\mathcal{I}} \cup \Sigma^\tau_{\mathcal{I}}$ and $\Sigma^?_{\mathcal{I}} = \Sigma^?_{\mathcal{S}}$ and $\Sigma^!_{\mathcal{I}} = \Sigma^!_{\mathcal{S}}$.

Quiescence: It is current practice that tests observe traces of the implementation, and also absence of reaction (quiescence) using *timers*. Tests should then distinguish between quiescences allowed or not by the specification. Several kinds of quiescence may happen in an IOLTS: a state q is *output quiescent* if it is only waiting for inputs from the environment, *i.e.* $\Gamma(q) \subseteq \Sigma^?_{\mathcal{M}}$, (a *deadlock i.e.* $\Gamma(q) = \emptyset$ is a special case of output quiescence), and a *livelock* if an infinite sequence of internal actions is enabled, *i.e.* $\forall n \in \mathbb{N}, \exists \sigma \in (\Sigma^\tau_{\mathcal{M}})^n, q \xrightarrow{\sigma}_{\mathcal{M}}$ [2]. We note $quiescent(q)$ if q is either an output quiescence or in a livelock. From an IOLTS $\mathcal{M}$ one can build a new IOLTS $\Delta(\mathcal{M})$ where quiescence is made explicit by a new output δ:

Definition 3. *Let $\mathcal{M} = (Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \Lambda_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \mathcal{C}_{\mathcal{M}}, \texttt{init}_{\mathcal{M}})$ be an IOLTS, $\Delta(\mathcal{M})$ is the IOLTS $\Delta(\mathcal{M}) = (Q_{\mathcal{M}}, \Sigma_{\Delta(\mathcal{M})}, \Lambda_{\mathcal{M}}, \rightarrow_{\Delta(\mathcal{M})}, \mathcal{C}_{\mathcal{M}}, \texttt{init}_{\mathcal{M}})$ where $\Sigma_{\Delta(\mathcal{M})} = \Sigma_{\mathcal{M}} \cup \{\delta\}$ with $\delta \in \Sigma^!_{\Delta(\mathcal{M})}$ (δ is considered as an output, observable by the environment), and $\rightarrow_{\Delta(\mathcal{M})} = \rightarrow_{\mathcal{M}} \cup \{(q, \delta, q) \mid q \in quiescent(M)\}$ is obtained from $\rightarrow_M$ by adding δ loops for each quiescent state q.*

In the sequel, we note $\Sigma^{!\delta}_{\mathcal{M}}$ for $\Sigma^!_{\mathcal{M}} \cup \{\delta\}$ and $\Sigma^{\circ\delta}_{\mathcal{M}}$ for $\Sigma^\circ_{\mathcal{M}} \cup \{\delta\}$. The traces of $\Delta(\mathcal{M})$ denoted by $\mathsf{STraces}(\mathcal{M})$ are called the *suspension traces* of $\mathcal{M}$. They represent the visible behaviour of $\mathcal{M}$, including quiescence and are the basis for the definition of the **ioco** conformance relation.

Conformance Relation: In the **ioco** formal conformance theory [13], the implementation $\mathcal{I}$ conforms to its specification $\mathcal{S}$ if after any suspension trace σ of $\mathcal{S}$ the implementation $\mathcal{I}$ exhibits only outputs and quiescences that are specified in S. Formally:

Definition 4. *Let $\mathcal{S}$ be an IOLTS and $\mathcal{I}$ be an input-complete IOLTS with same visible alphabet ($\Sigma^?_{\mathcal{S}} = \Sigma^?_{\mathcal{I}}$ and $\Sigma^!_{\mathcal{S}} = \Sigma^!_{\mathcal{I}}$),*
*$\mathcal{I}$ **ioco** $\mathcal{S} \triangleq \forall \sigma \in \mathsf{STraces}(\mathcal{S}), Out(\Delta(\mathcal{I})\ \texttt{after}\ \sigma) \subseteq Out(\Delta(\mathcal{S})\ \texttt{after}\ \sigma)$.*

It can be proved [10] that $\mathcal{I}$ **ioco** $\mathcal{S} \iff \mathsf{STraces}(\mathcal{I}) \cap \mathsf{MinFTraces}(\mathcal{S}) = \emptyset$, where $\mathsf{MinFTraces}(\mathcal{S}) \triangleq \mathsf{STraces}(\mathcal{S}).\Sigma^!_{\mathcal{S}} \setminus \mathsf{STraces}(\mathcal{S})$ is the set of non-conformant suspension traces, minimal for the prefix ordering.

Test Cases, Test Suites, Properties: The behaviour of a test case is modelled by an IOLTS equipped with colours representing verdicts assigned to executions.

Definition 5. *A* test case *for $\mathcal{S}$ is a deterministic and input-complete IOLTS $\mathcal{TC} = (Q_{\mathcal{TC}}, \Sigma_{\mathcal{TC}}, \Lambda_{\mathcal{TC}}, \rightarrow_{\mathcal{TC}}, \mathcal{C}_{\mathcal{TC}}, \texttt{init}_{\mathcal{TC}})$ where $Pass, Fail, Inc, None \in \Lambda_{\mathcal{TC}}$ are colours characterising verdicts. $\mathcal{C}_{\mathcal{TC}}(Pass)$, $\mathcal{C}_{\mathcal{TC}}(Fail)$, $\mathcal{C}_{\mathcal{TC}}(Inc)$ and $\mathcal{C}_{\mathcal{TC}}(None)$ forms a partition of $Q_{\mathcal{TC}}$. Its alphabet is $\Sigma_{\mathcal{TC}} = \Sigma^?_{\mathcal{TC}} \cup \Sigma^!_{\mathcal{TC}}$ where $\Sigma^?_{\mathcal{TC}} = \Sigma^{!\delta}_{\mathcal{S}}$ and $\Sigma^!_{\mathcal{TC}} = \Sigma^?_{\mathcal{S}}$ (outputs of $\mathcal{TC}$ are inputs of $\mathcal{S}$ and vice versa).* A test suite *is a set of test cases.*

The execution of a test case $\mathcal{TC}$ against an implementation $\mathcal{I}$ can be modelled by the parallel composition $\mathcal{TC} \| \mathcal{I}$ where common actions (inputs, outputs and quiescence) are synchronized. The effect is to intersect sets of suspension traces ($\mathsf{Traces}(\mathcal{TC} \| \mathcal{I}) =$

[2] We here consider both loops or internal actions and divergences, *i.e.* infinite sequences of internal actions traversing an infinite number of states.

$\mathsf{STraces}(\Delta(\mathcal{I})) \cap \mathsf{Traces}(\mathcal{TC}))$. Consequently, the possible failure of a test case on an implementation is defined as $\mathcal{TC}$ *fail* $\mathcal{I} \triangleq \mathsf{STraces}(\Delta(\mathcal{I})) \cap \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}) = \emptyset$. Similar definitions can be given for *pass* and *inconc* relative to *Pass* and *Inc*.

We now define some properties that should be satisfied by test cases in order to correctly relate conformance to rejection by a test case:

Definition 6. *Let $\mathcal{S}$ be a specification, and $\mathcal{TS}$ a test suite for $\mathcal{S}$.*
$\mathcal{TS}$ is sound *if no test case may reject a conformant implementation:*
$\forall \mathcal{I}, \forall \mathcal{TC} \in \mathcal{TS}, \mathcal{I}$ **ioco** $\mathcal{S} \implies \neg(\mathcal{TC}$ *fail* $\mathcal{I})$.
$\mathcal{TS}$ is exhaustive *if it rejects all non-conformant implementations:*
$\forall \mathcal{I}, \neg(\mathcal{I}$ **ioco** $S) \implies \exists \mathcal{TC} \in \mathcal{TS}, \mathcal{TC}$ *fail* $\mathcal{I}$.
It is complete *if it is both sound and exhaustive.*
$\mathcal{TS}$ is strict *if it detects non-conformance as soon as they happen:*
$\forall \mathcal{I}, \forall \mathcal{TC} \in \mathcal{TS}, \neg(\mathcal{TC} \| \mathcal{I}$ **ioco** $\mathcal{S}) \Rightarrow \mathcal{TC}$ *fail* $\mathcal{I}$.

The following characterisations derived from [10] are very convenient to prove those properties on generated test suites:

Proposition 1. *Let $\mathcal{TS}$ be a test suite for $\mathcal{S}$,*
$\mathcal{TS}$ is sound if $\bigcup_{\mathcal{TC} \in \mathcal{TS}} \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}) \subseteq \mathsf{MinFTraces}(\mathcal{S}).\Sigma_{\mathcal{S}}^*$,
$\mathcal{TS}$ is exhaustive if $\bigcup_{\mathcal{TC} \in \mathcal{TS}} \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}) \supseteq \mathsf{MinFTraces}(\mathcal{S})$,
$\mathcal{TS}$ is strict if $\bigwedge_{\mathcal{TC} \in \mathcal{TS}}(\mathsf{Traces}(\mathcal{TC}) \cap \mathsf{MinFTraces}(\mathcal{S}) \subseteq \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}))$.

3 Recursive Tiles Systems and Their Properties

In this section, we define the *Recursive Tiles Systems* (RTS), a model to define infinite state IOLTS based on the regular graphs of [7]. We present some key properties of these systems relative to ε-closure (suppression of internal actions), product and determinization that will be useful for test generation in the next sections.

Definition 7. *A* recursive tile system *(RTS) is a tuple $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$ where*

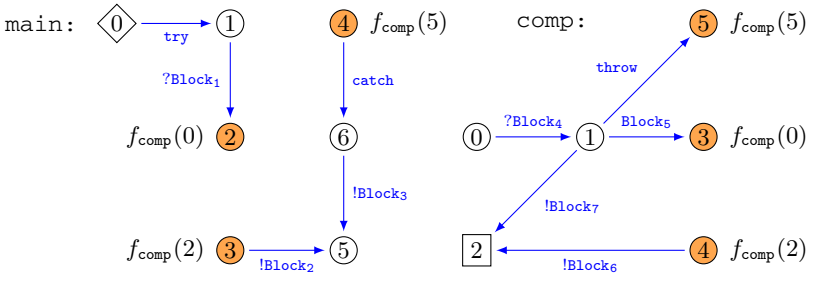
- *$\Sigma = \Sigma_? \cup \Sigma_! \cup \Sigma_\tau$ is a finite alphabet of* actions *partitioned into inputs, outputs and internal actions,*
- *Λ is a finite set of* colours *with a particular one* `init` *marking initial states.*
- *$\mathcal{T}$ is a set of tiles $t_\mathcal{A} = ((\Sigma, \Lambda), Q_\mathcal{A}, \rightarrow_\mathcal{A}, \mathcal{C}_\mathcal{A}, F_\mathcal{A})$ defined on (Σ, Λ) where*
  - *$Q_\mathcal{A} \subseteq \mathbb{N}$ is the set of* vertices*,*
  - *$\rightarrow_\mathcal{A} \subseteq Q_\mathcal{A} \times \Sigma \times Q_\mathcal{A}$ is a finite set of* transitions*,*
  - *$\mathcal{C}_\mathcal{A} \subseteq Q_\mathcal{A} \times \Lambda$ is a finite set of* coloured vertices*,*
  - *$F_\mathcal{A} \subseteq \mathcal{T} \times 2^{\mathbb{N}\times\mathbb{N}}$, the* frontier*, relates to some* tile*, $t_\mathcal{B}$, a partial function (often denoted $f_\mathcal{B}$) over $\mathbb{N}$, associating to vertices of $Q_\mathcal{B}$, vertices of $Q_\mathcal{A}$.*
- *$t_0 \in \mathcal{T}$ is an initial tile (the axiom).*

The frontier $F_\mathcal{A}$ of a tile $t_\mathcal{A}$ is used to append tiles $t_\mathcal{B}$ to $t_\mathcal{A}$: the frontier of $t_\mathcal{A}$ identifies tiles $t_\mathcal{B}$ and how some vertices of $t_\mathcal{B}$ are merged with vertices of $t_\mathcal{A}$.

A tile $t_\mathcal{A}$ defines an IOLTS $[t_\mathcal{A}] = (Q_\mathcal{A}, \Sigma, \Lambda, \rightarrow_\mathcal{A}, \mathcal{C}_\mathcal{A}, \mathtt{init})$.

Example 1. *The following example presents an RTS abstracting the program of Fig. 1.* $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_{\texttt{main}})$ *with* $\Sigma^{\tau} = \{\texttt{try}, \texttt{throw}, \texttt{catch}, \texttt{Block}_5\}$, $\Sigma^{?} = \{?\texttt{Block}_1, ?\texttt{Block}_4\}$, $\Sigma^{!} = \{!\texttt{Block}_2, !\texttt{Block}_3, !\texttt{Block}_6, !\texttt{Block}_7\}$, $\Lambda = \{\texttt{init}, \texttt{succ}\}$, $\mathcal{T} = \{t_{\texttt{main}}, t_{\texttt{comp}}\}$ *a set of tiles, and* $t_{\texttt{main}}$ *the initial tile.*

- $t_{\texttt{main}} = ((\Sigma, \Lambda), Q_{\texttt{main}}, \rightarrow_{\texttt{main}}, \mathcal{C}_{\texttt{main}}, F_{\texttt{main}})$ *with*
  $Q_{\texttt{main}} = \{0, 1, 2, 3, 4, 5, 6\}$, $\mathcal{C}_{\texttt{main}} = \{(0, \texttt{init})\}$ *(*$\texttt{init}$ *depicted by* $\diamond$*)*
  $F_{\texttt{main}} = \{(\texttt{comp}, \{0 \rightarrow 2, 2 \rightarrow 3, 5 \rightarrow 4\})\}$, *and* $\rightarrow_{\texttt{main}}$ *depicted below,*
- $t_{\texttt{comp}} = (\texttt{comp}, (X, \Sigma, \Lambda), Q_{\texttt{comp}}, \rightarrow_{\texttt{comp}}, \mathcal{C}_{\texttt{comp}}, F_{\texttt{comp}})$ *with*
  $Q_{\texttt{comp}} = \{0, 1, 2, 3, 4, 5\}$, $\rightarrow_{\texttt{comp}} \mathcal{C}_{\texttt{comp}} = \{(2, \texttt{succ})\}$ *(*$\texttt{succ}$ *depicted by* $\square$*),*
  $F_{\texttt{comp}} = \{(\texttt{comp}, \{0 \rightarrow 3, 2 \rightarrow 4, 5 \rightarrow 5\})\}$ *and* $\rightarrow_{\texttt{comp}}$ *depicted below.*



For the frontier, e.g., in the tile $t_{\texttt{main}}$, $f_{\texttt{comp}}(0)$ (2) *means that* $(\texttt{comp}, \{0 \rightarrow 2\})$ *belongs to* $F_{\texttt{main}}$*, i.e. the vertex* 0 *of* $t_{\texttt{comp}}$ *is associated to the vertex* 2 *of* $t_{\texttt{main}}$.

The semantics of an RTS is formally defined by an IOLTS by a *tiling* operation that appends tiles to another tile (initially, the axiom), inductively defining an IOLTS. Formally, given a set of tiles $\mathcal{T}$ and a tile $t_{\mathcal{E}} = ((\Sigma, \Lambda), Q_{\mathcal{E}}, \rightarrow_{\mathcal{E}}, \mathcal{C}_{\mathcal{E}}, F_{\mathcal{E}})$ with $F_{\mathcal{E}}$ defined on $\mathcal{T}$, the tiling of $t_{\mathcal{E}}$ by $\mathcal{T}$, denoted by $\mathcal{T}(t_{\mathcal{E}})$, is the tile $t'_{\mathcal{E}} = ((\Sigma, \Lambda), Q'_{\mathcal{E}}, \rightarrow'_{\mathcal{E}}, \mathcal{C}'_{\mathcal{E}}, F'_{\mathcal{E}})$ iteratively defined according to the elements of the frontier $F_{\mathcal{E}}$, as follows:

1. Initially, $Q'_{\mathcal{E}} = Q_{\mathcal{E}}$, $\rightarrow'_{\mathcal{E}} = \rightarrow_{\mathcal{E}}$, $\mathcal{C}'_{\mathcal{E}} = \mathcal{C}_{\mathcal{E}}$ $F'_{\mathcal{E}} = \emptyset$;
2. for each pair $(t_{\mathcal{B}}, f_{\mathcal{B}}) \in F_{\mathcal{E}}$, with $t_{\mathcal{B}} = ((\Sigma, \Lambda), Q_{\mathcal{B}}, \rightarrow_{\mathcal{B}}, \mathcal{C}_{\mathcal{B}}, F_{\mathcal{B}}) \in \mathcal{T}_{\mathcal{B}}$,
   let $\varphi_{\mathcal{B}} : Q_{\mathcal{B}} \rightarrow \mathbb{N}$ be the injection mapping vertices of $Q_{\mathcal{B}}$ to new vertices of $Q'_{\mathcal{E}}$ with $\varphi_{\mathcal{B}}(n) := f_{\mathcal{B}}(n)$ whenever $n \in dom(f_{\mathcal{B}}), n + \max(Q'_{\mathcal{E}}) + 1$ otherwise, where $\max(Q'_{\mathcal{E}})$ is the vertex with greatest value in $Q'_{\mathcal{E}}$. The tile $t'_{\mathcal{E}}$ is then defined by:
   - $Q'_{\mathcal{E}} = Q'_{\mathcal{E}} \cup Im(\varphi_{\mathcal{B}})$,
   - $\rightarrow'_{\mathcal{E}} = \rightarrow'_{\mathcal{E}} \cup \{(\varphi_{\mathcal{B}}(n), a, \varphi_{\mathcal{B}}(n')) \mid (n, a, n') \in \rightarrow_{\mathcal{B}}\}$,
   - $\mathcal{C}'_{\mathcal{E}} = \mathcal{C}'_{\mathcal{E}} \cup \{(\varphi_{\mathcal{B}}(n), \lambda) \mid (n, \lambda) \in \mathcal{C}_{\mathcal{B}}\}$,
   - $F'_{\mathcal{E}} = F'_{\mathcal{E}} \cup \{(t_{\mathcal{C}}, \{(\varphi_{\mathcal{B}}(j), f_{\mathcal{C}}(j)) \mid j \in dom(f_{\mathcal{C}})\}) \mid (t_{\mathcal{C}}, f_{\mathcal{C}}) \in F_{\mathcal{B}}\}$. The update of F' expresses that the frontier of the new tile $t'_{\mathcal{A}}$ is composed from those of the tiles that have been added.

Remark 1. *In a tiling, the order chosen to append a copy of the tiles that belong to the frontier is not important. Two different orders would produce isomorphic tiles (up to a renaming of vertices).*

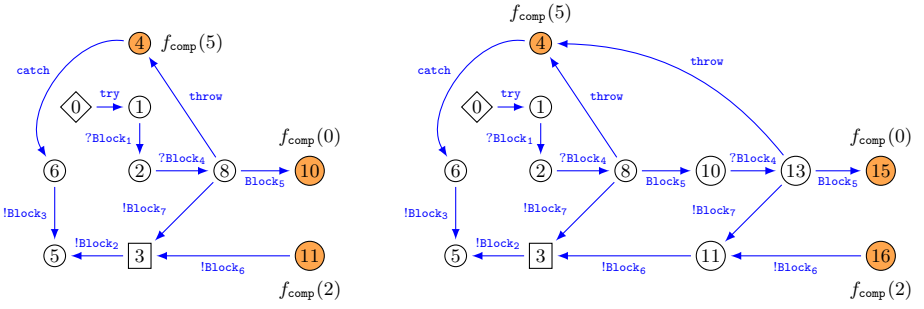


Fig. 2. $\mathcal{T}(t_{\mathtt{main}})$ and $\mathcal{T}^2(t_{\mathtt{main}})$ tiles

Example 2. *We illustrate the principle of tiling using the RTS defined in Example 1. Consider that $t_{\mathtt{main}}$ is the initial tile. Its tiling $\mathcal{T}(t_{\mathtt{main}})$, is performed as follows: there is a single element in its frontier; we add a copy of $t_{\mathtt{comp}}$ (with new vertices), identifying vertices $2, 3$ and 4 of $t_{\mathtt{main}}$ to vertices $0, 2$ and 5 of $t_{\mathtt{comp}}$.*

The resulting tile is depicted in Fig. 2 (left-hand side). This new tile may be in turn extended by adding a copy of $t_{\mathtt{comp}}$, identifying $4, 10$ and 11 to $0, 2$ and 5. Again, we illustrate the resulting tile in Fig. 2 (right-hand side) (observe that our definition of $\varphi_{\mathtt{comp}}$ induces that some elements of $\mathbb{N}$ are left out). Obviously iterating this process will result in vertex 4 having infinite in*-degree.*

An IOLTS is finally obtained from an RTS as the union of the IOLTS of tiles resulting from the iterated tilings from the axiom. Formally,

Definition 8. *Let $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$ be an RTS. $\mathcal{R}$ defines an IOLTS $[\![\mathcal{R}]\!] = (Q_{\mathcal{R}}, \Sigma, \Lambda, \rightarrow_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}}, \mathtt{init})$ given by*

$$\bigcup_k [\mathcal{T}^k(t_0)]$$

The infinite union of Definition 8 is valid because, by construction, for all $k \geq 0$: $[\mathcal{T}^k(t_0)] \subseteq [\mathcal{T}^{k+1}(t_0)]$, where $\subseteq$ is understood as the inclusion of IOLTS, *i.e.* inclusion of states, transitions and colourings.

For an RTS $\mathcal{R}$ with axiom t_0, and a state q in $[\![\mathcal{R}]\!]$, $\ell(q)$ denotes the *level* of q, *i.e.* the least $k \in \mathbb{N}$ such that q is a state of $[\mathcal{T}^k(t_0)]$, and $t(q)$ denotes the tile in $\mathcal{T}$ that created q. For a vertex v of a tile of $\mathcal{R}$, $[\![v]\!]$ denotes the set of states in $[\![\mathcal{R}]\!]$ corresponding to v.

Requirement 1. *In order to simplify proofs, we impose some technical restrictions on the RTS, $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$, that can be ensured by a normalisation step, without loss of generality:*

1. *for any state, q, of finite degree in $[\![\mathcal{R}]\!]$, every transition connected to q is either defined in $t(q)$ or one of the tiles of its frontier (this may be checked on $\mathcal{T}$)*
2. *the set of enabled actions in copies of a vertex v is uniform (for all vertices v in $\mathcal{R}$, for all q, q' in $[\![v]\!]$, $\Gamma_{[\![\mathcal{R}]\!]}(q) = \Gamma_{[\![\mathcal{R}]\!]}(q')$), thus can be written $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!])$. Furthermore, we may assume that each vertex possesses a colour reflecting this value (see Corollary 1 below).*

Remark 2. *The IOLTS obtained from RTS correspond to the equational, or regular graphs of [7] and [3]. These IOLTS are derived from an axiom using deterministic* HR*-grammars. Each such grammar may be transformed into a tiling system, and conversely. Our definition aims at a greater simplicity.*

Reachability. Computation of (co)reachability sets, that are central for verification and safety problems, as well as for test generation, are effective for RTS:

Proposition 2 ([3]). *Given an RTS $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$, a sub-alphabet $\Sigma' \subseteq \Sigma$, a colour $\lambda \in \Lambda$, and a new colour $r_\lambda \notin \Lambda$, an RTS $\mathcal{R}' = ((\Sigma, \Lambda \cup \{r_\lambda\}), \mathcal{T}', t'_0)$ can be effectively computed, such that $[\![\mathcal{R}']\!]$ is isomorphic to $[\![\mathcal{R}]\!]$ with respect to the transitions and the colouring by Λ, and states reachable from a state coloured λ by actions in Σ' are coloured r_λ: $\mathcal{C}_{\mathcal{R}'}(r_\lambda) = reach_{[\![\mathcal{R}']\!]}(\mathcal{C}(\lambda), \Sigma')$. The same result holds for states co-reachable from λ.*

Proposition 3.13 (b) of [3] enables to perform several computations related to our purpose. We rephrase it for RTS.

Proposition 3 ([3]). *Given an RTS $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$, for any subset S in $\mathbb{N} \cup \{\infty\}$ and new colour $\#_S \notin \Lambda$, it is possible to compute an RTS $\mathcal{R}' = ((\Sigma, \Lambda \cup \{\#_S\}), \mathcal{T}', t'_0)$ such that $[\![\mathcal{R}]\!]$ is isomorphic to $[\![\mathcal{R}']\!]$ with respect to the transitions and the colouring by Λ, and every state of $[\![\mathcal{R}']\!]$ of (in- or out- or total-) degree is in S is coloured by $\#_S$.*

In particular this result enables to identify on the set of tiles properties of the states, like deadlocks, inputlock. The following corollary is also a direct consequence of this proposition (performing successive colouring for computing the degree related to some actions).

Corollary 1. *Given an RTS $\mathcal{R}$ and a vertex v of a tile t of $\mathcal{R}$, for any q in $[\![v]\!]$ the allowed actions $\Gamma_{[\![\mathcal{R}]\!]}(q)$ in state q can be effectively computed.*

Observable Behaviour of RTS: Abstracting away internal transitions is important for test generation. With the following proposition, it is possible to do it for RTS.

Proposition 4. *From an RTS $\mathcal{R}$ with IOLTS $[\![\mathcal{R}]\!] = (Q_\mathcal{R}, \Sigma, \Lambda, \rightarrow_\mathcal{R}, \mathcal{C}_\mathcal{R}, \texttt{init})$ and visible actions $\Sigma^o \subseteq \Sigma$, one can effectively compute an RTS $Clo(\mathcal{R})$ with same colours Λ, whose IOLTS $[\![Clo(\mathcal{R})]\!] = (Q'_\mathcal{R}, \Sigma^o, \Lambda, \rightarrow'_\mathcal{R}, \mathcal{C}'_\mathcal{R}, \texttt{init})$ has no internal action, is of finite* out*-degree, and for any colour $\lambda \in \Lambda$, $\mathsf{Traces}_{\mathcal{C}_\mathcal{R}(\lambda)}([\![\mathcal{R}]\!]) = \mathsf{Traces}_{\mathcal{C}'_\mathcal{R}(\lambda)}([\![Clo(\mathcal{R})]\!])$.*

This result is classical and follows mainly from [3]. Infinite out-degree may occur whenever there is an infinite sequence of internal transitions. However, careful computation of $Clo(\mathcal{R})$ enables to avoid such occurrences.

Synchronous Product: The synchronous product of IOLTS is the operation used to intersect languages, and is useful for test selection using a test purpose. We can prove that the product of an RTS with a finite IOLTS is an RTS. More precisely, given any RTS $\mathcal{R}$ with IOLTS $[\![\mathcal{R}]\!]$, and a finite state IOLTS $\mathcal{A}$, one can compute an RTS denoted

by $\mathcal{R} \times \mathcal{A}$ such that $[\![\mathcal{R} \times \mathcal{A}]\!] = [\![\mathcal{R}]\!] \times \mathcal{A}$ (the $\times$ on the right-hand side of the equality is the product for IOLTS).

In general, the product of two RTS is not recursive. Indeed, the intersection of two context-free languages can be obtained by a product of two RTS, if such a product was recursive the intersection of two context-free languages would be a context-free language (*e.g.*, $\{a^n b^n c^k \mid n, k \in \mathbb{N}\} \cap \{a^n b^k c^k \mid n, k \in \mathbb{N}\}$ is not context-free).

Weighted RTS. In the following we will often consider an important class of RTS. This class possesses the valuable property of being determinizable.

Definition 9. *An RTS $\mathcal{R}$ with IOLTS $[\![\mathcal{R}]\!] = (Q_{\mathcal{R}}, \Sigma, \Lambda, \rightarrow_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}}, \texttt{init})$ is* weighted *if $\mathcal{C}_{\mathcal{R}}(\texttt{init})$ is a singleton $\{q_0\}$, and for any $u \in \Sigma^*$ and any states $q, q' \in Q_{\mathcal{R}}$, $q_0 \xrightarrow{u} q$ and $q_0 \xrightarrow{u} q'$ implies $\ell(q) = \ell(q')$ (same level).*

Note that determining if an RTS is weighted is decidable, using an algorithm from [5].

Example 3. *Assuming internal actions are not observable, the RTS defined in Example 1 may be weighted or not depending on the way the closure is performed. A backward closure ensures that the IOLTS is weighted: in fact, it is, then, deterministic. A forward closure induces non-determinism at* $?\texttt{Block}_4$. *Since path ending with this block would either be silently followed by* `throw` *and thus end in the initial tile (level* 0*), or be followed by* $\texttt{Block}_5$ *and terminate at the next level (at least* 1*).*

Determinization of recursive LTS. An RTS $\mathcal{R}$ is *deterministic* if its underlying IOLTS $[\![\mathcal{R}]\!]$ is deterministic. This is decidable from the set of tiles defining it (for example using Proposition 3). However, since PDA cannot be determinized in general, there is no hope to determinize an arbitrary RTS. Still, there are some classes of determinizable PDA, like visibly PDA [2], or, more recently, the *weighted grammars* of [4]. These grammars define a class of PDA that can be determinized and which both subsume the visibly PDA and the height deterministic PDA [11].

Proposition 5 ([5]). *Any weighted RTS $\mathcal{R}$ can be transformed into a* deterministic *one $\mathcal{D}(\mathcal{R})$ with same set of traces and, for any colour, same traces accepted in this colour.*

Example 4. *Following Example 3, assume that vertex* 5 *is not in any frontier anymore, and suppose that there are* 3 *transitions labelled* $?\texttt{Block}_4$ *between* 0 *and respectively* 1*,* 3 *and* 5*. This is a weighted system. In such a situation, determinization would simply perform a finite LTS determinization in the tile $t_{\texttt{comp}}$. In the general case some tiles need to be merged first.*

4 Off-Line Test Generation for Weighted RTS

In this section and the following, we consider the generation of test cases from RTS. We focus, here, on weighted RTS, which are determinizable, and propose an off-line test generation algorithm that operates a selection guided by a test purpose (specified by a finite IOLTS). Computations are performed at the RTS level with consequences on the underlying IOLTS semantics, enabling the proof of properties on generated test cases.

4.1 Construction of the Canonical Tester

Quiescence. As seen in Section 2 quiescence represents the absence of action in the specification. Given a specification defined by a RTS $\mathcal{S}$, detecting vertices where the absence of reaction is permitted enables to construct a suspended specification, $\Delta(\mathcal{S})$.

For finite state IOLTS, livelocks come from loops. On the contrary, for IOLTS defined by RTS, livelocks may come from infinite paths of silent actions involving infinitely many states. We call such paths *divergent*.

Lemma 1. *For a RTS $\mathcal{R}$, there exists a loop or a divergent path in $[\![\mathcal{R}]\!]$ if and only if there exists a vertex v and two states $q_1, q_2 \in [\![v]\!]$ with $\ell(q_1) \leq \ell(q_2)$ such that $q_1 \xrightarrow{\sigma} q_2$ for some $\sigma \in \Sigma^{\tau *}$ and for all states q on this path, $\ell(q_1) \leq \ell(q)$.*

Proof. ($\Rightarrow$) Let $p = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 ...$ be an infinite path in $[\![\mathcal{R}]\!]$, with $\forall k \in \mathbb{N}, a_k \in \Sigma^{\tau}$. If p contains a loop, there exits one state of minimal level in this loop, let q_1 be this state. Now consider an elementary path. As each state is only seen once, we build a sequence of states q_{i_k} such that $\forall i_k \leq j, \ell(q_{i_k}) \leq \ell(q_j)$. As there are only a finite number of vertices, there is a least one v such two states of $[\![v]\!]$ appear in this path. Let these two states be q_1 and q_2.

($\Leftarrow$) If there exist a vertex v and two states $q_1, q_2 \in [\![v]\!]$ with $\ell(q_1) = \ell(q_2)$ such that $q_1 \xrightarrow{\sigma} q_2$ for $\sigma \in \Sigma^{\tau +}$, and for all states q on this path, $\ell(q_1) \leq \ell(q)$, then $q_1 = q_2$, since any path from two distinct occurrences of the same tile at the same level involves vertices of lower level. Hence this path is a loop. Otherwise, $\ell(q_1) < \ell(q_2)$, let $p_0 := q_1 \xrightarrow{\sigma} q_2$ for $\sigma \in \Sigma^{\tau +}$, since for all q in this path, $\ell(q_1) \leq \ell(q)$. Thus, by definition, a similar path may be constructed reaching a state q_3, with, $q_2 \xrightarrow{\sigma'} q_3$ for $\sigma' \in \Sigma^{\tau +}$, $\ell(q_2) < \ell(q_3)$, and $\ell(q_2) \leq \ell(q)$ for all q involved. Iterating this process enables to produce an infinite path in $[\![\mathcal{R}]\!]$ satisfying the hypothesis. □

Proposition 6. *From any RTS $\mathcal{R}$, it is effective to build an RTS denoted $\Delta(\mathcal{R})$ such that $[\![\Delta(\mathcal{R})]\!] = \Delta([\![\mathcal{R}]\!])$. Consequently* $\mathsf{Traces}([\![\Delta(\mathcal{R})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!])$.

Proof. Let $\mathcal{R}$ be a RTS, we add self-loops δ as follows.

For deadlock and output lock, we use Requirement 1, item 2, which ensures that for a vertex v in a tile t of $\mathcal{R}$, has a uniform value for $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!])$. The δ-transitions are added to each v in $\mathcal{R}$ such that $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!]) = \emptyset$ or $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!]) \subseteq \Sigma^{\mathcal{R}}_{?}$. This operation produces a new RTS $\mathcal{R}'$.

For livelocks, there are two different cases: internal loops and *divergent paths*. From Lemma 1 we know that such situations may be detected from self-reaching vertices. This result also ensures that this detection may be performed taking each tile as an axiom. Then, for each tile t in $\mathcal{R}'$:

- Colour each vertex v of tile t by a colour λ_v not in $\Lambda_{\mathcal{R}'}$.
- Use Proposition 2 to colour by λ'_v vertices in $reach_{[\![\mathcal{R}_t]\!]}(\Sigma^{\tau}, \lambda)$, where $\mathcal{R}'_t$ is the RTS identical to $\mathcal{R}'$, with initial tile t. This computation simply enables to detect vertices involved in an infinite path, but the resulting RTS is not kept.
- Each vertex v coloured by both λ_v and λ'_v is involved in a livelock. We add quiescence to each such vertex in $\mathcal{R}'$ to produce $\Delta(\mathcal{R})$.

□

Output Completion. After using Proposition 6 for the computation of $\Delta(\mathcal{S})$ from the specification $\mathcal{S}$, the next step is to complete $\Delta(\mathcal{S})$ to recognise $\mathsf{STraces}(\mathcal{S}).\Sigma^{!\delta}$. The complete suspended specification, denoted by $CS(\mathcal{S})$, is computed from $\Delta(\mathcal{S})$ as follows: a new colour *UnS* is added to detect paths leading to unspecified behaviours. Then, for every tile t, a new vertex, v_t^{UnS}, is added (having colour *UnS*), new transitions leading to v_t^{UnS} are added as well:

$$\left\{ v \xrightarrow{a} v^{UnS} \mid v \in Q_A \wedge a \in \Sigma^{!\delta} \wedge a \notin \Gamma_{[\![\Delta(\mathcal{S})]\!]}([\![v]\!]) \right\}.$$

By construction, we get $\mathsf{Traces}([\![CS(\mathcal{S})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!]).\Sigma^{!\delta}_{\mathcal{S}} \cup \mathsf{STraces}([\![\mathcal{S}]\!])$ and $\mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!])$.

Canonical Tester. Whenever $CS(\mathcal{S})$ is weighted, Proposition 5 enables to determinize it into $\mathcal{D}(CS(\mathcal{S}))$. From $\mathcal{D}(CS(\mathcal{S}))$ we build a new RTS $Can(\mathcal{S})$ called the *canonical tester* of $\mathcal{S}$ as follows:

- a new colour *Fail* is considered and vertices of $\mathcal{D}(CS(\mathcal{S}))$ are coloured by *Fail* if composed of vertices all coloured by *UnS* in $CS(\mathcal{S})$.
- inputs and outputs are mirrored in $Can(\mathcal{S})$ wrt. $\mathcal{S}$.

From this construction we can deduce that

$$\mathsf{Traces}_{\mathcal{C}(_{Fail})}([\![Can(\mathcal{S})]\!]) = \mathsf{MinFTraces}([\![\mathcal{S}]\!]) \quad (1)$$

$$\mathsf{Traces}_{\overline{\mathcal{C}}(_{Fail})}([\![Can(\mathcal{S})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!]) \quad (2)$$

and $\mathsf{Traces}([\![Can(\mathcal{S})]\!])$ is their disjoint union.

In fact $\mathsf{Traces}_{\overline{\mathcal{C}}(_{Fail})}([\![Can(\mathcal{S})]\!]) = \mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!])$ and

$$\begin{aligned}
\mathsf{Traces}_{\mathcal{C}(_{Fail})}([\![Can(\mathcal{S})]\!]) &= \mathsf{Traces}_{\mathcal{C}(_{UnS})}([\![CS(\mathcal{S})]\!]) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!]) \\
&= \mathsf{Traces}([\![CS(\mathcal{S})]\!]) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!]) \\
(\text{as}\mathsf{Traces}([\![CS(\mathcal{S})]\!]) \text{ is} &\text{ the union } \mathsf{Traces}_{\mathcal{C}(_{UnS})}([\![CS(\mathcal{S})]\!]) \cup \mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!])\,) \\
&= \mathsf{STraces}([\![\mathcal{S}]\!]).\Sigma^{!\delta}_{\mathcal{S}} \setminus \mathsf{STraces}([\![\mathcal{S}]\!]) \\
&= \mathsf{MinFTraces}([\![\mathcal{S}]\!])
\end{aligned}$$

From (1) it immediately follows that the test suite $\mathcal{TS}$ reduced to $\{Can(\mathcal{S})\}$ is sound and exhaustive (see Section 2). $\mathcal{TS}$ is also strict, which is proved as follows: $\mathsf{Traces}([\![Can(\mathcal{S})]\!]) \cap \mathsf{MinFTraces}([\![\mathcal{S}]\!]) = (\mathsf{Traces}_{\mathcal{C}(_{Fail})}([\![Can(\mathcal{S})]\!]) \cup \mathsf{STraces}([\![\mathcal{S}]\!]) \cap \mathsf{MinFTraces}([\![\mathcal{S}]\!]) = \mathsf{Traces}_{\mathcal{C}(_{Fail})}([\![Can(\mathcal{S})]\!])$ using the disjoint union and (1).

Test Case Selection with a Test Purpose. The canonical tester has important properties, but one may want to focus on particular behaviours, using a test purpose. In our formal framework, a *test purpose* is a deterministic finite IOLTS $\mathcal{TP}$ over $\Sigma^{\circ\delta}$, with a particular colour *Accept*. States coloured by *Accept* have no successors.

As seen in the previous section, the product $\mathcal{P}$ between $Can(\mathcal{S})$ and $\mathcal{TP}$ is an RTS. On this product, new colours are specified as follows :

- $\mathcal{C}_{\mathcal{P}}(Fail) = \mathcal{C}_{Can(\mathcal{S})}(Fail) \times Q_{\mathcal{TP}}$
- $\mathcal{C}_{\mathcal{P}}(Pass) = \overline{\mathcal{C}}_{\mathcal{P}}(Fail) \times \mathcal{C}_{\mathcal{TP}}(Accept)$

- $\mathcal{C}_{\mathcal{P}}(None) = Coreach(\mathcal{C}_{\mathcal{P}}(Pass)) \setminus \mathcal{C}_{\mathcal{P}}(Pass)$
- $\mathcal{C}_{\mathcal{P}}(Inc) = Q_P \setminus (\mathcal{C}_{\mathcal{P}}(Fail) \cup \mathcal{C}_{\mathcal{P}}(Pass) \cup \mathcal{C}_{\mathcal{P}}(None))$

Note that, by construction, each state has a unique colour in $\{Fail, Pass, None, Inc\}$. States coloured by *Fail* or *Pass* have no successors, and states coloured by *Inc* have only *Fail* or *Inc* successors.

In order to avoid states coloured by *Inc* where the test purpose cannot be satisfied anymore, transitions labelled by an output (input of $\mathcal{S}$, controllable by the environment) and leading to a state coloured by *Inc* may be pruned, as well as those leaving *Inc*. Consequently, runs leading to an *Inc* coloured state necessarily end with an input action.

Finally, the test case $\mathcal{TC}$ generated from $\mathcal{S}$ and $\mathcal{TP}$ is the product $\mathcal{P}$, equipped with new colours *Fail*, *Pass*, *None*, *Inc* and pruned as above.

Example 5. *Figure 3, below, represents the test case obtained from Example 1 with the test purpose accepting only* $(\Sigma^{o\delta})^*\,?\mathtt{Block}_4\,?\mathtt{Block}_4\,(\Sigma^{o\delta})^*\,!\mathtt{Block}_2$. *The vertices labelled by* F *correspond to the one coloured by Fail but is split for better readability. Triangle vertices are those coloured by Inc. Observe that each vertex is a set of pairs, so indices depicted below are not related to the original ones.*

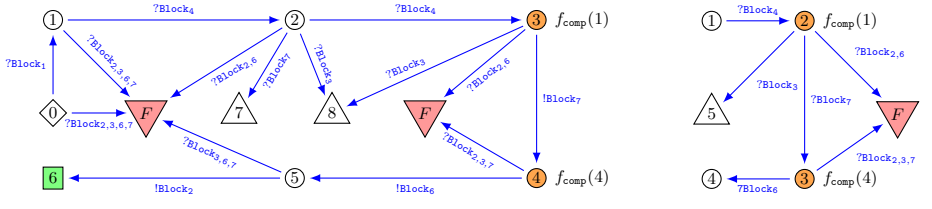


Fig. 3. Example of a test case

4.2 Properties of Generated Test Cases

We now prove the requested properties of test cases defined in Section 2, relating test case failure to non-conformance, and a new property, precision, that relates test case success (*Pass* verdict) to the satisfaction of the test purpose.

Soundness and Strictness. According to the construction of $\mathcal{P}$, the definition of $\mathcal{C}_{\mathcal{P}}(Fail)$, and pruning, selection by $\mathcal{TP}$ do not add any colouring by *Fail* with respect to $Can(\mathcal{S})$, thus $\mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket\mathcal{TC}\rrbracket) = \mathsf{Traces}(\llbracket\mathcal{TC}\rrbracket) \cap \mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket Can(\mathcal{S})\rrbracket)$. By (1) we deduce $\mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket\mathcal{TC}\rrbracket) = \mathsf{Traces}(\llbracket\mathcal{TC}\rrbracket) \cap \mathsf{MinFTraces}(\llbracket\mathcal{S}\rrbracket) \subseteq \mathsf{MinFTraces}(\llbracket\mathcal{S}\rrbracket)$ which proves both strictness (equality) and soundness (inclusion).

Exhaustiveness. We prove that the test suite $\mathcal{TS}$ composed of all test cases that can be generated from arbitrary test purposes $\mathcal{TP}$ is exhaustive. We thus need to establish the inequality $\bigcup_{\mathcal{TC}\in\mathcal{TS}} \mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket\mathcal{TC}\rrbracket) \supseteq \mathsf{MinFTraces}(\llbracket\mathcal{S}\rrbracket)$.

Let $\sigma' = \sigma.a \in \mathsf{MinFTraces}(\llbracket\mathcal{S}\rrbracket) = \mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket Can(S)\rrbracket)$ be a minimal non-conformant trace for $\mathcal{S}$. We have $\sigma \in \mathsf{STraces}(\llbracket\mathcal{S}\rrbracket)$ and there exists $b \in \Sigma^{!\delta}$ such

that $\sigma.b \in \mathsf{STraces}([\![\mathcal{S}]\!])$ (if no output continues σ in $\mathsf{STraces}([\![\mathcal{S}]\!])$, a δ does). Now consider a test purpose $\mathcal{TP}$ such that $\sigma.b \subseteq \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP})$ and let $\mathcal{TC}$ be the test case generated from $\mathcal{S}$ and $\mathcal{TP}$. By construction of $\mathcal{TC}$, we get $\sigma' \in \mathsf{Traces}_{Fail}([\![\mathcal{TC}]\!])$.

Precision. As a complement to the above properties, *precision* relates test cases to test purposes. It says that the verdict *Pass* is returned as soon as possible, once the test purpose is satisfied. Formally, a test case $\mathcal{TC}$ is precise with respect to $\mathcal{TP}$ if $\mathsf{Traces}_{\mathcal{C}(Pass)}([\![\mathcal{TC}]\!]) = \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \mathsf{STraces}([\![\mathcal{S}]\!]) \cap \mathsf{Traces}([\![\mathcal{TC}]\!])$.

By construction, states coloured by *Pass* are those coloured by *Accept* in $\mathcal{TP}$ and not by *Fail* in $Can(\mathcal{S})$. Thus $\mathsf{Traces}_{\mathcal{C}(Pass)}([\![\mathcal{TC}]\!]) = \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \mathsf{STraces}([\![\mathcal{S}]\!])$ which (since $\mathsf{Traces}_{\mathcal{C}(Pass)}([\![\mathcal{TC}]\!]) \subseteq \mathsf{Traces}([\![\mathcal{TC}]\!])$) implies precision.

5 On-Line Test Generation from RTS

For the general case, determinization is an issue, as seen in Section 3. As usual in similar cases [13], one may rely on "on-line" test generation (executing test cases while generating them) or equivalently produce test cases as finite trees.

5.1 Test Case Generation

Output-Completion and ϵ-Closure. The process starts from the output-completed specification $CS(\mathcal{S})$ defined in Section 4. This time, the canonical tester cannot be built from $CS(\mathcal{S})$. However, using Proposition 4, one can built $Clo(CS(\mathcal{S}))$, ensuring the following properties:

$$\mathsf{MinFTraces}(\mathcal{S}) \subseteq \mathsf{Traces}_{\mathcal{C}(UnS)}(Clo(CS(\mathcal{S}))) \subseteq \mathsf{STraces}(\mathcal{S}).\Sigma^{!\delta}$$

$$\mathsf{Traces}_{\overline{\mathcal{C}}(UnS)}(Clo(CS(\mathcal{S}))) = \mathsf{STraces}(\mathcal{S})$$

Product and Colouring. The next step consists in the computation of the product of $Clo(CS(\mathcal{S}))$ with a test purpose given as a complete finite IOLTS $\mathcal{TP}$. Let $\mathcal{P} = Clo(CS(\mathcal{S})) \times \mathcal{TP}$ be this product, one may define the following new colours on $\mathcal{P}$ using a co-reachability analysis:

- $\mathcal{C}_{\mathcal{P}}(UnS) = \mathcal{C}_{Clo(CS(\mathcal{S}))}(UnS) \times Q_{\mathcal{TP}}$
- $\mathcal{C}_{\mathcal{P}}(Pass) = \overline{\mathcal{C}}_{Clo(CS(\mathcal{S}))}(UnS) \times \mathcal{C}_{\mathcal{TP}}(Accept)$
- $\mathcal{C}_{\mathcal{P}}(None) = Coreach(\mathcal{C}_{\mathcal{P}}(Pass)) \setminus \mathcal{C}_{\mathcal{P}}(Pass)$
- $\mathcal{C}_{\mathcal{P}}(Inc) = Q_{\mathcal{P}} \setminus (\mathcal{C}_{\mathcal{P}}(Fail) \cup \mathcal{C}_{\mathcal{P}}(Pass) \cup \mathcal{C}_{\mathcal{P}}(None))$

Computing Test Cases. The last step consists in computing test cases in a way similar to [13]. These test cases will be modelled as finite trees. Formally such a finite tree will be a prefix-closed set of words in $\Sigma^{o\delta^*}.(\{Fail, Pass, None, Inc\} \cup \{\varepsilon\})$. Given a tree θ, for some symbol a, the notation $a;\theta \triangleq \{au \mid u \in \theta\}$, furthermore, given two trees θ, θ', the tree formed by the union of those trees is denoted by $\theta + \theta'$.

A test case $\mathcal{TC}$ is a tree built from $\mathcal{P}$ by taking as argument a set of states PS. Let us define test cases by applying the following algorithm recursively, starting from the initial state $\mathcal{C}_{\mathcal{P}}(init)$.

Choose non deterministically between one of the following operations.

1. (* Terminate the test case *)
   $\theta := \{None\}$
2. (* Give a next input to the implementation *)
   Choose any $a \in out(PS)$ such that
   $(PS \texttt{ after } a) \cap (\mathcal{C}_{\mathcal{P}}(Pass) \cup \mathcal{C}_{\mathcal{P}}(None)) \neq \emptyset$
   $\theta := a; \theta'$
   where θ' is obtained by applying the algorithm with $PS' = (PS \texttt{ after } a)$
3. (* Check the next output of the implementation *)

$$\theta := \sum_{a \in X_1} a; Fail + \sum_{a \in X_2} a; Inc + \sum_{a \in X_3} a; Pass + \sum_{a \in X_4} a; \theta'$$

with:

- $X_1 = \{a \mid PS \texttt{ after } a \subseteq \mathcal{C}_{\mathcal{P}}(UnS)\}$
- $X_2 = \{a \mid (PS \texttt{ after } a \subseteq (\mathcal{C}_{\mathcal{P}}(Inc) \cup \mathcal{C}_{\mathcal{P}}(UnS))) \wedge (PS \texttt{ after } a \cap \mathcal{C}_{\mathcal{P}}(Inc) \neq \emptyset)\}$
- $X_3 = \{a \mid PS \texttt{ after } a \cap \mathcal{C}_{\mathcal{P}}(Pass) \neq \emptyset\}$
- $X_4 = \{a \mid (PS \texttt{ after } a \cap \mathcal{C}_{\mathcal{P}}(Pass) = \emptyset) \wedge (PS \texttt{ after } a \cap \mathcal{C}_{\mathcal{P}}(None) \neq \emptyset)\}$
- θ' is obtained by applying the algorithm with $PS' = (PS \texttt{ after } a)$

Formally, a tree needs to be transformed into a test case IOLTS $\mathcal{TC}$ by an appropriate colouring of states ending in *Fail*, *Pass*, *Inc* or *None* after a suspension trace. We skip this for readability.

5.2 Properties of the Test Cases Generated On-Line

Soundness and Strictness. By definition of X_1, those traces of $\mathcal{TC}$ falling in a state coloured by *Fail* are those in $\mathsf{Traces}(\llbracket CS(\mathcal{S}) \rrbracket) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}_{(UnS)}}(\llbracket CS(\mathcal{S}) \rrbracket) = \mathsf{MinFTraces}(\llbracket \mathcal{S} \rrbracket)$. Thus $\mathsf{Traces}_{\mathcal{C}(Fail)}(\mathcal{TC}) = \mathsf{MinFTraces}(\llbracket \mathcal{S} \rrbracket) \cap \mathsf{Traces}(\mathcal{TC})$ which proves both soundness and strictness, as in the off-line case.

Exhaustiveness. The proof of exhaustiveness is similar to the one in Section 4, consisting in building a test purpose $\mathcal{TP}$ for each non-conformant trace, and proving that a possible resulting test case would produce a *Fail* after this trace.

Precision. From the construction of $\mathcal{TC}$, in particular, the set X_3, we have $\mathsf{Traces}_{\mathcal{C}(Pass)}(\mathcal{TC}) = \mathsf{Traces}_{\mathcal{C}(Pass)}(Clo(CS(\mathcal{S})) \times \mathcal{TP}) \cap \mathsf{Traces}(\mathcal{TC})$. Then, by definitions of the colours, we obtain: $\mathsf{Traces}_{\mathcal{C}(Pass)}(\mathcal{TC}) = \mathsf{Traces}_{\overline{\mathcal{C}}_{UnS}}(Clo(CS(\mathcal{S}))) \cap \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP})) \cap \mathsf{Traces}(\mathcal{TC})$. Which eventually proves precision: $\mathsf{Traces}_{\mathcal{C}(Pass)}(\mathcal{TC}) = \mathsf{STraces}(\mathcal{S}) \cap \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \mathsf{Traces}(\mathcal{TC})$.

6 Conclusion

In this paper we have presented recursive tile systems, a general model of IOLTS allowing for recursion. We have provided algorithms to produce sound, strict and exhaustive test suites, either off-line or on-line. These algorithms enable to employ test purposes (even, for the on-line case) which are a classical way to drive tests towards sensitive properties. We have also established the precision of our tests with respect to test purposes.

An interesting perspective would be to incorporate known results on probabilistic RTS. This would enable to take into account quantitative properties of systems, or to express coverage properties of finite test suites.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Analysis of Recursive State Machines. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 207–220. Springer, Heidelberg (2001)
2. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC 2004), pp. 202–211. ACM (2004)
3. Caucal, D.: Deterministic graph grammars. Texts in Logics and Games, vol. 2, pp. 169–250 (2007)
4. Caucal, D.: Synchronization of Regular Automata. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 2–23. Springer, Heidelberg (2009)
5. Caucal, D., Hassen, S.: Synchronization of Grammars. In: Hirsch, E.A., Razborov, A.A., Semenov, A., Slissenko, A. (eds.) CSR 2008. LNCS, vol. 5010, pp. 110–121. Springer, Heidelberg (2008)
6. Constant, C., Jeannet, B., Jéron, T.: Automatic Test Generation from Interprocedural Specifications. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 41–57. Springer, Heidelberg (2007)
7. Courcelle, B.: Graph rewriting: an algebraic and logic approach. In: Handbook of Theoretical Computer Science. Elsevier (1990)
8. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
9. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. Software Tools for Technology Transfer (STTT) 7(4), 297–315 (2005)
10. Jeannet, B., Jéron, T., Rusu, V.: Model-Based Test Selection for Infinite-State Reactive Systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 47–69. Springer, Heidelberg (2007)
11. Nowotka, D., Srba, J.: Height-Deterministic Pushdown Automata. In: Kučera, L., Kučera, A. (eds.) MFCS 2007. LNCS, vol. 4708, pp. 125–134. Springer, Heidelberg (2007)
12. Tretmans, G.J., Brinksma, H.: Torx: Automated model-based testing. In: Hartman, A., Dussa-Ziegler, K. (eds.) First European Conference on Model-Driven Software Engineering, pp. 31–43 (December 2003)
13. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)

Generation of Test Data Structures Using Constraint Logic Programming

Valerio Senni[1] and Fabio Fioravanti[2]

[1] DISP, University of Rome Tor Vergata, Rome, Italy
senni@disp.uniroma2.it
[2] Dipartimento di Scienze, University 'G. D'Annunzio', Pescara, Italy
fioravanti@sci.unich.it

Abstract. The goal of Bounded-Exhaustive Testing (BET) is the automatic generation of *all* the test cases satisfying a given invariant, within a given bound. When the input has a complex structure, the development of correct and efficient generators becomes a very challenging task. In this paper we use Constraint Logic Programming (CLP) to systematically develop generators of structurally complex test data.

Similarly to *filtering*-based test generation, we follow a declarative approach which allows us to separate the issue of (i) defining the test structure and invariant, from that of (ii) generating admissible test input instances. This separation helps improve the correctness of the developed test case generators. However, in contrast with filtering approaches, we rely on a symbolic representation and we take advantage of efficient search strategies provided by CLP systems for generating test instances.

Through some experiments on examples taken from the literature on BET, we show that CLP, by combining the use of constraints and recursion, allows one to write intuitive and easily understandable test generators. We also show that these generators can be much more efficient than those built using ad-hoc filtering-based test generation tools like Korat.

1 Introduction

The identification of test cases, which is a central task in the testing process, is very often carried out as a manual activity. As a consequence, it is error-prone, it has limited applicability, and can be very expensive (around 50% of the cost of software development). Formal and automated techniques have thus received interest from the testing community because they can be used to develop test suites in a more systematic and affordable way, by enforcing correctness and allowing flexible integration with the considered code coverage criteria.

In this paper we focus on the *bounded-exhaustive testing* [7,26] approach (BET), whose goal is to test a program on *all* the input instances satisfying a given invariant, up to a given bound on their size. The motivation underlying the BET approach is based on the observation that defects, if any, are likely to appear already in small-sized instances of the inputs.

Automated test input generators should be (i) *correct*, that is, they should generate only test input instances which satisfy the considered invariant, and

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 115–131, 2012.
© Springer-Verlag Berlin Heidelberg 2012

(ii) *efficient*, when generating test input candidates and filtering out those which are not admissible, so that they can be applied to large and realistic domains.

Modern software often manipulates input data with complex structure (like trees and graphs) and satisfying non-trivial invariants (like sorting, coloring, depth balancing). The correct and efficient generation of structurally complex inputs is a challenging task because the number of test input candidates can grow very fast, but only a few inputs, which satisfy the desired invariants, are to be selected as admissible. The generation of large and complex test objects is also required by some recent application domains, such as XML documents generation, considered in [18], where an RSS feed parser is tested for HTML injection vulnerabilities, and in [4], where they are used for testing Web Services.

In this paper we propose a framework based on Constraint Logic Programming (CLP) for the systematic development of generators of large sets of structurally complex test data.

Similarly to filtering-based techniques, we adopt a declarative approach which allows us to separate the issue of (i) defining the test input structure and invariants, from that of (ii) generating admissible test input instances. This separation helps improve the correctness of the developed test case generators, because it lets testing engineers write *what* to generate, in a very intuitive and easily understandable way. Efficient CLP search strategies are then used for specifying *how* test instances should be generated.

Although heavy optimizations require in-depth knowledge of CLP techniques, we will show that excellent results can be already obtained by following some simple programming guidelines. In particular, we show that test generators should be written following the so-called *constrain-and-generate* approach, according to which the structural and invariant constraints should be computed first, postponing as much as possible the actual generation of test instances. This allows the CLP computation engine to prune the search space at the symbolic level, avoiding useless executions of the expensive instantiation phase.

We experiment with some examples taken from the literature on BET, and we show that CLP, by combining the use of constraints and recursion, allows us to write test generators which are simpler and more efficient than those built using the ad-hoc test generation tool Korat [29]. However, our focus is not on deriving CLP generators from Korat ones. Rather, we assume that those generators have been derived from a given, abstract, model and we evaluate their efficiency.

Our evaluation shows that modern CLP systems can be used effectively as a core component to construct fast and correct test generators and for more complex test suite development frameworks.

In Sec. 2, we briefly recall the Korat approach and illustrate, as a case-study, a Red-Black Trees generator. In Sec. 3, we introduce our CLP-based approach and we illustrate its expressiveness by providing a clean and strongly declarative definition of a Red-Black Trees generator. We also show how to use some optimization techniques known in the Logic Programming community to obtain even faster generators. Finally, in Sec. 4, we carry out an extensive comparison between our CLP-based approach and that of Korat.

2 The Korat Approach

We now illustrate the Korat approach for writing automated test generators. Korat [29] is a tool for bounded-exhaustive testing of Java programs, which is specifically tailored for the construction of structurally complex test inputs. It allows the generation of complex data structures by providing primitives to populate an object domain, to initialize objects, and to set links among them.

Given a data structure definition, Korat requires (1) a so-called finitization method, which defines the bounds of the search space, and (2) a method `repOK()`, which specifies the data structure invariant. Korat performs a systematic search of the program input space, avoiding the full exploration of failing regions and the generation of isomorphic structures (i.e. equal modulo Java objects identity). Details of the optimizations used in the search can be found in [5,25,29].

We now illustrate how the Korat approach works by applying it for writing a test input generator for Red-Black Trees.

Example 1. A Red-Black Tree [8] is a binary search tree where each node has two labels: a *color*, which is either red or black, and an integer, called *key* (for the purpose of test generation, node values are abstracted away in the definition of the data structure). Therefore, it satisfies the following type equation:

```
Color ::= 0 | 1
Key   ::= ... | -1 | 0 | 1 | ...
Tree  ::= e | Color x Key x Tree x Tree
```

where `0` and `1` denote *red* and *black*, respectively, and `e` denotes the emtpy tree. A Red-Black Tree must also satisfy the following three invariants:

($\mathbf{I}_1$) no red node has a red child,
($\mathbf{I}_2$) every path from the root to a leaf has the same number of black nodes, and
($\mathbf{I}_3$) for every node n, all the nodes in the left (respectively, right) subtree of n, if any, have keys which are smaller (respectively, bigger) than the key labeling n.

Since Red-Black Trees enjoy a weak form of balancing, operations such as inserting, deleting, and finding values are more efficient, in the worst-case, than in ordinary binary search trees.

We consider the Red-Black Tree generator implementation taken from the Korat repository[1]. The `RedBlackTree` class, shown in Fig. 1, uses an internal class `Node` defining the generic node of the Red-Black Tree data structure. The `Node` class has integer attributes `key`, `color` and `value`, and attributes `left`, `right` and `parent` of type `Node`.

The finitization method `finRedBlackTree`, shown in Fig. 2, is used to define the search space for generating the test candidates. It accepts the following arguments: `numEntries`, denoting the number of objects of class `Node` which can be used for building the Red-Black Tree, `minSize` and `maxSize`, denoting the minimum and maximum number of nodes of the tree (`maxSize` is expected to be

[1] https://korat.svn.sourceforge.net/

```
public class RedBlackTree {
    private Node root = null;
    private int size = 0;
    private static final int RED = 0;
    private static final int BLACK = 1;

    public static class Node  {
        int key, value;
        Node left = null, right = null, parent;
        int color = BLACK;
    }

    METHODS...
}
```

Fig. 1. Red-Black Trees java class

```
public static IFinitization finRedBlackTree
    (int numEntries, int minSize, int maxSize, int numKeys) {

        IFinitization f          = FinitizationFactory.create(RedBlackTree.class);
        IClassDomain entryDomain = f.createClassDomain(Node.class, numEntries);
        IObjSet entries          = f.createObjSet(Node.class, true);
        entries.addClassDomain(entryDomain);

        IIntSet sizes = f.createIntSet(minSize, maxSize);
        IIntSet keys = f.createIntSet(-1, numKeys - 1);
        IIntSet values = f.createIntSet(0);
        IIntSet colors = f.createIntSet(0, 1);

        f.set("root",        entries);   f.set("size",       sizes);
        f.set("Node.left",   entries);   f.set("Node.color", colors);
        f.set("Node.right",  entries);   f.set("Node.key",   keys);
        f.set("Node.parent", entries);   f.set("Node.value", values);

        return f;
    }
```

Fig. 2. Red-Black Trees finitization method

not bigger than `numEntries`), and `numKeys`, denoting the upper bound for keys values (with lower bound `0`). The methods `createClassDomain`, `createObjSet`, and `addClassDomain` populate the object domain, while the calls to the method `createIntSet` populate the integer domains for tree sizes and node keys, values, and colors, respectively. Finally, the method `set` is used to map class attributes to the appropriate domains (nodes or integers), which will be used by Korat during the candidate instantiation phase. Notice that, though `color` and `value` are declared as integers, `color` can only take values in $\{0, 1\}$ and `value` is a constant, (so, in practice, values are abstracted away).

The method `repOK()`, which for lack of space is not shown here, is an imperative specification of the Red-Black Tree data structure invariants $\mathbf{I}_1$, $\mathbf{I}_2$, and $\mathbf{I}_3$. It is used by Korat to filter, among the many candidate trees generated, only those that satisfy the Red-Black Trees invariants.

3 The CLP-Based Approach

Logic Programming [24] is a declarative programming paradigm based on a computational interpretation of resolution-based first-order theorem proving. Sets of

formulas can be regarded as programs and proof search can be used as a general-purpose problem solving mechanism.

Constraint Logic Programming (briefly, CLP) [20] extends Logic Programming with constraints, which are managed by fast, domain-specific, constraint solvers. During the proof search process, constraints are collected in a store which is required to be consistent at each step, and the problem solving process amounts to reducing the initial problem to a satisfiable set of constraints. Among several other applications, CLP has shown to be well suited for encoding and solving combinatorial problems [11].

Let us now briefly recall the CLP framework and its operational semantics, for more details we refer the reader to [20]. Let Σ be a logic language signature $\Sigma = \langle \mathcal{F}, \mathcal{V}, \Pi \cup \Pi_C \rangle$, where $\mathcal{F}$ is a finite set of function and constant symbols, $\mathcal{V}$ is a denumerable collection of variables, $\Pi \cup \Pi_C$ is a finite set of predicate symbols, where Π and Π_C are disjoint sets. *Atoms* are of the form $p(s_1, \ldots, s_n)$ where p is a predicate symbol in Π and s_i's are $(\mathcal{F}, \mathcal{V})$-terms, A *constraint* is a first-order formula over $\mathcal{F}$, $\mathcal{V}$, and Π_C, (typically, a conjunction such as `X#>3,X+Y#<0`). In logic programming notation, a comma denotes a conjunction and the symbol `:-` denotes the implication $\leftarrow$. Strings denote variables if they start with a capital letter and constants, otherwise. Comments are started by `%`. When variables need not be named, they are replaced by `_`. A CLP *program* P over Σ is a finite set of *clauses* of the form:

$$H \ :- \ c, \ A_1, \ldots, A_m.$$

where c is a constraint, and H and A_i's are atoms.

A CLP system computes answers to user queries (called *goals*) of the form $c, A_1, \ldots, A_k$ against a program P, where c is a constraint and $A_1, \ldots, A_k$ is a finite conjunction of atoms. Given a program P, an *answer* to a goal $c, A_1, \ldots, A_k$ is a substitution ϑ such that $\forall (A_1, \ldots, A_k)\vartheta$ is a logical consequence of P and $c\vartheta$ is satisfiable in the constraints theory. We denote by $[\![G]\!]_P$ the set of all the answers to the goal G against the program P. We will feel free to omit the subscript P whenever the intended program is clear from the context. An answer ϑ is *ground* whenever $(c, A_1, \ldots, A_k)\vartheta$ contains no variables.

The programming paradigm of CLP, sometimes referred to as constrain-and-generate [27], is structured mainly in two phases: first constraints are added to the constraint store and checked for consistency (*constrain*), then the solver instantiates variables to produce actual values that satisfy the constraints in the store (*generate*). Since in the constrain phase the constraint store is checked for consistency at each modification, several unsatisfiable cases are rejected at the symbolic level. When the problem is satisfiable, the search for a satisfying substitution is committed to a dedicated solver. This behavior is quite different from the generate-and-test approach of Korat and other filter-based techniques [14,22,29]. In particular, we focus on Constraint Logic Programming over Finite Domains (CLP(FD) [27]) where constraints are linear arithmetic equalities and inequalities on variables ranging over finite integer domains.

The FD comparison predicates in Π_C are `#=`, `#>`, `#>=`, `#<`, `#=<`. The function signature $\mathcal{F}$ extends the set $\{$`+`, `-`, `*`$\} \cup \mathbb{Z}$ and the set $\{$`[]`, `[_|_]`$\}$ of list constructors.

Predicates and function symbols in FD have the standard interpretation over $\mathbb{Z}$. We assume to have the built-in predicates: `fd_domain(Vs,Min,Max)`, that constrains all the variables in the list `Vs` to range over $[\texttt{Min},\ldots,\texttt{Max}] \subset \mathbb{Z}$, and `fd_labeling(Vs)`, that forces each variable in `Vs` to assume a concrete value among those allowed by the current constraint store (an additional `Settings` argument is contemplated, for configuring the search process).

We propose the following instantiation of the constrain-and-generate paradigm for the implementation of efficient (filter-based) test case generators:

```
gen_structure(Struct,P1,...,Pk) :-
    % Preamble                                  (constrain)
      ...definition of the variables in Vars and their domain,
    % Symbolic Definition                       (constrain)
      structure(Struct,P1,...,Pk,Vars),         % data structure shape
      ...filters,                               % invariants
    % Instantiation                             (generate)
      fd_labeling(Vars).
```

The semantics of this predicate is that, for any given value of the parameters `P1`,...,`Pk` we build a structure `Struct` that satisfies the desired invariants.

The `Preamble` contains the definition of the set `Vars` of the required variables and their domains. The `Symbolic Definition` phase contains: (i) a call to a predicate `structure` which defines, by structural induction, the data structure shape (e.g. list, tree, graph) using variables in `Vars` as placeholders for values, and (ii) a `filters` part which contains a conjunction of predicates that assert constraints among the variables in `Vars`. Finally, the `Instantiation` phase invokes the built-in labeling mechanism, possibly using problem-specific settings to configure the search strategy. The solver tries to minimize backtracking on assignments and each assignment triggers a deterministic propagation towards related variables, which reduces their domain and the set of future choices.

Concerning point (i) of the `Symbolic Definition` phase above, in this paper we consider simple tree-like structures, where Logic Programming can show the advantage of the built-in unification mechanism. Graph-like structures are a bit more involved to deal with and one can rely on a classical incidence/adjacency-matrix representation or rely on more sophisticated decompositions [31].

Let `gen_structure` be a generator predicate and let `p1`,...,`pk` be concrete values for the parameters, the set $\mathcal{T}$ of *test cases* induced by the generator is $\mathcal{T} = \{\texttt{Struct}\vartheta \mid \vartheta \in [\![\texttt{gen\_structure(Struct,p1,...,pk)}]\!]\}$. Note that, due to the `Instantiation` phase, we have that $\mathcal{T}$ contains only ground test cases.

3.1 Red-Black Trees

Let us now illustrate the CLP-based specification of a Red-Black Tree generator. This generator is parameterized, as for Korat, by the maximum and minimum tree size (defined as the number of its nodes), and by the maximum value for the keys. Since we do not generate nodes beforehand, but on demand, we do not need an extra parameter for counting the number of nodes, as Korat does. The following clause defines the predicate `rbtree`:

```
rbtree(T,MinSize,MaxSize,NumKeys) :-
    % Preamble
      MinSize#=<S, S#=<MaxSize, fd_labeling([S]),
      varlist(S,Keys), varlist(S,Colors), Max#=NumKeys-1,
      fd_domain(Keys,0,Max), fd_domain(Colors,0,1),
    % Symbolic Definition
      lbt(T,S,Keys,[]),                              % data structure shape
      ci(T,Colors,[]), pi(T,_), ordered(T,0,Max),    % filters
    % Instantiation
      fd_labeling(Keys), fd_labeling(Colors).
```

where the predicate `varlist(N,L)`, is used for constructing a list `L` of `N` fresh variables. Given the ground (non-negative) input integers `minSize`, `maxSize`, and `numKeys`, the set ⟦`rbtree(T,minSize,maxSize,numKeys)`⟧ contains all the red-black trees of size ranging in {`minSize`,...,`maxSize`}, with keys ranging in {`0`,...,`numKeys-1`}.

The first line of the Preamble chooses a tree size value `S`. Then, two lists of (distinct) variables are defined, `Keys` and `Colors`, with the corresponding domains, {`0`,...,`NumKeys-1`} and {`0`,`1`}, respectively. These variables are placed along the tree structure in the `Symbolic Definition` part by the predicate `lbt`, which defines (2-)labeled binary trees by structural induction:

```
1. lbt(         e,S,    Ks, Ks) :- S#=0.
2. lbt(t(_,K,L,R),S,[K|Ks],NKs) :- S#>=1, SL#>=0, SR#>=0,
                                   NS#=S-1, fdsum(NS,SL,SR),
                                   lbt(L,SL,Ks,TKs), lbt(R,SR,TKs,NKs).
```

The first argument is either the constant `e` denoting the empty tree or a term `t(C,K,L,R)` denoting a (non-empty) tree with left subtree `L`, right subtree `R`, and whose root node is labeled with color `C` and key `K`. The second argument is the size of the tree (the number of nodes) which, in clause `2`, is at least `1` and is non-deterministically split by the `fdsum` predicate into a pair of non-negative integers `SL` and `SR` denoting the size of the left and right subtrees, respectively, such that $\texttt{S} = \texttt{SL} + \texttt{SR} + 1$. The left and right subtrees are then constructed recursively. Note that the variables in `Keys` are placed in distinct nodes.

The predicate `ci` (for *color invariant*) encodes the invariant $(\mathbf{I}_1)$ of Sec. 1 and is defined as follows:

```
4. ci(         e,    Cs, Cs).
5. ci(t(C,_,L,R),[C|Cs],NCs) :- C#=1,                    % root is black
                                ci(L,Cs,TCs), ci(R,TCs,NCs).
6. ci(t(C,_,L,R),[C|Cs],NCs) :- C#=0,                    % root is red
                                not_redroot(L), not_redroot(R),
                                ci(L,Cs,TCs), ci(R,TCs,NCs).
7. not_redroot(         e).
8. not_redroot(t(C,_,_,_)) :- C#=1.                      % root is black
```

Note that the color variables are placed in distinct nodes. In clause `6` the color invariant is enforced by testing the color of the roots of the left and right subtrees.

The predicate `pi` (for *path invariant*) encodes the invariant $(\mathbf{I}_2)$ of Sec. 1 and is defined as follows:

```
 9. pi(e          ,C) :- C#=0.
10. pi(t(C,_,L,R),D) :- ND#>=0, D#=ND+C, pi(L,ND), pi(R,ND).
```

The semantics of `pi` is the following: for a given tree `t`, `pi(t,d)` holds if and only if on all root-to-leaf paths in `t` there are exactly `d` black nodes. In this case, we say that `d` is the value of the *black-nodes counter* of `t`. Note that if a tree is empty then its black-nodes counter is `0`, otherwise, the black-nodes counter is computed by adding the 'color' of the root (i.e. `0`, if red, and `1`, if black) to the black-nodes counter of (both) its subtrees (that must have the same value).

Finally, the predicate `ordered` defines the invariant $(\mathbf{I}_3)$ in Sec. 2, concerning the ordering of the keys, and is defined as follows:

```
11. ordered(          e,  _,  _).
12. ordered(t(_,N,L,R),Min,Max) :- Min #=< N, N #< Max, M #= N+1,
                                   ordered(L,Min,N), ordered(R,M,Max).
```

There is a simple correspondence between the CLP(FD) generator and the Korat generator. The predicate `lbt` is essentially a definition of the tree data type, as given in Sec. 1. This allows us to start from *trees* rather than from *graphs*, which is a significant advantage over Korat, which must perform the acyclicity check. The filter predicates `ci`, `pi`, and `ordered`, are very similar to the Korat code for `repOk()`, which can be retrieved in the Korat repository. Note that `repOk()` executes three tests: (i) acyclicity, using the `repOkAcyclic()` procedure, (ii) invariants $(\mathbf{I}_1)$ and $(\mathbf{I}_2)$ using the `repOkColors()` procedure, and (iii) ordering invariant $(\mathbf{I}_3)$, using the `repOkKeysAndValues()` procedure. The `repOkColors()` procedure returns `true` iff the goal `ci(T,Colors,[]),pi(T,_)` succeeds and `repOkKeysAndValues()` returns `true` iff the goal `ordered(T,0,Max)` succeeds.

We compared the efficiency of the CLP-based Red-Black Trees generator with the Korat-based one. Following the approach of [25], we consider the 'canonical set' $[\![$`rbtree(T,s,s,s)`$]\!]$, which is the set of all red-black trees of `s` nodes and keys ranging in {`0`,...,`s-1`}.

The results of this comparison are summarized in Fig. 3 (columns 1-4,9) and show that the CLP-based Red-Black Trees generator, which has been run using two different CLP systems, SICStus[2] and GNU Prolog[3], is very efficient with respect to the Korat generator. Further details and discussion about the experimental evaluation can be found in Sec. 4.

3.2 Optimizations

In this section we discuss some optimization techniques available in the field of logic programming that can be used for building even more efficient test case generators. Simple optimizations can be done by using information coming from groundness analysis [19] and determinacy checking [23], that allow us to determine when a predicate has *at most* one answer. In order to improve determinism, one can put such predicates in the `Preamble` so that they are evaluated only once (we applied this optimization to the Heaparrays example discussed in Sec. 4).

[2] http://www.sics.se/sicstus/
[3] http://www.gprolog.org/

Size	Trees	Time						
		Original		Partially evaluated		Synchronized		Korat
		GNU	SICStus	GNU	SICStus	GNU	SICStus	Korat
6	20	0	0.01	0	0	0	0	0.47
7	35	0.01	0.06	0	0.03	0	0.01	0.63
8	64	0.02	0.20	0.01	0.07	0	0.03	1.49
9	122	0.08	0.71	0.04	0.28	0	0.06	4.51
10	260	0.29	2.60	0.16	0.98	0.01	0.13	21.14
11	586	1.07	9.52	0.58	3.55	0.03	0.26	116.17
12	1296	3.98	35.19	2.17	13.00	0.06	0.54	-
13	2708	14.85	131.13	8.15	47.90	0.12	1.17	-
14	5400	55.77	-	30.73	177.63	0.26	2.49	-
15	10468	-	-	115.59	-	0.55	5.35	-
20	279264	-	-	-	-	25.90	-	-

Fig. 3. Comparison of Red-Black Trees generators. The table reports the size of the red-black trees (column 1), the number of computed red-black trees (2), the time, in seconds, for generating all the structures running the original CLP generator of Sec. 3 using GNU Prolog and SICStus (3-4), the partially evaluated CLP generator of Sec. 3.2 (5-6), the synchronized CLP generator of Sec 3.2 (7-8), and Korat (9). Zero means less than 10 ms and (-) means more than 200 seconds.

A more sophisticated technique is Program Transformation, that is a semantics-preserving program rewriting technique which is applicable, among other languages, also to Constraint Logic Programming [12]. It can be used to optimize and synthesize programs [13] and it is based on rewriting *rules*, whose application is directed by *strategies*. Here we use Program Transformation to optimize our Red-Black Trees generator. The transformations are performed by using our transformation tool MAP [1].

Step 1. We perform a *partial evaluation* [21] of `lbt`, `ci`, `pi`, and `ordered` w.r.t. their first argument and the term domain `T ::= e | t(_,_,T,T)`, by using the *unfolding* rule (which unrolls predicate calls by using their definitions). For reasons of space, we show the effect of this transformation only for predicate `pi`:

```
pi(                                    e,0).
pi(t(C,_,              e,              e),D) :- D#=C.
pi(t(C,_,              e,t(Cr,Lr,Kr,Rr)),D) :- D#=C, pi(t(Cr,Kr,Lr,Rr),0).
pi(t(C,_,t(Cl,Ll,Kl,Rl),               e),D) :- D#=C, pi(t(Cl,Kl,Ll,Rl),0).
pi(t(C,_,t(Cl,Ll,Kl,Rl),t(Cr,Lr,Kr,Rr)),D) :- ND#>=0, D#=ND+C,
                                               pi(t(Cl,Kl,Ll,Rl),ND),
                                               pi(t(Cr,Kr,Lr,Rr),ND).
```

Performance improvement is due to the instantiation of the heads of the clauses which reduces unification successes and prunes the search tree, at the cost of an increase in the number of clauses. The performance of the partially evaluated generator w.r.t. that of the original one is shown in Fig. 3. ■

The execution of a conjunction of atoms that share a variable ranging over a term domain (such as `T ::= e | t(_,_,T,T)` above) may be inefficient and even non-terminating, under the standard depth-first CLP evaluation strategy. In the Red-Black Trees generator the predicates `lbt`, `ci`, `pi`, and `ordered` share the tree variable `T`. Each tree is traversed four times. Indeed, there are productive

interactions between those predicates that we do not take advantage of. Namely, the check of the path invariant of `pi` may produce earlier failure for a given distribution of the nodes of `T`. Similarly for the choice of nodes colors in `ci` and the lengths of the paths computed in `T`.

Step 2. We apply a program transformation strategy that optimizes programs to avoid *multiple traversals* of a data structure [30]. This strategy replaces a conjunction of atoms G by a new atom that performs a synchronized execution on the shared variables, and produces the same results. The effect is to obtain (i) a single traversal of terms in the shared domain and (ii) a better interaction among the constraints of each predicate in G to promote as early as possible failures/successes. We take advantage of the previous partial evaluation step by applying the current transformation to the partially evaluated versions of `lbt`, `ci`, `pi`, and `ordered`. We don't have enough space here to discuss the details of this transformation, but we give here a sketch of it.

In goal G=`lbt(T,S,Keys,[]),ci(T,Colors,[]),pi(T,_),ordered(T,0,Max)` (occurring in the definition of `rbtree`) the atoms share the variable `T`. We introduce a new predicate `sync` (which stands for *synchronized*) defined by the following clause:

```
def. sync(T,S,Keys,NewKeys,Colors,NewColors,Min,Max,D) :-
          lbt(T,S,Keys,NewKeys),
          pi(T,D), ci(T,Colors,NewColors), ordered(T,Min,Max).
```

where the goal G is abstracted to one containing only variables. Then we derive a recursive definition of `sync` in two steps. First, we perform a partial evaluation, by *unfolding*, and we obtain (after rearrangement of the constraints to the left of the other atoms) the following new definition:

```
sync(                          e,A,    B,B,    C,C,_,_,0) :- A#=0.
sync(t(A,B,         e,         e),C,[B|D],D,[A|E],E,F,G,A) :- C#=1, F#=<B, B#<G.
sync(t(A,B,         e,t(C,D,E,F)),G,[B|H],I,[A|J],K,L,M,A) :-
      G#>=2, O#=G-1, A+C#>0, L#=<B, B#<M, P#=B+1,
      lbt(t(C,D,E,F),O,H,I), pi(t(C,D,E,F),0), ci(t(C,D,E,F),J,K), ordered(t(C,D,E,F),P,M).
sync(t(A,t(B,C,D,E),F,         e),G,[F|H],I,[A|J],K,L,M,A) :-
      G#>=2, O#=G-1, A+B#>0, L#=<F, F#<M,
      lbt(t(B,C,D,E),O,H,I), pi(t(B,C,D,E),0), ci(t(B,C,D,E),J,K), ordered(t(B,C,D,E),L,F).
sync(t(A,F,t(B,C,D,E),t(G,H,I,J)),K,[F|L],M,[A|N],O,P,Q,R) :-
      K#>=3, S#=K-1, T#>0, U#>0, A+B#>0, A+G#>0, V#>=0,
      R#=V+A, P#=<F, F#<Q, W#=F+1, fdsum(S,T,U),
      lbt(t(B,C,D,E),T,L,X), pi(t(B,C,D,E),V), ci(t(B,C,D,E),N,Y), ordered(t(B,C,D,E),P,F),
      lbt(t(G,H,I,J),U,X,M), pi(t(G,H,I,J),V), ci(t(G,H,I,J),Y,O), ordered(t(G,H,I,J),W,Q).
```

Next, by *folding*, we replace goals that match the body of the clause `def` by the corresponding instances of the head and we obtain the following clauses:

```
1. sync(                          e,A,    B,B,    C,C,_,_,0) :- A#=0.
2. sync(t(A,B,         e,         e),C,[B|D],D,[A|E],E,F,G,A) :- C#=1, F#=<B, B#<G.
3. sync(t(A,B,         e,t(C,D,E,F)),G,[B|H],I,[A|J],K,L,M,A) :-
        G#>=2, O#=G-1, A+C#>0, L#=<B, B#<M, P#=B+1,
        sync(t(C,D,E,F),O,H,I,J,K,P,M,0).                    % replacement
4. sync(t(A,F,t(B,C,D,E),         e),G,[F|H],I,[A|J],K,L,M,A) :-
        G#>=2, O#=G-1, A+B#>0, L#=<F, F#<M,
        sync(t(B,C,D,E),O,H,I,J,K,L,F,0).                    % replacement
5. sync(t(A,F,t(B,C,D,E),t(G,H,I,J)),K,[F|L],M,[A|N],O,P,Q,R) :-
        K#>=3, S#=K-1, T#>0, U#>0, A+B#>0, A+G#>0, V#>=0,
```

```
R#=V+A, P#=<F, F#<Q, W#=F+1, fdsum(S,T,U),
sync(t(B,C,D,E),T,L,X,N,Y,P,F,V),                    % replacement
sync(t(G,H,I,J),U,X,M,Y,O,W,Q,V).                    % replacement
```

The definition of the predicate `sync` is now made of the clauses $\{1, 2, 3, 4, 5\}$. The final step of this transformation consists in a further application of *folding* which replaces the goal G in the definition of `rbtree` (which is an instance of the body of clause `def`) by the goal `sync(T,S,Keys,[],Colors,[],0,MaxNat,_)`. The performance improvements of the synchronized generator w.r.t. the partially evaluated generator and the original one are shown in Fig. 3. ■

By correctness of the transformation rules and of the transformation strategy, we are ensured that the two generators are equivalent, in terms of their least Herbrand models [24] and, thus, define the same set of test cases.

4 Experimental Evaluation

In order to measure the effectiveness of our technique, we retrieved the code of several test case generators from the Korat repository and we encoded the corresponding CLP(FD)-based generators (their code can be found in the technical report [32]).

We considered generators for: (i) sorted lists of integers, (ii) an array-based representation of the heap data structure, (iii) integer-labeled search trees, and (iv) an array-based representation of disjoint partitions of a set. For disjoint sets, we found it difficult to reverse-engineer the exact specification from the Java code in the Korat repository and, thus, we decided to recode the Korat version from scratch w.r.t. an abstract model (the code can be found in [32]).

In all our experiments, we found that the performance of CLP-based test generators is always much better than the performance of the corresponding Korat generators. We should stress here some points discussed in Sec. 3. Korat builds data structures starting from a domain of graphs. In logic programming, on the contrary, terms are first-class objects, and graphs are represented through terms (see standard textbooks encodings). This is an advantage of logic programming, which allows us to choose the most adequate and simple primitive data structure. In contrast, Korat generates trees through the computationally expensive process of generating directed graphs and filtering the acyclic ones.

All the experiments were performed on an Intel Core 2 Duo E7300 2.66 GHz under the Linux operating system, and the timings were collected using built-in CLP and Java statistics predicates.

In particular, the CLP timings were measured by exploring the whole search space through a call of the form `gen_structure(Struct,p1,...,pk),fail` for the parameter values of interest. The idea is to exploit the CLP backtracking mechanism to explore each success of `gen_structure`, while trying to succeed. For every success of the subgoal `gen_structure(Struct,p1,...,pk)`, a concrete structure is generated. Since we are not interested in keeping all the structures in memory (which could be unmanageable), each structure is deleted as soon as it has been constructed. Due to the presence of the `fail` built-in, the

whole goal fails, and, thus, the CLP system backtracks and tries to find another solution to `gen_structure(Struct,p1,...,pk)`. The computation terminates after all the structures have been generated. At this stage, we are not yet addressing the issue of converting the CLP structures to proper Java objects. This issue will be briefly discussed in Sec. 5.

We selected two different CLP(FD) systems for running our CLP-based test generators: SICStus, for its diffusion and industrial strength, and GNUProlog, for its efficient compilation. We found that, in our experiments, GNUProlog outperforms SICStus, due to its efficient compilation. However, we chose to keep also the SICStus timings, because they revealed to be much more stable w.r.t. different encodings we experimented with (such as moving term comparison constraints from the head to the body). Therefore, SICStus seems to be more reliable in a setting where the user is not aware of the inner evaluation mechanism and cannot take advantage of it, while being still efficient.

In Fig. 4 we show the tables containing the timings for GNU Prolog, SICStus Prolog, and Korat, with a timeout of 200 seconds. The memory consumption of the CLP generators is negligible and grows very slowly on the size of the structures (as in Korat) so we did not report it.

The results show that the CLP(FD)-based approach outperforms Korat in all the examples we considered. In some examples the CLP(FD)-based approach allowed us to explore a much larger input domain.

We did not explore different tunings of the CLP(FD)-solver, other than the default ones, which revealed to be already satisfactory. However, more complex problems (involving, for example, conditions based on minimization) may benefit of the many built-in predicates implementing more sophisticated solution search algorithms [27].

These promising results allow us to draw first conclusions on the validity of our CLP(FD)-based approach. The CLP(FD) encoding of generators requires no more

Size	Sorted Lists	Time		
		GNU	SICStus	Korat
8	6435	0.00	0.01	0.61
9	24310	0.00	0.05	1.08
10	92378	0.02	0.17	1.83
11	352716	0.09	0.65	6.37
12	1352078	0.36	2.51	24.95
13	5200300	1.40	9.63	125.73
14	20058300	5.40	37.35	-
15	77558760	21.16	143.79	-
16	300540195	82.22	-	

Size	Heaparrays	Time		
		GNU	SICStus	Korat
6	13139	0.00	0.01	0.30
7	117562	0.03	0.15	0.86
8	1005075	0.17	1.27	3.41
9	10391382	1.66	12.65	34.103
10	111511015	17.57	134.26	-

Size	Search Trees	Time		
		GNU	SICStus	Korat
7	429	0.01	0.03	0.87
8	1430	0.02	0.11	4.43
9	4862	0.08	0.43	33.99
10	16796	0.28	1.67	-
11	58786	1.11	6.53	-
12	208012	4.43	25.42	-
13	742900	17.68	100.09	-
14	2674440	70.75	-	-

Size	Disjoint Sets	Time		
		GNU	SICStus	Korat
6	203	0.00	0.00	1.60
7	877	0.00	0.01	37.10
8	4140	0.01	0.06	-
9	21147	0.10	0.28	-
10	115975	0.61	1.58	-
11	678570	3.90	9.83	-
12	-	26.63	65.29	-
13	-	189.42	-	-

Fig. 4. Generators performance evaluation

ingenuity that the Korat encoding. On the contrary, we claim that correctness is a natural outcome of this approach and the programmer confidence in the developed generators is greatly increased. Furthermore, while being more declarative, this approach is also much more efficient and deserves to be further investigated for better integration into real-world testing frameworks.

5 Related Work and Conclusions

Constraint-based techniques have been widely used in the field of test case generation, since pioneering work in [10]. Early use of CLP for test generation can be found in the tool ATGen [28], developed for testing Spark ADA programs.

Several approaches using constraints are white-box and aim at the automatic extraction of CLP test generators from program source code, according to given coverage criteria. Moreover, most of them are not directly concerned with the efficiency of bounded-exhaustive generation of complex inputs.

In particular, in [9], white-box testing of an imperative language with pointers and heap is performed by symbolic execution of a small-step operational semantics in CLP, guided by coverage criteria. This approach can generate pointer-based data structures, at the expense of defining ad-hoc constraints solvers for the structures considered (mainly lists). Further work on white-box testing has been done in [6,17] for the generation of heap-allocated data structures, following a fixed coverage criteria for the choice of the test cases. The work in [16] presents a technique for white-box testing of object-oriented programming languages, which is more general and language independent than previous ones. Indeed, test case generators are obtained by partial evaluation of a language interpreter w.r.t. a given program.

The declarative approach has also been adopted by test generation tools such as Korat [29], which has been used in our experimental evaluation, UDITA [14], and TestEra [22]. These tools are quite efficient in practice but require careful implementations of clever ad-hoc backtracking mechanisms and search strategies, which are either built-in (like non-deterministic choice) or easily implementable in standard CLP systems. Lazy instantiation strategies in UDITA [14] can be seen as a particular CP solution strategy. Moreover, these tools are language-specific and they are not easily adaptable to other languages. Their integration with homogeneous but more general testing environments such as JPF [33] can be expensive and lead to suboptimal performance w.r.t. their original version [15].

In contrast, the proper interaction between a CLP test generator and, for example, a Java-based testing environment can be achieved either by using a bidirectional Java-Prolog interface, provided by most CLP systems, or by using an intermediate string representation of CLP data structures combined with Java's (de)serialization. In the latter case, for example, one could (i) generate XML encodings of CLP terms, and (ii) use libraries like XStream[4] or Simple[5] for constructing Java objects from XML. The problem of obtaining XML from CLP data

[4] http://xstream.codehaus.org
[5] http://simple.sourceforge.net

structures can be solved, once and for all, by writing a single, universal, translator which constructs XML elements while recursively traversing a generic CLP term. We expect the CLP to Java translation to add negligible overhead. It should be noted, indeed, that such translation would be triggered only for structures which are of actual interest for testing (in contrast with the partial building of the Java objects and their destruction, if unsatisfactory, as in Korat [5]).

Efficiency aspects are considered in [34,35] where the process of generating the shape of the data structure is separated from that of generating proper values for data. In our approach, this separation is achieved transparently by following the constrain-and-generate programming approach of CLP, which prescribes the invocation of the instantiation mechanism only after all the constraints have been generated. Many unfeasible structures can, therefore, be eliminated at the symbolic level by constraint consistency checks.

In this paper we focus on showing that CLP can be used as a core component for efficient test case generators of complex input data. In contrast to some of the above-mentioned techniques, our method does not start from source code and has been designed for black-box testing. It does not require the development of ad-hoc constraint solvers or search strategies, but it leverages commonly available CLP systems and libraries. However, strategies can easily be customized, if needed, and further LP instruments (some of which discussed in Sec. 3.2) are available to optimize the generators obtained according to our general scheme.

For example, in [7] it is shown that the use of BET for verifying large systems is feasible and provides effective results, but requires significant effort to be tuned and combined with abstraction techniques to reach the generation of useful test sets. For this purpose, our approach could easily benefit from decades of research on program analysis and abstract interpretation of constraint logic programs.

This work can be extended along several directions.

In [3] the Korat engine is modified to reduce the search space, by trying to skip structures which are in the same equivalence class of already considered structures. A similar issue is addressed in [4] for partition-based testing. Although we did not experiment on this subject (because the focus was on building all the structures) we believe that this optimization can be easily integrated in CLP by generating exactly one or a small set of witnesses per equivalence class.

There are several issues that deserve further study. Among these, we plan to explore the relationship between constraint solution strategies and test coverage criteria. Indeed, one may be interested into exploring the set of possible structures according to an ordering, parameterized by a given coverage criteria.

Furthermore, while in this paper we focused on model-based input generation only, we believe that the CLP approach can be successfully applied also for generating test oracles which can be used for verifying the post-conditions of the methods under test, since CLP generators can also be used as *acceptors*.

Regarding the application of program transformation and other optimization techniques, we plan to develop fully automatic optimization techniques tuned for this specific problem and for our CLP-based approach. A further application

domain of program transformation is the automated extraction of test generators from (formal) specifications.

In conclusion, we believe that, due to its inherent symbolic execution mechanism, Constraint Logic Programming has a promising application field in test-case generation. CLP provides a highly declarative language and ensures efficiency by using dedicated constraint solvers and heuristics. On the basis of the results presented in this work we claim that correctness and efficiency of generators can take advantage from CLP-based techniques, especially in the case of complex input data.

Acknowledgements. We would like to thank Maurizio Proietti for many stimulating conversations. We would also like to thank the anonymous referees for their constructive comments on a preliminary version of this paper.

References

1. The MAP transformation system (1995-2012), http://www.iasi.cnr.it/~proietti/system.html.
2. ICST 2009, Second International Conference on Software Testing Verification and Validation, April 1-4. IEEE Computer Society, Denver (2009)
3. Aguirre, N., Bengolea, V.S., Frias, M.F., Galeotti, J.P.: Incorporating Coverage Criteria in Bounded Exhaustive Black Box Test Generation of Structural Inputs. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 15–32. Springer, Heidelberg (2011)
4. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: Ws-taxi: A wsdl-based testing tool for web services. In: ICST [2], pp. 326–335 (2009)
5. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on java predicates. In: ISSTA, pp. 123–133 (2002)
6. Charreteur, F., Botella, B., Gotlieb, A.: Modelling dynamic memory management in constraint-based testing. Journal of Systems and Software 82(11), 1755–1766 (2009)
7. Coppit, D., Yang, J., Khurshid, S., Le, W., Sullivan, K.J.: Software assurance by bounded exhaustive testing. IEEE Trans. Software Eng. 31(4), 328–339 (2005)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)
9. Degrave, F., Schrijvers, T., Vanhoof, W.: Towards a Framework for Constraint-Based Test Case Generation. In: De Schreye, D. (ed.) LOPSTR 2009. LNCS, vol. 6037, pp. 128–142. Springer, Heidelberg (2010)
10. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Trans. Software Eng. 17(9), 900–910 (1991)
11. Dovier, A., Formisano, A., Pontelli, E.: An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. J. Exp. Theor. Artif. Intell. 21(2), 79–121 (2009)
12. Fioravanti, F., Pettorossi, A., Proietti, M.: Transformation Rules for Locally Stratified Constraint Logic Programs. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in CL. LNCS, vol. 3049, pp. 291–339. Springer, Heidelberg (2004)
13. Fioravanti, F., Pettorossi, A., Proietti, M., Senni, V.: Program transformation for development, verification, and synthesis of programs. Intelligenza Artificiale 5(1), 119–125 (2011)

14. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in udita. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) ICSE (1), pp. 225–234. ACM (2010)
15. Gligoric, M., Gvero, T., Lauterburg, S., Marinov, D., Khurshid, S.: Optimizing generation of object graphs in java pathfinder. In: ICST [2], pp. 51–60
16. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Test case generation for object-oriented imperative languages in clp. TPLP 10(4-6), 659–674 (2010)
17. Gotlieb, A., Botella, B., Rueher, M.: A CLP Framework for Computing Structural Test Data. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 399–413. Springer, Heidelberg (2000)
18. Wang, D., Chang, H.-Y., Ly-Gagnon, M., Hoffman, D.: Grammar based testing of html injection vulnerabilities in rss feeds. In: ICST [2], pp. 105–110 (2009)
19. Howe, J.M., King, A.: Efficient groundness analysis in prolog. Theory Pract. Log. Program. 3, 95–124 (2003)
20. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. Log. Program. 19/20, 503–581 (1994)
21. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, Inc., Upper Saddle River (1993)
22. Khalek, S.A., Yang, G., Zhang, L., Marinov, D., Khurshid, S.: Testera: A tool for testing java programs using alloy specifications. In: Alexander, P., Pasareanu, C.S., Hosking, J.G. (eds.) ASE, pp. 608–611. IEEE (2011)
23. Kriener, J., King, A.: RedAlert: Determinacy Inference for Prolog. Theory and Practice of Logic Programming 11(4-5), 537–553 (2011)
24. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer (1987)
25. Marinov, D.: Automatic Testing of Software with Structurally Complex Inputs. PhD thesis. MIT (2005)
26. Marinov, D., Andoni, A., Daniliuc, D., Khurshid, S., Rinard, M.: An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921 (2003)
27. Marriott, K., Stuckey, P.J.: Programming with constraints: an introduction. MIT Press, Cambridge (1998)
28. Meudec, C.: Atgen: automatic test data generation using constraint logic programming and symbolic execution. Software Testing, Verification and Reliability 11(2), 81–96 (2001)
29. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs. In: ICSE, pp. 771–774. IEEE Computer Society (2007)
30. Proietti, M., Pettorossi, A.: Unfolding - definition - folding, in this order, for avoiding unnecessary variables in logic programs. Theor. Comput. Sci. 142(1), 89–124 (1995)
31. Robinson, R.W.: Counting unlabeled acyclic digraphs. In: Little, C. (ed.) Combinatorial Mathematics V. Lecture Notes in Mathematics, vol. 622, pp. 28–43. Springer, Heidelberg (1977), 10.1007/BFb0069178
32. Senni, V., Fioravanti, F.: Generation of test data structures using constraint logic programming. Technical Report 12-04, IASI-CNR, Roma, Italy (2012)

33. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with java pathfinder. In: Avrunin, G.S., Rothermel, G. (eds.) ISSTA, pp. 97–107. ACM (2004)
34. Visvanathan, S., Gupta, N.: Generating test data for functions with pointer inputs. In: ASE, p. 149. IEEE Computer Society (2002)
35. Zhao, R., Li, Q.: Automatic test generation for dynamic data structures. In: SERA, pp. 545–549. IEEE Computer Society (2007)

Constructive Finite Trace Analysis with Linear Temporal Logic

Martin Sulzmann and Axel Zechner

Informatik Consulting Systems AG, Germany
{martin.sulzmann,axel.zechner}@ics-ag.de

Abstract. We consider linear temporal logic (LTL) for run-time testing over limited time periods. The technical challenge is to check if the finite trace produced by the system under test matches the LTL property. We present a constructive solution to this problem. Our finite trace LTL matching algorithm yields a proof explaining why a match exists. We apply our constructive LTL matching method to check if LTL properties are sufficiently covered by traces resulting from tests.

1 Introduction

Linear temporal logic (LTL) [4] is a powerful formalism for the concise specification of complex, temporal interaction patterns and has numerous applications to verify the static and dynamic behavior of software systems.

Our interest here is in the application of LTL for run-time testing. Specifically, our focus is on off-line testing where the system produces a *finite* trace log. The trace log is obtained by stimulation of the system by a test case. The resulting traces are then matched against some LTL formulas which express test properties.

There exists several prior works which study finite LTL trace matching, e.g. see [3,5]. The problem is that existing algorithms for finite trace matching only yield yes/no answers. That is, either the answer is yes and the trace could be matched, or the answer is no and there is no match. In our view this is often not sufficient. For example, we wish to have a more detailed explanation why a trace could be matched or why is there no match.

Our novel idea is to apply a constructive algorithm for finite trace matching where the algorithm yields a *proof* in case of a successful match. Proofs are representations of parse trees (a.k.a. derivation trees) and provide detailed explanations why there is a match. Thus, we can for example inspect some suspicious test cases which succeeded unexpectedly. There are several further advantages of representing the result of finite LTL trace matching in terms of proofs.

Proofs provide for independent verification of the test results. This is important in case we apply a finite trace LTL trace matching tool in the context of a formal software certification process such as DO-178B [6] where the tools output either must be formally certified or alternatively are manually verifiable. Formal tool certification is often too cost-intensive and requires a potential costly re-certification in case of software changes. Based on the proof representation it is straightforward to verify the test results manually.

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 132–148, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Via proofs it is also easy to accommodate for various matching semantics, e.g. weak or strong [2]. The advantage here is that we don't need to re-run the entire matching algorithm if for example we favor a weak semantics. We simply compute the proof and then afterwards we choose the appropriate proof interpretation, e.g. either weak or strong.

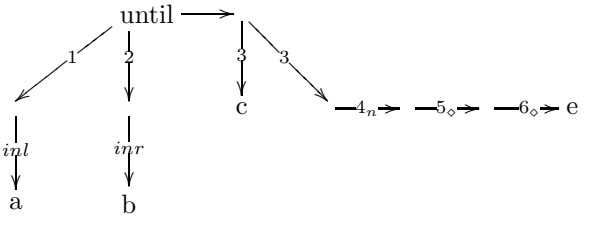
Proofs allow us to check to what extent the LTL properties are covered by tests (traces). For example, if some pre-condition is never satisfied the LTL property may be vacuously true but clearly the LTL property is then not fully covered. One of our new contributions is a method to *check* for a fixed set of LTL properties and traces if the LTL properties are sufficiently covered. This complements earlier works [7] which shows how to *generate* traces to sufficiently cover a given set of LTL properties. In practice, automatic generation of test cases is often not possible due to the lack of a formal test model on which we could apply a model checker. Therefore, tests are written by hand. The ability to check coverage of LTL properties is clearly a big win to evaluate the quality of a test suite.

The Key Ideas. We consider an example to highlight the key ideas of our work. We assume a finite trace of the form $[a, b, c, d, d, e, f]$ where letters $a - f$ stand for atomic propositions representing the inputs and outputs of the system under test recorded at some specific measuring points. The test property is specified via the LTL formula $(a \vee b)\ \mathit{until}\ (c \wedge \mathit{next}\ (\Diamond(e \vee f)))$.

Our constructive matching algorithm generates the following proof term

$$[\mathit{inl}\ a^{\checkmark}, \mathit{inr}\ b^{\checkmark}]\ \mathit{until}_{\mathit{prf}}\ (c^{\checkmark}, \mathit{fwd}_{\mathit{next}}\ \ \mathit{fwd}_{\Diamond}\ \mathit{fwd}_{\Diamond}\ \mathit{stop}_{\Diamond}\ e^{\checkmark}) \tag{1}$$

whose graphical representation is as follows



Edges are labeled with numbers where numbers refer to specific trace positions. Subscripts $_n$ and $_\Diamond$ tell us whether the next trace position is reached due to *next* or $\Diamond$. Label *inl* (*inr*) indicates matching against the left (right) component of choice ($\vee$). Leaf nodes represent matched atomic propositions. Based on this representation it is now quite clear that the trace matches the LTL formula.

There are further possible matches, i.e. proofs:

$$[\mathit{inl}\ a^{\checkmark}, \mathit{inr}\ b^{\checkmark}]\ \mathit{until}_{\mathit{prf}}\ (c^{\checkmark}, \mathit{fwd}_{\mathit{next}}\ \ \mathit{fwd}_{\Diamond}\ \mathit{fwd}_{\Diamond}\ \mathit{fwd}_{\Diamond}\ \mathit{stop}_{\Diamond}\ f^{\checkmark}) \tag{2}$$

$$[\mathit{inl}\ a^{\checkmark}, \mathit{inr}\ b^{\checkmark}]\ \mathit{until}_{\mathit{prf}}\ (c^{\checkmark}, \mathit{fwd}_{\mathit{next}}\ \ \mathit{fwd}_{\Diamond}\ \mathit{fwd}_{\Diamond}\ \mathit{fwd}_{\Diamond}\ \mathit{fwd}_{\Diamond}\ (\Diamond(e \vee f))^{?}) \tag{3}$$

In proof (2) we match f instead of e in $\Diamond(e \vee f)$. Proof (3) represents a match where we reach the end of trace without having matched the sub-formula $\Diamond(e \vee f)$.

This proof is of course quite silly because we know there is a match without $^?$. The point is that proofs can possibly represent 'partial' matches. That is, sub-formulas could not be matched due to a prematurely ending trace.

In particular, we are interested in 'shortest' proof. Informally, for a shortest proof the longest derivation path from the root to a leaf is minimal among all other proofs. Proof (1) is the shortest for our running example. Based on shortest proofs we can check if a test suite satisfies the *unique first cause* (UFC) coverage criteria [7]. Briefly, a test suite achieves UFC coverage of a set of requirements expressed as LTL formulas if each condition in an LTL formula has been shown to affect the formula's outcome as the unique first cause for some trace.

For our running example, we find that UFC coverage is *not* achieved. Condition f affects the formula's outcome (the formula is satisfied for this trace). But clearly f is not the unique first cause because in the trace $[a, b, c, d, d, e, f]$ there is the 'earlier' condition e due to which the formula is satisfied as well. This is easy to see by inspecting the shortest proof (1). $e^{\surd}$ appears in the proof (1) but $f^{\surd}$ is absent.

Via the additional trace $[a, b, c, d, d, f, e]$ we achieve UFC coverage. In the shortest proof

$$[inl\ a^{\surd}, inr\ b^{\surd}]\ until_{prf}\ (c^{\surd}, fwd_{next}\ \ fwd_{\diamond}\ fwd_{\diamond}\ stop_{\diamond}\ f^{\surd})$$

resulting from matching the above trace against the LTL formula we find $f^{\surd}$.

Based on the above observations, we can re-phrase the UFC coverage criteria as follows. To achieve UFC coverage, for each condition a in an LTL formula there must exist a shortest proof in which $a^{\surd}$ appears.

Summary of Contributions and Outline of Paper

- We give a constructive explanation of finite trace LTL matching where the particular finite trace semantics can be chosen a-posteriori (Section 2).
- We provide for an efficient algorithm to compute shortest proofs (Section 3).
- We re-phrase the UFC coverage criteria in terms of shortest proofs (Section 4).

Related work is discussed in Section 5.

2 Constructive Finite Trace LTL Matching

We formalize matching of finite trace T against an LTL formula L, see Figure 1. A trace T is a finite list of atomic propositions where we represent atomic propositions by letters a, b etc. In our actual implementation, propositions are compositions of more elementary basic conditions, e.g. $Key == On \wedge Speed > 100$. For brevity, we ignore this level of detail and only consider *atomic* propositions.

LTL in negation normal form:

$$\begin{array}{lll} B ::= a \mid b \mid ... & & \text{Atomic propositions} \\ L ::= B \mid \mathit{True} \mid \mathit{False} \mid L \wedge L \mid L \vee L & & \text{Boolean layer} \\ \quad \mid \mathit{next}\ L \mid \diamond L \mid L\ \mathit{until}\ L \mid \Box L & & \text{Temporal layer} \end{array}$$

Finite trace:

$$\begin{array}{lll} T ::= [] & & \text{Empty list/trace} \\ \quad \mid B : T & & \text{Trace with head } B \text{ and tail } T \end{array}$$

Proofs:

$$\begin{array}{l} P ::= B^{\checkmark} \mid \mathit{True}^{\checkmark} \mid L^{?} \mid \mathit{inl}\ P \mid \mathit{inr}\ P \mid (P, P) \\ \quad \mid \mathit{fwd}_{\mathit{next}}\ P \mid \Box\ Ps \mid \mathit{stop}_{\diamond}\ P \mid \mathit{fwd}_{\diamond}\ P \mid Ps\ \mathit{until}_{\mathit{prf}}\ P \\ Ps ::= [] \mid P : Ps \end{array}$$

Short-hand for list of proofs: $[P_1, ..., P_n] = P_1 : ... : P_n : []$

$$\boxed{T \vdash L \rightsquigarrow P}$$

$$\text{(True)}\quad T \vdash \mathit{True} \rightsquigarrow \mathit{True}^{\checkmark} \qquad \text{(Base)}\quad \frac{B' = B}{B' : T \vdash B \rightsquigarrow B^{\checkmark}}$$

$$\text{(EndOfTrace)}\quad \frac{L \neq \mathit{True}}{[] \vdash L \rightsquigarrow L^{?}} \qquad (\mathit{next})\quad \frac{T \vdash L \rightsquigarrow P}{B : T \vdash \mathit{next}\ L \rightsquigarrow \mathit{fwd}_{\mathit{next}}\ P}$$

$$(\vee\text{-Left})\quad \frac{B : T \vdash L_1 \rightsquigarrow P_1}{B : T \vdash L_1 \vee L_2 \rightsquigarrow \mathit{inl}\ P_1} \qquad (\vee\text{-Right})\quad \frac{B : T \vdash L_2 \rightsquigarrow P_2}{B : T \vdash L_1 \vee L_2 \rightsquigarrow \mathit{inr}\ P_2}$$

$$(\wedge)\quad \frac{B : T \vdash L_1 \rightsquigarrow P_1 \quad B : T \vdash L_2 \rightsquigarrow P_2}{B : T \vdash L_1 \wedge L_2 \rightsquigarrow (P_1, P_2)}$$

$$(\diamond\text{-Stop})\quad \frac{B : T \vdash L \rightsquigarrow P}{B : T \vdash \diamond L \rightsquigarrow \mathit{stop}_{\diamond}\ P} \qquad (\diamond\text{-Fwd})\quad \frac{T \vdash \diamond L \rightsquigarrow P}{B : T \vdash \diamond L \rightsquigarrow \mathit{fwd}_{\diamond}\ P}$$

$$(\Box\text{-1})\quad \frac{\begin{array}{c} B : T \vdash L \rightsquigarrow P_1 \\ T \vdash \Box L \rightsquigarrow \Box\ Ps \end{array}}{B : T \vdash \Box L \rightsquigarrow \Box\ (P_1 : Ps)} \qquad (\Box\text{-2})\quad \frac{[B] \vdash L \rightsquigarrow P_1}{[B] \vdash \Box L \rightsquigarrow \Box\ [P_1]}$$

$$(\mathit{until}\text{-1})\quad \frac{B : T \vdash L_2 \rightsquigarrow P_2}{B : T \vdash L_1\ \mathit{until}\ L_2 \rightsquigarrow []\ \mathit{until}_{\mathit{prf}}\ P_2}$$

$$(\mathit{until}\text{-2})\quad \frac{B : T \vdash L_1 \rightsquigarrow P_1 \quad T \vdash L_1\ \mathit{until}\ L_2 \rightsquigarrow Ps\ \mathit{until}_{\mathit{prf}}\ P_1}{B : T \vdash L_1\ \mathit{until}\ L_2 \rightsquigarrow P_1 : Ps\ \mathit{until}_{\mathit{prf}}\ P_2}$$

$$(\mathit{until}\text{-3})\quad \frac{[B] \vdash L_1 \rightsquigarrow P_1}{[B] \vdash L_1\ \mathit{until}\ L_2 \rightsquigarrow [P_1]\ \mathit{until}_{\mathit{prf}}\ L_2^{?}}$$

Fig. 1. Constructive Finite Trace LTL Matching

We use the standard LTL syntax. In our formulation, we assume that formulas are in negation normal form. For brevity, we omit the negation operator $\neg$ and assume that the negation of a proposition is simply represented as an atomic proposition, e.g. a.

The matching relation among finite traces and LTL formulas is described in terms of judgments of the form $T \vdash L \rightsquigarrow P$. The judgment $T \vdash L \rightsquigarrow P$ states that trace T matches formula L and the additional parameter P represents a proof for a match. We generally assume that for the matching relation $\cdot \vdash \cdot \rightsquigarrow \cdot$, T and L are input values and P is the output in case of a successful match.

Figure 1 contains the rules to derive judgments $T \vdash L \rightsquigarrow P$. A derivation with final judgment $T \vdash L \rightsquigarrow P$ is essentially a (up-side-down) tree where the leaves (judgments) are connected to base rules (EndOfTrace), (True) and (Base) and intermediate nodes are connected to the other rules. In essence, P represents a compact representation of the derivation tree.

The proof $B^{\checkmark}$ states that proposition B could be verified. Such proofs are derived via rule (Base) which states that B matches the head of the trace. $\mathit{True}^{\checkmark}$ is a proof for the always True formula which matches any trace. See rule (True).

Rule (EndOfTrace) and the corresponding proof $L^{?}$ indicate that L is unmatched because the trace ended prematurely. Depending on the interpretation of the match, i.e. proof, we can consider $L^{?}$ either as a valid match or not and can thus accommodate a weak or strong LTL matching semantics.

Rules ($\vee$-Left) and ($\vee$-Right) deal with matching a non-empty trace against a disjunction of formulas. Proofs $\mathit{inl}\ P$ and $\mathit{inr}\ P$ indicate which branch of a choice formula ($\vee$) could be matched. We use pairs (P, P) to represents proofs for matching a conjunction of formulas. See rule ($\wedge$).

Proof $\mathit{fwd}_{\mathit{next}}\ \ P$ represents a match for a $\mathit{next}\ L$ formula. See rule (next).

Proof $\mathit{stop}_{\diamond}\ P$ indicates that the eventually ($\diamond$) quantified formula could be matched at the present trace position whereas proof $\mathit{fwd}_{\diamond}\ P$ shows that we have to make a step forward in the future to find a match. See rules ($\diamond$-Stop) and ($\diamond$-Fwd).

For example, here's a derivation making use of rule ($\diamond$-Stop). For clarity, each derivation step is annotated with the respective rule applied (read upwards).

$$\dfrac{\dfrac{a = a}{[a, a] \vdash a \rightsquigarrow a^{\checkmark}}\,(\text{Base})}{[a, a] \vdash \diamond a \rightsquigarrow \mathit{stop}_{\diamond}\ (a^{\checkmark})}\,(\diamond\text{-Stop})$$

In case of the always (a.k.a. globally) operator $\Box$ the quantified LTL formula must hold at each position of the trace. The corresponding proof $\Box\ Ps$ contains therefore a list of proofs Ps where each individual proof represents a proof for a particular position. All of these proofs are collected in a list. See rule ($\Box$-1). In the last step, we reach the empty trace. The resulting proof $(\Box L)^{?}$ is ignored. See rule ($\Box$-2). For example, consider the following sample derivation.

$$\dfrac{\dfrac{a = a}{[a] \vdash a \rightsquigarrow a^{\surd}}\,(\text{Base}) \quad \dfrac{\Box a \neq \mathit{True}}{[] \vdash \Box a \rightsquigarrow (\Box a)^{?}}\,(\text{EndOfTrace})}{[a] \vdash \Box a \rightsquigarrow \Box\,[a^{\surd}]}\,(\Box\text{-}2)$$

Similarly to $\Box$, the proof for *until* uses a list to represent the sub-proof for the left operand. See rules (*until*-1) and (*until*-2). In our formulation, we also build a match/proof in case the right operand can never be matched but the left operand is matched at each position. See rule (*until*-3).

In summary, a proof P represents a compact representation of a derivation where a trace T matches an LTL formula L. That is, from the shape of P we can conclude which rules have been applied to build the derivation and we can reconstruct the entire derivation. In fact, each proof implies a trace and a formula such that the trace matches the formula. This is easy to see, by viewing P as the input and T and L as outputs of the matching relation $T \vdash L \rightsquigarrow P$. It follows:

Lemma 1 (Proofs Represent Derivations). *(1) Let $T \vdash L \rightsquigarrow P$ be the final judgment of a derivation. Then, proof P exactly tells us which rules have been applied and in which order.*

(2) Let P be a proof. Then, $T \vdash L \rightsquigarrow P$ is derivable for some trace T and formula L.

A-Posteriori Weak Interpretation of Proofs. The general problem with LTL and finite traces is how to deal with cases where the trace ends prematurely. For example, consider the proof resulting from the derivation

$$\dfrac{\dfrac{a \neq \mathit{True}}{[] \vdash a \rightsquigarrow a^{?}}\,(\text{EndOfTrace})}{[a] \vdash \mathit{next}\ a \rightsquigarrow \mathit{fwd}_{\mathit{next}}\ \ (a^{?})}\,(\mathit{next})$$

In the context of testing with LTL, it is likely that some test cases (traces) are too short for some test properties (LTL formulas). To avoid false negatives, we favor a weak interpretation of proofs.

Definition 1 (Weak Proof Interpretation). *We say that formula L is* weakly matched *by trace T, written $T \vdash_{weak} L$, iff there exists a proof P such that $T \vdash L \rightsquigarrow P$.*

We find that the formula in the above example is weakly matched.

Similarly, we can give a strong proof interpretation following the strong finite trace semantics introduced in [2].

Definition 2 (Strong Proof Interpretation). *We say that formula L is* strongly matched *by trace T, written $T \vdash_{strong} L$, iff there exists a proof P such that $T \vdash L \rightsquigarrow P$ and P does not contain any term of the form $\cdot^{?}$.*

$$\boxed{size : P \mapsto \mathbb{N}}$$

$$\begin{array}{ll}
size(B^{\surd}) = 1 & size(inl\ P) = size(P) \\
size(True^{\surd}) = 0 & size(inr\ P) = size(P) \\
size(L^{?}) = size(L) & size(P_1, P_2) = max(size(P_1), size(P_2)) \\
size(stop_{\diamond}\ P) = size(P) & size(fwd_{\diamond}\ P) = 1 + size(P) \\
size(fwd_{next}\ \ P) = 1 + size(P) & \\
size(\Box[P_1, ..., P_n]) = 1 + max_{i=1}^{i \leq n}((i-1) + size(P_i)) & \\
size([P_1, ..., P_n]\ until_{prf}\ P) = 1 + max(n + size(P), max_{i=1}^{i \leq n}((i-1) + size(P_i))) &
\end{array}$$

$$\boxed{size : L \mapsto \mathbb{N}}$$

$$\begin{array}{ll}
size(B) = 1 & size(True) = 1 \\
size(False) = 1 & size(L_1 \wedge L_2) = size(L_1) + size(L_2) \\
size(L_1 \vee L_2) = size(L_1) + size(L_2) & size(next\ L) = 1 + size(L) \\
size(\diamond L) = 1 + size(L) & size(L_1\ until\ L_2) = 1 + size(L_1) + size(L_2) \\
size(\Box L) = 1 + size(L) &
\end{array}$$

Fig. 2. Size of Proofs and Formulas

In case of a weak proof interpretation, it is important to check that the LTL properties are sufficiently covered by test cases. As motivated in the introduction, we use shortest proofs for coverage checking. Next, we formalize shortest proofs.

Shortest Proofs. Figure 2 defines the size of a proof and a formula. The size of the proof is the longest possible path from the root to a leaf. Leafs $B^{\surd}$ have size 1 whereas leafs which contain $True^{\surd}$ represent trivial matches and therefore we set their size to 0.

For example, consider the proofs obtained by matching $[a, a]$ against $next\ a \vee \diamond a$:

$$[a, a] \vdash next\ a \vee \diamond a \rightsquigarrow inr\ (stop_{\diamond}\ a^{\surd}) \quad [a, a] \vdash next\ a \vee \diamond a \rightsquigarrow inl\ (fwd_{next}\ \ a^{\surd})$$

where $size(inr\ (stop_{\diamond}\ a^{\surd})) = 1 < 2 = size(inl\ (fwd_{next}\ \ a^{\surd}))$.

In case of a proof for $\wedge$ the size of the overall proof is determined by the maximum of the size of the sub-proofs. Similarly, we compute the maximum of the sub-proofs for $\Box$ and $until$. The difference compared to $\wedge$ is that for $\Box$ and $until$ we add $i - 1$ to take into account the iterations through the trace. The additional 1+ in e.g. $size(\Box L) = 1 + size(L)$ ensures to unambiguously select among proofs for $\Box$ and $until$ and proofs for some unfolding of $\Box$ and $until$ for a specific trace. For example, consider

$$[a] \vdash \Box a \vee a \rightsquigarrow inl\ \Box\ [a^{\surd}] \qquad [a] \vdash \Box a \vee a \rightsquigarrow inr\ a^{\surd}$$

For trace $[a]$, proposition a is essentially the unfolded version of $\Box a$. We strictly favor the unfolded version by adding 1 in case of $\Box$. For the above, we find that

$$size(inr\ a^{\surd}) = 1 < 2 = size(inl\ \Box\ [a^{\surd}])$$

The only remaining ambiguity arises in pathological cases such as matching $[a]$ against $a \vee a$ and matching $[a]$ against $\mathit{next}\ b \vee \mathit{next}\ c$. In the first case, we find two identical matches by either choosing the left or right branch. In the second case, the trace ended prematurely and we end up with unresolved formulas of equal size in the left and right branch.

To resolve such un-ambiguities, we favor the "left-most" proof in case of several shortest proofs. For brevity, we omit a formal definition and only provide the intuition. We say a proof P_1 is *left-most* w.r.t. some other proof P_2 iff along the longest paths from the root of P_1 and P_2 we find that P_1's path takes earlier a left turn than P_2's path.

Definition 3 (Shortest Left-Most Proof). *Let $T \vdash L \rightsquigarrow P$. We say that P is the* shortest left-most proof *w.r.t. trace T and formula L iff for any other proof P' such that $T \vdash L \rightsquigarrow P'$ we have that either*

- $\mathit{size}(P) < \mathit{size}(P')$, *or*
- $\mathit{size}(P) = \mathit{size}(P')$ *and P is left-most w.r.t. P'.*

Obviously, there exists other strategies to make the matching relation deterministic. For example, instead of choosing the left-most proof among the shortest proofs, we could choose the shortest proof among the left-most proofs.

Definition 4 (Left-Most Shortest Proof). *Let $T \vdash L \rightsquigarrow P$. We say that P is the* left-most shortest proof *w.r.t. trace T and formula L iff P is a left-most proof and for any other proof left-most proof P' such that $T \vdash L \rightsquigarrow P'$ we have that $\mathit{size}(P) < \mathit{size}(P')$.*

For example, proof $\mathit{inl}\ \Box\ [a^{\checkmark}]$ is the left-most shortest proof for trace $[a]$ and $\Box a \vee a$. But as shown above, this proof is not the shortest left-most.

3 Deterministic Matching with Derivatives

We first develop an algorithm to compute the left-most shortest match. Based on that we then derive an algorithm for computing the shortest left-most match.

The straightforward approach to obtain the left-most shortest match would be to employ a back-tracking algorithm where we interpret the judgments in Figure 1 as Prolog clauses. However, such an approach easily leads to undesirable high run-time behavior.

For example, consider the trace $[a, ..., a, c]$ and the formula $\Diamond(a \wedge \Diamond b)$. In each step, besides the last step, we can match a and then seek for b which cannot be matched. Thus, we end up with a quadratic run-time behavior where we would expect that a linear scan of the trace ought to be sufficient. This situation is similar to the regular expression for which it is well-known that a back-tracking matching algorithm easily leads to exponential run-time behavior.

To avoid unnecessary back-tracking, we seek for a matching algorithm which strictly guarantees to make progress towards computing a proof. The basic idea is to reduce the matching problem $B : T \vdash L$ to the simpler problem $T \vdash L \backslash B$

Expressions and functions over proofs:

$$
\begin{array}{lll}
e ::= P & & \text{Proofs} \\
\quad \mid \ \mathsf{case}\ P\ \mathsf{of}\ \overline{P \to P} & & \text{Case expression} \\
f ::= \lambda P.e & & \text{Functions with input pattern } P \\
\quad \mid \ \bot & & \text{Undefined}
\end{array}
$$

$$\boxed{L \backslash B \vdash_d \ (L \mid P \to P)}$$

$$(\text{True}_d) \quad True \backslash B \vdash_d \ (True \mid \lambda True^{\surd}.True^{\surd}) \qquad (\text{False}_d) \quad False \backslash B \vdash_d \ (False \mid \bot)$$

$$(\text{Succ-B}_d) \ \frac{B' = B}{B' \backslash B \vdash_d \ (True \mid \lambda True^{\surd}.B^{\surd})} \qquad (\text{Fail-B}_d) \ \frac{B' \neq B}{B' \backslash B \vdash_d \ (False \mid \bot)}$$

$$(\vee_d) \ \frac{\begin{array}{l} L_1 \backslash B \vdash_d \ (L_1' \mid f_1) \\ L_2 \backslash B \vdash_d \ (L_2' \mid f_2) \\ f = \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \qquad inl\ P' \to inl\ (f_1\ P') \\ \qquad inr\ P' \to inr\ (f_2\ P') \end{array}}{(L_1 \vee L_2) \backslash B \vdash_d \ (L_1' \vee L_2' \mid f)} \qquad (\wedge_d) \ \frac{\begin{array}{c} L_1 \backslash B \vdash_d \ (L_1' \mid f_1) \\ L_2 \backslash B \vdash_d \ (L_2' \mid f_2) \\ f = \lambda(P_1, P_2).(f_1\ P_1, f_2\ P_2) \end{array}}{(L_1 \wedge L_2) \backslash B \vdash_d \ (L_1' \wedge L_2' \mid f)}$$

$$(next_d) \quad (next\ L) \backslash B \vdash_d \ (L \mid \lambda P.fwd_{next}\ \ P)$$

$$(\diamond_d) \ \frac{\begin{array}{l} L \backslash B \vdash_d \ (L' \mid f') \\ f = \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \qquad inl\ P' \to stop_{\diamond}\ (f`\ P') \\ \qquad inr\ P' \to fwd_{\diamond}\ P' \end{array}}{(\diamond L) \backslash B \vdash_d \ (L' \vee \diamond L \mid f)} \qquad (\Box_d) \ \frac{\begin{array}{c} L \backslash B \vdash_d \ (L', f') \\ f = \lambda(P, P').\ \mathsf{case}\ P'\ \mathsf{of} \\ \qquad \Box\ Ps \to \Box\ (f'\ P : Ps) \\ \qquad (\Box L)^? \to \Box\ [f'\ P] \end{array}}{(\Box L) \backslash B \vdash_d \ (L' \wedge \Box L \mid f)}$$

$$(until_d) \ \frac{\begin{array}{l} L_1 \backslash B \vdash_d \ (L_1', f_1) \quad L_2 \backslash B \vdash_d \ (L_2', f_2) \\ f = \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \qquad inl\ P' \qquad\qquad\qquad\quad \to []\ until_{prf}\ f_2\ P' \\ \qquad inr\ (P_1, P_2\ until_{prf}\ P_3) \to ((f_1\ P_1) : P_2)\ until_{prf}\ P_3 \\ \qquad inr\ (P_1, P_2^?) \qquad\qquad \to [f_1\ P_1]\ until_{prf}\ (P_2^?) \end{array}}{(L_1\ until\ L_2) \backslash B \vdash_d \ (L_2' \vee (L_1' \wedge (L_1\ until\ L_2)) \mid f)}$$

Fig. 3. Building Derivatives and Proof Transformers

where formula $L \backslash B$ is obtained from L by consuming the current head B of the trace. The formula $L \backslash B$ is referred to as the *derivative* of L with respect to B and can be obtained by structural induction over the shape of L. The concept of derivatives, originally developed for regular expressions [1], also applies to linear temporal logic as first shown in [3] We extend this idea to compute the left-most shortest and shortest left-most match.

One of the challenges we face is to build the proof of the original formula out of the proof of the derivative. Roughly, we attack this challenge as follows. For a trace $[B_1, ..., B_n]$, we build the sequence of derivatives $L \to_{f_1} L \backslash B_1 \to_{f_2} ... \to_{f_n} L \backslash B_1 ... \backslash B_n$. The purpose of the f_i's will be explained shortly. By using

$$\boxed{L \vdash_p P}$$

$$(\text{True}_p)\quad \mathit{True} \vdash_p \mathit{True}^{\checkmark} \qquad (\wedge)\quad \frac{L_1 \vdash_p P_1 \quad L_2 \vdash_p P_2}{(L_1 \wedge L_2) \vdash_p (P_1, P_2)}$$

$$(\vee\text{-Left}_p)\quad \frac{L_1 \vdash_p P_1}{(L_1 \vee L_2) \vdash_p \mathit{inl}\ P_1} \qquad (\vee\text{-Right}_p)\quad \frac{\text{there is no } P_1 \text{ such that } L_1 \vdash_p P_1 \quad L_2 \vdash_p P_2}{(L_1 \vee L_2) \vdash_p \mathit{inl}\ P_2}$$

$$(\text{Base}_p)\quad B \vdash_p B^{?} \qquad (\diamond_p)\quad \diamond L \vdash_p (\diamond L)^{?} \qquad (\mathit{next}_p)\quad \mathit{next}\ L \vdash_p (\mathit{next}\ L)^{?}$$

$$(\Box_p)\quad \Box L \vdash_p (\Box L)^{?} \qquad (\mathit{until}_p)\quad L_1\ \mathit{until}\ L_2 \vdash_p (L_1\ \mathit{until}\ L_2)^{?}$$

Fig. 4. Building the Final Left-Most Proof (weak version)

Boolean laws we check if the final formula $L\backslash B_1 ... \backslash B_n$ yields true. If yes, we can build a proof P. The proof for the original formula L is obtained by applying the proof transformers f_i. In each derivative step, we compute a proof transformer function f_i which tells us how to build the proof of the original formula given the proof of the derivative. Thus, we obtain the proof of the initial formula L by application of $(f_1 \circ \ ... \circ \ f_n)\ P$. Next, we formalize this idea.

Computing the Left-Most Shortest Match. Figure 3 defines judgments $L\backslash B \vdash_d \ (L' \mid f)$ which build the derivative $L' = L\backslash B$ and also a proof transformation function which transforms a proof for L' into a proof for L.

The base cases (True_d), (False_d), (Succ-B_d) and (Fail-B_d) are straightforward. False formulas are represented by $\perp$, the undefined proof transformer. As we will see, false formulas and their $\perp$ proofs only appear in intermediate steps. They will be eventually discarded because they are not derivable in our matching rule system.

Rules ($\vee_d$) and ($\wedge_d$) are defined by structural induction and contain no surprises. In rule (next_d), we simply drop the $\mathit{next}\ \cdot$ operator.

More interesting is rule ($\diamond_d$). The derivative of $\diamond L$ w.r.t. B is $(L\backslash B) \vee \diamond L$ where $L\backslash B$ is the derivative of L w.r.t. B. As we will see, we favor the 'left-most' match and therefore we first try to find a match for L at the current position B and only in case of failure we will continue with the next position by trying again $\diamond L$. The proof transformation function f checks if a proof is found in either the left or right component of the resulting derivative and then applies the proof transformer resulting from $L\backslash B$ to construct a proof for $\diamond L$.

In rule ($\Box_d$), we assume that eventually $\Box L$ is matched against the empty trace which then results in the proof $(\Box L)^{?}$. Therefore, the second case when building the proof for $\Box L$ given the proof for $L\backslash B \wedge \Box L$.

In rule ($until_d$), the derivative for L_1 *until* L_2 is $L_2\backslash B \vee (L_1\backslash B \wedge (L_1 \textit{ until } L_2))$ and expresses that we either immediately satisfy L_2, or we must further unroll the until formula. The resulting proof transformer f covers all the until cases (1-3) we have seen in Figure 1.

Figure 4 builds a proof for the final LTL formula. Any unmatched LTL formula is considered as possibly true. Recall that we postpone the decision of how to interpret proofs. The rules strictly favor the left-most match. For example, see rules ($\vee$-Left$_p$) and ($\vee$-Right$_p$).

We have now everything at hand to formalize the derivative-based algorithm for matching a trace against a formula.

Definition 5 (Derivatives Matching Algorithm). *Let L be an LTL formula, T be a finite trace of the form $[B_1, ..., B_n]$ and P be a proof. We say that P is the derivative matching result of matching T against L, written $T \vdash_d L \rightsquigarrow P$, iff*

- $L\backslash B_1 \vdash_d (L_1 \blacktriangleright f_1)$,...,$L_{n-1}\backslash B_n \vdash_d (L_n \blacktriangleright f_n)$ *for some* L_1,...,L_n *and* f_1, ...,f_n, *and*
- $L_n \vdash_p P'$ *for some* P', *and*
- $P = (f_1 \circ \ ... \circ \ f_n)\ P'$.

For example, consider trace $[a, a]$ and formula *next* $a \vee \diamond a$. We first build the derivatives of $L = \textit{next } a \vee \diamond a$:

$$L\backslash a = \underbrace{a \vee (\mathit{True} \vee \diamond a)}_{L_1} \quad L_1\backslash a = \underbrace{\mathit{True} \vee (\mathit{True} \vee (\mathit{True} \vee \diamond a))}_{L_2}$$

The proof transformers connected to the derivative steps are as follows:

$$L\backslash a \vdash_d L_1 \blacktriangleright \underbrace{\begin{array}{ll} \lambda P.\ \mathsf{case}\ P\ \mathsf{of} & \\ \quad \mathit{inl}\ P' \rightarrow & \mathit{inl}\ (\mathit{fwd}_{next}\ \ P') \\ \quad \mathit{inr}\ P' \rightarrow & \mathsf{case}\ P'\ \mathsf{of} \\ & \quad \mathit{inl}\ P'' \rightarrow \mathit{stop}_\diamond\ a^\surd \\ & \quad \mathit{inr}\ P'' \rightarrow \mathit{fwd}_\diamond\ P'' \end{array}}_{f_1}$$

$$L_1\backslash a \vdash_d L_2 \blacktriangleright \underbrace{\begin{array}{lll} \lambda P.\mathsf{case}\ P\ \mathsf{of} & \mathit{inl}\ P' \rightarrow & \mathit{inl}\ a^\surd \\ & \mathit{inr}\ P' \rightarrow & \mathsf{case}\ P'\ \mathsf{of} \\ & & \quad \mathit{inl}\ P'' \rightarrow \mathit{inl}\ \mathit{True}^\surd \\ & & \quad \mathit{inr}\ P'' \rightarrow \qquad \mathsf{case}'\,P''\ \mathsf{of} \\ & & \qquad\qquad \mathit{inl}\ P''' \rightarrow \mathit{stop}_\diamond\ \mathit{True}^\surd \\ & & \qquad\qquad \mathit{inr}\ P''' \rightarrow \mathit{fwd}_\diamond\ P''' \end{array}}_{f_2}$$

For the final formula, we find

$$\mathit{True} \vee (\mathit{True} \vee (\mathit{True} \vee \diamond a)) \vdash_p \mathit{inl}\ \mathit{True}^\surd$$

We now transform the final proof into a proof of the initial formula by applying the proof transformers connected to the derivative steps:

$$(f_1 \circ f_2)(\mathit{inl}\ \mathit{True}^{\surd}) = \mathit{inl}\ (\mathit{fwd}_{\mathit{next}}\ \ a^{\surd})$$

The proof on the right is the proof for the original formula $\mathit{next}\ a \vee \Diamond a$. This proof is also the left-most shortest proof. This result holds in general.

In a first step, we verify that the proof transformer connected to the derivative computes the proof of the original formula given the proof of the derivative.

Lemma 2 (Derivatives Matching Correctness). *Let* $L \backslash B \vdash_d (L' \mid f)$ *and* $T \vdash L' \rightsquigarrow P'$. *Then,* $B : T \vdash L \rightsquigarrow P$ *for some* P *such that* $f\ P' = P$.

Proof. (Sketch) By induction over the structure of L and the derivation $T \vdash L' \rightsquigarrow P'$. For example, consider $L_1\ \mathit{until}\ L_2$. Case (until_d) applies. By assumption we have that $T \vdash L_2' \vee (L_1' \wedge (L_1\ \mathit{until}\ L_2)) \rightsquigarrow P''$. For brevity, we only consider the case $P'' = \mathit{inl}\ P'$. Thus, we conclude that $T \vdash L_2' \rightsquigarrow P'$ (1). From the premise of case (until_d), we conclude $L_2 \backslash B \vdash_d (L_2', f_2)$ (2). By induction hypothesis applied to (1) and (2) we conclude that $B : T \vdash L_2 \rightsquigarrow f_2\ P'$. Via rule rule ($\mathit{until}$-1) we conclude that $B : T \vdash L_1\ \mathit{until}\ L_2 \rightsquigarrow []\ \mathit{until}_{prf}\ f_2\ P'$. By construction we find that $f\ \mathit{inl}\ P' = []\ \mathit{until}_{prf}\ f_2\ P'$ and thus we are done.

The other cases can be proven similarly.

The following result follows immediately by construction.

Lemma 3 (Correctness of Final Left-Most Proof). *Let* $L \vdash_p P$. *Then* $[] \vdash L \rightsquigarrow P$ *and* P *is the left-most shortest proof w.r.t.* $[]$ *and* L.

The composition of the individual proof transformers clearly yields a valid proof of the original formula. We further know that the final proof is the left-most shortest proof. The important observation is that the derivatives matching step $L \backslash B \vdash_d (L' \mid f)$ preserves left-most shortest proofs. That is, if the proof P' of L' is left-most shortest, then it follows that proof $f\ P'$ of L is also left-most shortest. Thus, we obtain the following result.

Theorem 1 (Left-Most Shortest Derivatives Matching Correctness). *Let* $T \vdash_d L \rightsquigarrow P$ *for some trace* T, *LTL formula* L *and proof* P. *Then,* $T \vdash L \rightsquigarrow P$ *and* P *is the left-most shortest proof w.r.t.* T *and* L.

Computing the Shortest Left-Most Match. We are now interested in the *shortest* match. There are several adjustments we need to make to the derivative-based matching algorithm:

- (1) We must aggressively simplify formulas by using Boolean laws such as $L \vee \mathit{True} = L$. Thus, we favor formulas which evaluate as early as possible to *True* and the resulting proofs are shorter.

$$\boxed{L \vdash_s (L \mid P \to P)}$$

$$(\mathrm{T}_s)\ \ \mathit{True} \vdash_s (\mathit{True} \mid \lambda P.P) \qquad (\mathrm{F}_s)\ \ \mathit{False} \vdash_s (\mathit{False} \mid \bot) \qquad (\mathrm{B}_s)\ \ B \vdash_s (B \mid \lambda P.P)$$

$$(\vee\text{-}1_s)\ \ (\mathit{True} \vee L) \vdash_s (\mathit{True} \mid \lambda P.\mathit{inl}\ P) \qquad (\vee\text{-}2_s)\ \ (L \vee \mathit{True}) \vdash_s (\mathit{True} \mid \lambda P.\mathit{inr}\ P)$$

$$(\vee\text{-}3_s)\ \ (\mathit{False} \vee L) \vdash_s (L \mid \lambda P.\mathit{inr}\ P) \qquad (\vee\text{-}4_s)\ \ (L \vee \mathit{False}) \vdash_s (L \mid \lambda P.\mathit{inl}\ P)$$

$$(\vee\text{-}5_s)\ \ \frac{\begin{array}{c} L_1 \neq \mathit{False} \text{ and } L_1 \neq \mathit{True} \text{ and } L_2 \neq \mathit{False} \text{ and } L_1 \neq L_2 \\ L_1 \vdash_s (L_1' \mid f_1) \quad L_2 \vdash_s (L_2' \mid f_2) \\ f = \lambda P.\ \mathsf{case}\ P\ \mathsf{of} \\ \qquad \mathit{inl}\ P' \to \mathit{inl}\ (f_1\ P') \\ \qquad \mathit{inr}\ P' \to \mathit{inr}\ (f_2\ P') \end{array}}{(L_1 \vee L_2) \vdash_s ((L_1' \vee L_2') \mid f)}$$

$$(\vee\text{-}6_s)\ \ \frac{L \vdash_s (L' \mid f') \quad f = \lambda P.\mathit{inl}\ (f'\ P)}{(L \vee L) \vdash_s (L' \mid f)}$$

$$(\wedge\text{-}1_s)\ (\mathit{True} \wedge L) \vdash_s (L \mid \lambda P.(\mathit{True}^{\checkmark}, P)) \qquad (\wedge\text{-}2_s)\ (L \wedge \mathit{True}) \vdash_s (L \mid \lambda P.(P, \mathit{True}^{\checkmark}))$$

$$(\wedge\text{-}3_s)\ \ (\mathit{False} \wedge L) \vdash_s (\mathit{False} \mid \bot) \qquad (\wedge\text{-}4_s)\ \ (L \wedge \mathit{False}) \vdash_s (\mathit{False} \mid \bot)$$

$$(\wedge\text{-}5_s)\ \ \frac{\begin{array}{c} L_1 \neq \mathit{True} \text{ and } L_1 \neq \mathit{False} \text{ and } L_2 \neq \mathit{True} \text{ and } L_2 \neq \mathit{False} \text{ and } L_1 \neq L_2 \\ L_1 \vdash_s (L_1' \mid f_1) \quad L_2 \vdash_s (L_2' \mid f_2) \end{array}}{(L_1 \wedge L_2) \vdash_s ((L_1' \wedge L_2') \mid \lambda(P_1, P_2).(f_1\ P_1, f_2\ P_2))}$$

$$(\wedge\text{-}6_s)\ \ \frac{L \vdash_s (L' \mid f') \quad f = \lambda P.(f'\ P, f'\ P)}{(L \wedge L) \vdash_s (L' \mid f)}$$

Fig. 5. Simplifications and Proof Transformers I

- (2) The simplifications must be applied in intermediate derivative matching steps.
- (3) We currently built the left-most shortest final proof. Here, we need some additional rules to guarantee that we built the shortest left-most final proof.

To motivate (1) and (2) we consider formula $\mathit{next}\ a \vee a$ and trace $[a, a]$. For brevity, we only consider the resulting derivatives which are:

$$(\mathit{next}\ a \vee a)\backslash a = a \vee \mathit{True} \qquad (a \vee \mathit{True})\backslash a = \mathit{True} \vee \mathit{True}$$

From $\mathit{True} \vee \mathit{True}$ we obtain the final proof $\mathit{inl}\ \mathit{True}^{\checkmark}$. Application of the proof transformers connected to derivatives then leads to $\mathit{inl}\ \mathit{fwd}_{\mathit{next}}\ \ a^{\checkmark}$. This is the left-most shortest proof but clearly not the shortest left-most proof which is $\mathit{inr}\ a^{\checkmark}$.

To obtain the shortest proof we must apply simplifications also in intermediate steps. For our example, in the first derivative matching step we simplify $a \vee \mathit{True}$ to True. The subsequent derivative step $\mathit{True}\backslash a = \mathit{True}$ then yields the

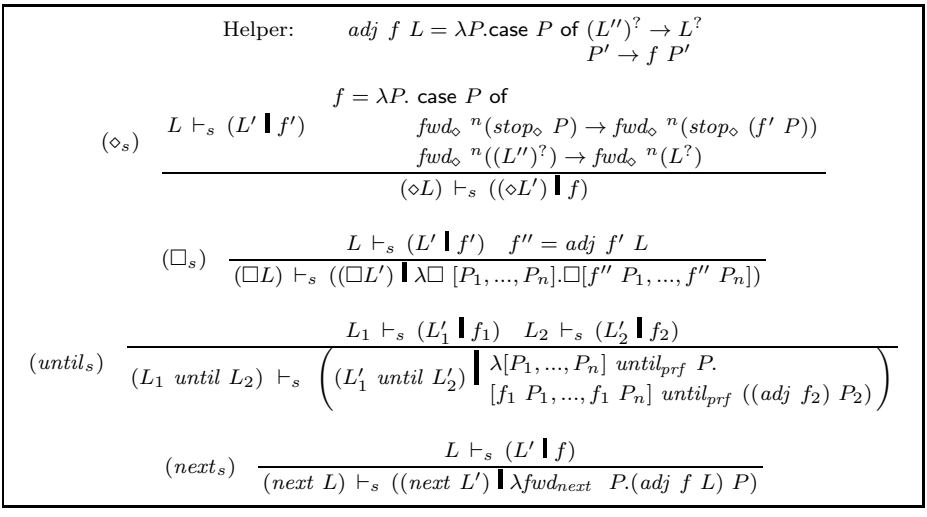


Fig. 6. Simplifications and Proof Transformers II

final formula *True* which has the final proof $True^\surd$. Application of the proof transformers connected to the derivative matching and simplification step then leads to $inr\ a^\surd$. This is the shortest left-most proof we were looking for.

Next, we provide the details of the simplification step in terms of judgments $L \vdash_s (L \mid P \to P)$. Similar to the derivative matching step, each simplification step yields a proof transformer which builds a proof of the original formula given a proof of the simplified formula. The simplification rules are specified in Figures 5 and 6.

Figure 5 contains the standard Boolean simplifications concerning $\vee$ etc. In the LTL context, (Boolean) simplification also need to be applied 'below' LTL operators. For example, consider $next\ (a \vee True)$ which shall be simplified to $next\ True$. For such simplifications, we apply the rules in Figure 6.

In rule $(\diamond_s)$, we make use of the short-hand notation $fwd_\diamond{}^n(P)$:

$$fwd_\diamond{}^0(P) = P \qquad fwd_\diamond{}^{n+1}(P) = fwd_\diamond\ (fwd_\diamond{}^n(P))$$

The proof transformation function f in this rule distinguishes between the case that a proof for L could be found, resp. the trace ended prematurely. In the first case, we follow the chain of $fwd_\diamond$ steps until we reach $stop_\diamond\ P$ which is then replaced by $stop_\diamond\ (f'\ P)$. In case the trace ended, represented by some proof $L''^?$, we use the original (non-simplified) formula L to represent the proof $fwd_\diamond{}^n(L^?)$ for $\diamond L$.

In rule $(\Box_s)$, we apply the proof transformer f' to each of the sub-proofs. The exception is in case of a sub-proof of the form $^?$. This must be the last sub-proof. Like in case of rule $(\diamond_s)$, we use the original (non-simplified) formula L to represent the last sub-proof of $\Box L$. For brevity, we make use of the helper function $adj\ f\ L$ to either apply f or simply return $L^?$. This helper function is also used in rules $(until_s)$ and $(next_s)$.

We always assume that simplification rules are applied aggressively by traversing an LTL formula from top to bottom and from left to right. Then, the following result follows.

Lemma 4 (Simplification Correctness and Preservation of Shortest Left-Most). *Let $L \vdash_s (L' \blacktriangleright f)$ and $T \vdash L' \rightsquigarrow P'$ such that P' is the shortest left-most proof w.r.t. T and L'. Then, $T \vdash L \rightsquigarrow P$ for some P such that $f\ P' = P$ and P is the shortest left-most proof w.r.t. T and L.*

We yet need to address (3) from above. For example, consider formula *next* $(next\ a) \vee next\ b$ and trace $[c]$. The final formula is *next* $a \vee b$. The current final proof construction algorithm in Figure 4 yields $inl\ (next\ a)^?$ but the shortest final proof is $inr\ a^?$.

Hence, we extend Figure 4 with two additional rules. We write $L \vdash_{p_s} P$ to denote proof construction form formulas using the extended set of rules.

$$(\vee\text{-L}^?{}_p)\ \frac{L_1 \vdash_{p_s} L_1'^?\quad L_2 \vdash_{p_s} L_2'^?\quad size(L_1'^?) \leq size(L_2'^?)}{(L_1 \vee L_2) \vdash_{p_s} inl\ L_1'^?} \qquad (\vee\text{-R}^?{}_p)\ \frac{L_1 \vdash_{p_s} L_1'^?\quad L_2 \vdash_{p_s} L_2'^?\quad size(L_2'^?) < size(L_1'^?)}{(L_1 \vee L_2) \vdash_{p_s} inr\ L_2'^?}$$

The above rules apply if both branches of a 'choice' formula are unmatched. Otherwise, we will apply the existing rules ($\vee$-Left$_p$) and ($\vee$-Right$_p$). Thus, $(next\ a \vee b) \vdash_{p_s} inr\ a^?$.

Lemma 5 (Correctness of Final Shortest Proof). *Let $L' \vdash_s (L \blacktriangleright f)$ and $L \vdash_{p_s} P$. Then $[] \vdash L \rightsquigarrow P$ and P is the shortest left-most proof w.r.t. $[]$ and L.*

The definition of the shortest-left most match algorithm follows addressing the above points (1-3).

Definition 6 (Shortest Left-Most Match Algorithm). *Let L be an LTL formula, T be a finite trace of the form $[B_1, ..., B_n]$ and P be a proof. We define $T \vdash_{d_{slm}} L \rightsquigarrow P$ iff*

- $L \vdash_s (L' \blacktriangleright f')$, $L \backslash B_1 \vdash_d (L_1 \blacktriangleright f_1)$,
  $L_1 \vdash_s (L_1' \blacktriangleright f_1')$, $L_1' \backslash B_2 \vdash_d (L_2 \blacktriangleright f_2)$,
  ...,
  $L_{n-1} \vdash_s (L_{n-1}' \blacktriangleright f_{n-1}')$, $L_{n-1}' \backslash B_n \vdash_d (L_n \blacktriangleright f_n)$,
  for some $L', L_1, L_1' ..., L_n$ *and* $f', f_1, f_1' ..., f_n$, *and*
- $L_n \vdash_s (L_n' \blacktriangleright f_n')$, $L_n' \vdash_{p_s} P'$ *for some* P', L_n', f_n', *and*
- $P = (f_1 \circ f_1' \circ \ ... \circ \ f_n \circ f_n')\ P'$.

Theorem 2 (Computing the Shortest Left-Most Proof). *Let $T \vdash_{d_{slm}} L \rightsquigarrow P$ for some trace T, LTL formula L and proof P. Then, we have that $T \vdash L \rightsquigarrow P$ and P is the shortest left-most proof w.r.t. T and L.*

The above result provides the basis for checking coverage of a set of requirements expressed as LTL formulas.

4 Checking LTL Coverage by Inspecting Proofs

We repeat the *unique first cause* (UFC) coverage condition proposed in [7]: A test suite achieves UFC coverage of a set of requirements expressed as temporal formulas, if: (1) every basic condition in any formula has taken on all possible outcomes at least once and (2) each basic condition has been shown to affect the formula's outcome as the unique first cause. A condition a is the unique first cause (UFC) for ϕ along a path π if, in the first state along π in which ϕ is satisfied, it is satisfied because of a.

Condition (1) essentially corresponds to the MC/DC coverage criteria. In our formulation, we ignore this level of detail here because we only consider atomic propositions at the Boolean propositional level.

The important point is that condition (2) can be characterized precisely in terms of shortest left-most proofs. Roughly, conditions a in some test property L must be covered by some shortest left-most proof P. That is, $a^{\surd}$ in P. To unambiguously distinguish among several occurrences of a, e.g. as in $a \vee a$, we attach distinct labels k to conditions a, written a_k. For example, $a_1 \vee a_2$. Thus, we can re-phrase the unique first cause coverage condition as follows.

Definition 7 (Unique First Cause Coverage Revisited Condition). *A test suite is a set $\{T_1, ..., T_n\}$ of traces and a set $\{L_1, ..., L_m\}$ of LTL test properties.*

We say that a test suite satisfies the unique first cause coverage revisited condition *iff for all test properties L_i and for all atomic condition a_k in L_i we find some trace T_j such that $T_j \vdash L_i \rightsquigarrow P$ for some P where P is the shortest left-most proof and $a_k^{\surd}$ is in P.*

Based on Theorem 2 it immediately follows that the Unique First Cause Coverage Revisited Condition is checkable.

5 Related Work and Conclusion

There are various prior works which study finite trace matching algorithms, e.g. see [3,5], and the design space of the semantics of finite trace LTL matching, e.g. see [2]. To the best of our knowledge, we are the first to study constructive finite trace matching. Such a matching approach has several advantages as discussed in the introduction.

Of particular interest is the application of checking coverage of LTL test properties. Our focus here is the UFC coverage condition introduced in [7]. We can give a precise definition of the UFC condition in terms of shortest left-most proofs and thus we can easily check if a test suite satisfies the UFC condition.

The LTL matching and coverage approach as described has been fully implemented and is in actual use in some mission-critical embedded system applications. We check coverage of LTL properties w.r.t. manually written test cases. As our implementation language we use Haskell which fits very well the rewriting nature of our matching algorithms. We incorporate several optimizations such

as hash consing for efficient comparison etc. Haskell's lazy evaluation strategy is of advantage in case of larger formulas with short proofs. Thanks to laziness we only need to evaluate the necessary parts. Due to space constraints, we postpone a more detailed description of our implementation and experiences from several industrial case studies to some future work.

Another interesting topic is the issue of providing sensible explanation whys a trace does not match the formula. Currently, we simply return the *first* failure position in the trace and the formula. We believe that often there can be better, e.g. *shortest*, explanations. This is something we will pursue in future work.

Acknowledgements. We thank the reviewers for their comments.

References

1. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM 11(4), 481–494 (1964)
2. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with Temporal Logic on Truncated Paths. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003)
3. Jard, C., Jéron, T.: On-line model checking for finite linear temporal logic specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 189–196. Springer, Heidelberg (1990)
4. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
5. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automated Software Engg. 12, 151–197 (2005)
6. RTCA/DO-178B. Software considerations in airborne systems and equipment certification (1992)
7. Whalen, M.W., Rajan, A., Heimdahl, M.P.E., Miller, S.P.: Coverage metrics for requirements-based testing. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA 2006, pp. 25–36. ACM, New York (2006)

Towards Scenario-Based Testing of UML Diagrams*

Petra Brosch[2], Uwe Egly[1], Sebastian Gabmeyer[2], Gerti Kappel[2], Martina Seidl[3], Hans Tompits[1], Magdalena Widl[1], and Manuel Wimmer[2]

[1] Institute for Information Systems, Vienna University of Technology, Austria
{uwe,tompits,widl}@kr.tuwien.ac.at
[2] Business Informatics Group, Vienna University of Technology, Austria
{brosch,gabmeyer,gerti,wimmer}@big.tuwien.ac.at
[3] Institute of Formal Models and Verification, Johannes Kepler University, Austria
martina.seidl@jku.at

Abstract. In model-driven engineering, models are not primarily developed for documentation and requirement specification purposes, but promoted to first-class artifacts, from which executable code is generated. As a consequence, typical development activities like testing must be performed on the model level. In this paper, we propose to use overlapping information inherent in multiple views of models for automatic testing. Using a prototype based on the model checker SPIN we show the feasibility of this approach and identify future challenges.

1 Introduction

Multi-view modeling languages like UML [6] offer different diagram types to lower the complexity of describing software systems. Each diagram provides a distinct view on the system, allowing for splitting a complex model into various areas of concern [4]. In that way, the diagrams complement one another, altogether providing a holistic representation of the system. The views are connected by information redundant in the different diagrams and consistency has to be assured [4]. In this paper, we investigate how this information can be used as test data.

Consider the following example modeled in Fig. 1. Two state machines show a typical behavior of a PhD student (PhD) and a coffee machine (CM). Both state machines change their states according to the messages they receive. Conditions for the state transitions are given in terms of transition labels. The transition labels consist of two parts: The left part denotes an action triggering the transition, and the right part indicates a set of actions performed during the transition. If no triggering action is defined ("$-$"), the transition is executed unconditionally. Starting in state $Tired$, the PhD student turns the coffee machine on and optimistically waits until it is ready. If she receives the $error()$ message, she becomes desperate, then tired and tries again. Otherwise, she is happy, demands coffee, and waits until it is completed. The sequence diagram in Fig. 1 models a forbidden scenario inside a neg fragment: After the coffee machine has sent an error, it

* This work was partially funded by the Vienna Science and Technology Fund (WWTF) through project ICT10-018, by the fFORTE WIT Program of the Vienna University of Technology and the Austrian Federal Ministry of Science and Research, and by the Austrian Science Fund (FWF) under grants P21698, J3159-N23, and S11409-N23.

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 149–155, 2012.
© Springer-Verlag Berlin Heidelberg 2012

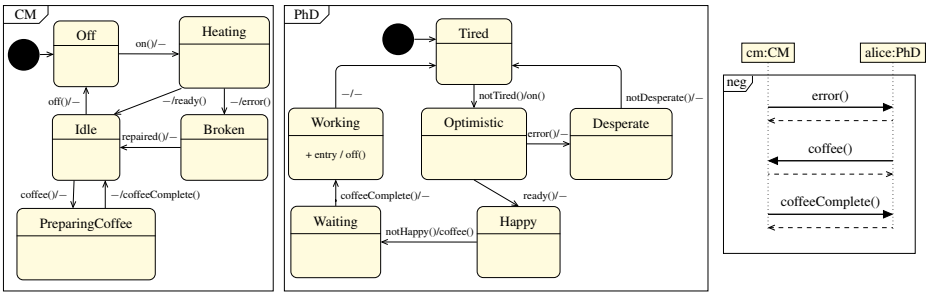


Fig. 1. Sequence diagram modeling a forbidden interaction for two state machines

receives a coffee request and then sends a $coffeeComplete()$ message. Obviously, during the parallel execution of the state machines the forbidden sequence can occur.

In model-driven engineering, models are not only used as mere design documents but they serve as artifacts from which code is generated. It is important to detect faults in the models; otherwise they may propagate to the code. Designing test cases on the model level has been subject to extensive research, but often testing itself is transferred to the code level or requires a simulation engine. To circumvent this problem, we use communication scenarios modeled in sequence diagrams. Testing is thus shifted to model level. Hence, serious design and implementation errors in the model are detected at an early point in time by using the information available due to multi-view modeling.

We start from a restricted subset of UML state machines and sequence diagrams for which we provide a formal description. This description is designed in such a way that it is extensible. The concept of model checking UML interactions has been described in [8]. We formulate an alternative encoding more natural for our use case with multiple communicating state machines. For experimental evaluation, we built a first prototype with PROMELA, the input language of SPIN, a highly configurable, state-of-the-art model checker. This allows us to derive challenges which have to be solved to put our vision of testing multi-view models into practice.

2 Preliminaries

We consider a subset of the UML state machine and sequence diagrams modeling only forbidden scenarios. Note that, to model forbidden scenarios, we consider only sequences that are enclosed in a neg fragment. The model is consistent if the sequences given in the sequence diagrams do not occur on any path of the state machines executing in parallel. The problem is formally defined as follows: A *software model* is a triple $(\mathcal{M}, \mathcal{S}, \mathcal{A})$ where $\mathcal{M}$ is a set of state machines, $\mathcal{S}$ is a set of sequence diagrams, and $\mathcal{A}$ is a set of actions, including the empty action ϵ, necessary to model that a transition is triggered by a completion event (denoted by "–" in Fig. 1). Note that we omitted the actions within the states which cause the completion event, because they are not relevant for our purposes. For example, Fig. 1 shows a software model of two state machines and one sequence diagram. The set of actions comprises all method calls indicated on the transitions and entry or exit actions inside states.

Definition 1. *A* state machine *is a tuple* $M = (S, \iota, A^T, A^P, T)$*, where* S *is a set of* states*,* $\iota \in S$ *is a designated* initial state*,* $A^T \subseteq \mathcal{A}$*,* $A^P \subseteq \mathcal{A}$*, and* $T \subseteq S \times A^T \times \mathcal{P}(A^P) \times S$ *is a transition relation. Each transition contains a triggering action* $a \in A^T$*, called* event*, which triggers a state transition, and a set* $B \in \mathcal{P}(A^P)$ *of actions, called* effects*, which are performed when the transition is executed.*

Note that this definition also handles *entry* and *exit* actions defined inside states: An entry action in state s_i is included in the effects of each incoming transition to s_i, and an exit action in the effects of each outgoing transition from s_i.

Figure 1 shows six states for the state machine $M_{PhD} = (S, \iota, A^T, A^P, T)$. The initial state $\iota = \mathit{Tired}$ is denoted by an incoming transition from the black circle. The transition labels consist of two parts, separated by a backslash. The set A^T contains the string on the left side of the transition labels, and A^P the set of strings indicated on the right side of the transition labels or as entry or exit actions. For example, the transition from $\mathit{Waiting}$ to $\mathit{Working}$ is $(\mathit{Waiting}, \mathit{coffeeComplete}(), \{\mathit{off}()\}, \mathit{Working})$.

Definition 2. *A neg fragment in a* sequence diagram $S \in \mathcal{S}$ *is a triple* (L, m, N)*, where* L *is a set of* lifelines*,* $m : L \to \mathcal{M}$ *is a bijective function assigning a state machine to each lifeline, and* N *is a forbidden sequence of triples* $L \times \mathcal{A} \times L$*.*

In the sequence diagram of our running example, there are two lifelines, cm and $alice$. The state machine assigned to *alice* is $m(alice) = PhD$. The sequence of messages is $N = (\langle cm, error(), alice\rangle, \langle alice, coffee(), cm\rangle, \langle cm, coffeeComplete(), alice\rangle)$.

The behavior of a set $\mathcal{M}$ of synchronously communicating parallel state machines is defined as the composition $\mathcal{M}_{||}$ as follows [2].

Definition 3. *Let* $M_k = (S_k, \iota_k, A_k^T, A_k^P, T_k)$*,* $k \in \{1, \ldots, n\}$*, be* n *state machines, let* $A_k = A_k^T \cup A_k^P$*, and let two state machines* M_i, M_j*,* $i \neq j$*, synchronize over all actions in* $H_{ij} = ((A_i^T \cap A_j^P) \cup (A_j^T \cap A_i^P)) \backslash \{\epsilon\}$ *such that communication is pairwise, i.e.,* $H_{ij} \cap A_l = \emptyset$ *for* $l \notin \{i, j\}$ *(obviously,* $H_{ij} = H_{ji}$*). Then, the* composition *of a set* $\mathcal{M}$ *of state machines is given by* $\mathcal{M}_{||} = (S_1 \times \ldots \times S_n, \langle \iota_1, \ldots, \iota_n\rangle, A_1 \cup \ldots \cup A_n, R)$ *with*

1. $(\langle s_1, \ldots, s_i, \ldots, s_n\rangle, a, \langle s_1, \ldots, s_i', \ldots, s_n\rangle) \in R$ *iff* $(s_i, a, E, s_i') \in T_i$ *and* $a \in (A_1 \cup \ldots \cup A_n) \setminus \bigcup_{0<j\leq n, i\neq j} H_{ij}$ *with* $1 \leq i \leq n$*. The action* a *is called* local*.*
2. $(\langle s_1, \ldots s_i, \ldots, s_j, \ldots, s_n\rangle, b, \langle s_1, \ldots, s_i', \ldots, s_j', \ldots, s_n\rangle) \in R$ *iff* $b \in H_{ij}$ *and there exist transitions* $(s_i, v, B, s_i') \in T_i$ *and* $(s_j, b, G, s_j') \in T_j$ *with* $b \in B$ *and* $1 \leq i, j \leq n$ *and* $i \neq j$*. The action* b *is called* global*.*

Definition 4. *A sequence* $\pi = \langle a_1, a_2, \ldots, a_l\rangle$ *is a* path *in* $\mathcal{M}_{||} = (S, \iota, A, R)$ *iff there exist triples* $(s, a_i, s'), (s', a_{i+1}, s'') \in R$ *for all* i *where* $1 \leq i < l$*. A software model* $(\mathcal{M}, \mathcal{S}, \mathcal{A})$ *is* consistent *iff for any neg fragment* (L, m, N) *in any sequence diagram with* $N = \langle n_1, n_2, \ldots, n_k\rangle$ *and* $n_j = (l_j, a_j, l_j')$ *for all* j *where* $1 \leq j < k$*, its sequence of actions* $\langle a_1, a_2, \ldots, a_k\rangle$ *does not occur as subsequence of any path in* $\mathcal{M}_{||}$*.*

3 Formulation of the Model Checking Problem

Inspired by previous work [8], we use the model checker SPIN [7] and its input language PROMELA to verify whether a set of state machines fulfills a safety property described

as *neg* fragment of a sequence diagram. To this end, we encode the state machine as a set of active PROMELA processes and the *neg* fragment as `notrace` assertion. In verification mode, SPIN checks whether the behavior specified in the assertion occurs on any execution trace of the processes executing in parallel. If this is the case, SPIN returns the erroneous execution path on which the `notrace` behavior occurred. Otherwise, it returns no error. We evaluated this approach on several examples. The following elements of PROMELA are relevant for our encoding: `active proctype` (process behavior automatically instantiated at program start), `label` (identifier of a unique control state), `mtype` (declaration of symbolic names for constant values), `chan` (asynchronous or synchronous channel), and `notrace` (assertion defining unwanted sequences of channel activities). A software model $(\mathcal{M}, \mathcal{S}, \mathcal{A})$ is encoded in PROMELA as follows: Each action label $a \in \mathcal{A} \setminus \epsilon$ is encoded as an element of `mtype`. For each state machine $M \in \mathcal{M}$ we define an `active proctype` and a synchronous global channel `chan` of type `mtype`. Each `active proctype` representing a state machine $M = (S, \iota, A^T, A^P, T)$ contains a `label` for each state $s \in S$. The label representing ι is placed at the beginning of the process to be executed first. For each state machine, each transition $T = (s_i, a, B, s_j)$ is implemented within the PROMELA label representing state s_i: A transition consists of a receive statement for a if $a \neq \epsilon$ or nothing otherwise, a statement for each $b \in B \setminus \epsilon$ or nothing if $B = \{\epsilon\}$, and a `goto` statement directing to the label representing s_j. If s_i has more than one outgoing transition, the set of transitions is put inside an `if` statement. The sequence of messages on each lifeline is encoded as PROMELA `notrace` assertion. A `notrace` assertion is defined over some or all global channels and monitors all actions on these channels during program execution. When all channel actions defined by the assertion have been executed, an error is returned. Note that `notrace` assertions can contain `accept` labels to model forbidden infinite behavior. The encoding of Fig. 1 is available online at `http://www.modelevolution.org/media/scenario-based-testing/coffee.pml`.

4 Related Work

In the following, we focus on works which present results on the successful application of verification techniques for multi-view system specifications. Cimatti et al. [5] use Hybrid Automata (HA) to describe a system of message exchanging components and verify the system against a scenario-based specification modeled with a Message Sequence Chart (MSC). They present an extension to bounded model checking using k-induction to prove that there exists no trace which satisfies a given scenario. Li et al. [9] use MSCs as scenario-based specifications for concurrent systems modeled with Petri nets and discuss an approach to check if a Petri net either satisfies a mandatory scenario on all of its traces, a forbidden scenario on none of its traces, or a dependent scenario on all traces once a given, other scenario is satisfied. The CHARMY tool suite [10] offers a modeling, simulation, and verification environment for software architectures (SA). SAs describe the static and behavioral structures of systems with component, state transition, and sequence diagrams. CHARMY employs SPIN and translates the SA to PROMELA to detect deadlocks and unreachable states. The work most closely related to ours is the one by Schäfer et al. [11]. They propose to verify a set of message-exchanging

state machines against a specification described by UML collaboration diagrams. They implement their approach in HUGO, which automatically translates the state machine diagrams to PROMELA and generates Büchi automata, so-called "never claims", from the collaboration diagrams. The generated artifacts form the input for SPIN, which performs the verification. Knapp and Wuttke [8] extend the approach of Schäfer et al. [11] to accommodate UML 2.0 sequence diagrams. Their encoding focuses on integrating many language concepts, while we present an encoding suitable for our testing use case.

Another, more widely related research area is the synthesis of state machines from sequence diagrams. Synthesis aims at automatically deriving design models from requirements given as scenarios, as described by Whittle and Schumann [14]. An extension of the latter synthesis algorithm is proposed by Grønmo and Møller-Pedersen [13] by considering also combined fragments in sequence diagrams. The synthesis of model transition systems from scenarios is discussed by Uchitel et al. [12] who also consider safety properties besides scenarios. Common to all these approaches is that the consistency between the scenarios and the state machines are given by construction. However, the synthesis rules may form an important input for the extension of our approach.

5 Discussion and Future Challenges

In this paper, we discussed the use of model checking to detect errors in multi-view system specifications expressed with UML diagrams. We employ sequence diagrams to model test cases, which express forbidden scenarios with *neg* fragments. As this allows us to perform testing on the level of models, modelers remain within one level of abstraction. Our current prototype is a proof of concept and restricted to the modeling language elements discussed in this paper. Yet, it serves as test bed for various interesting application scenarios. In future work, we plan to integrate positive scenarios in sequence diagrams and additional constructs of state machines, like hierarchies, asynchronous communication, or transition guards, into our framework. Also, other techniques to assemble the information of the sequence diagram and more advanced encodings (including model checkers other than SPIN) will be considered. Further, we intend to compare our encoding with the one of Knapp and Wuttke [8] with respect to scalability and ease of information extraction. We conclude with lessons learned from building our prototype.

Variations in Semantics. The UML standard's informal definition of its diagrams' semantics leaves much room for varying and even contradicting interpretations. For example, a scenario modeled by a sequence diagram describing the interaction of a set of parallel state machines may be interpreted such that either (i) at least one execution path over the set of state machines must satisfy the scenario, (ii) all possible execution paths must satisfy the scenario, or (iii) the occurrence of the scenario's first element implies the occurrence of all subsequent elements on *all* execution paths. By its very nature, the encoding provides one such interpretation that has to eliminate all semantic variation points. This in turn requires a rigorous formalization of the UML standard which should incorporate the smallest set of unambiguous constructs that retain a maximum of the UML's expressiveness. Presently, we started from a simplified version of UML with a concise semantics, but for more language features we will consider works on UML formalization [1,3].

Incomplete Information. In general, models do not describe a system in full detail, but capture only certain aspects. This way, the modeler is not distracted by temporarily irrelevant details. For building an executable system, the missing information is then gathered in multiple refinement steps, eventually at the code level. For automated testing, this kind of information may be necessary and has therefore to be collected.

State-Space Explosion. The most significant problem in model checking is the large state space to be searched. To shrink the state space, techniques like partial order reduction have been proposed, where equivalent traces are considered only once. Although implemented in model checkers like SPIN, we assume that such optimizations may be performed at the encoding level by exploiting particularities of the modeling language.

Co-Evolution of Code. So far, we have treated sequence diagrams as a visualization of safety properties. Alternatively, sequence diagrams may be used as visualizations of excerpts of a program. Then, the role of sequence diagrams and state machines is inverted, and sequence diagrams are verified against the state machine. In this manner, we shift the focus to the detection of inconsistencies between the model and the code, which may be introduced due to the evolution of the software system.

Presentation Issues. When a model checker determines that a specification is not satisfied, it returns a counterexample, which explains the cause of the problem. Providing an adequate representation of the counterexample, e.g., in the concrete syntax of the employed modeling language, is indispensable for user-friendliness.

References

1. de Boer, F.S., Bonsangue, M.M., Steffen, M., Ábrahám, E.: A Fully Abstract Semantics for UML Components. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 49–69. Springer, Heidelberg (2005)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
3. Broy, M., Cengarle, M.: UML Formal Semantics: Lessons Learned. SoSyM 10(4) (2011)
4. Rivera, J., Romero, J., Vallecillo, A.: Behavior, Time and Viewpoint Consistency: Three Challenges for MDE. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 60–65. Springer, Heidelberg (2009)
5. Cimatti, A., Mover, S., Tonetta, S.: Proving and Explaining the Unfeasibility of Message Sequence Charts for Hybrid Systems. In: FMCAD (2011)
6. OMG. Unified Modeling Language (UML), Superstructure V2.4.1 (August 2011), http://www.omg.org/spec/UML/2.4.1/
7. Holzmann, G.J.: The Model Checker SPIN. TSE 23(5), 279–295 (1997)
8. Knapp, A., Wuttke, J.: Model Checking of UML 2.0 Interactions. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 42–51. Springer, Heidelberg (2007)
9. Li, X., Hu, J., Bu, L., Zhao, J., Zheng, G.: Consistency Checking of Concurrent Models for Scenario-Based Specifications. In: Prinz, A., Reed, R., Reed, J. (eds.) SDL 2005. LNCS, vol. 3530, pp. 298–312. Springer, Heidelberg (2005)
10. Pelliccione, P., Inverardi, P., Muccini, H.: CHARMY: A Framework for Designing and Verifying Architectural Specifications. TSE 35(3), 325–346 (2008)
11. Schäfer, T., Knapp, A., Merz, S.: Model Checking UML State Machines and Collaborations. ENTCS 55(3), 357–369 (2001)

12. Uchitel, S., Kramer, J., Magee, J.: Synthesis of Behavioral Models from Scenarios. TSE 29(2), 99–115 (2003)
13. Grønmo, R., Møller-Pedersen, B.: From UML 2 Sequence Diagrams to State Machines by Graph Transformation. JOT 10(8), 1–22 (2011)
14. Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios. In: ICSE, pp. 314–323. ACM (2000)

Evaluating and Debugging OCL Expressions in UML Models

Jens Brüning[1], Martin Gogolla[2], Lars Hamann[2], and Mirco Kuhlmann[2]

[1] University of Rostock
[2] University of Bremen

Abstract. This paper discusses the relationship between tests and proofs with focus on a tool for UML and OCL models. Tests are thought of as UML object diagrams and theorems or properties which are to be checked are represented as OCL constraints, i.e., class invariants or operation pre- and postconditions. The paper shows for the UML and OCL tool USE (UML-based Specification Environment) how to trace and debug the validity of an expected theorem (an OCL constraint) within a given test case (a state model in the form of a UML object diagram).

1 Introduction

A central issue in the relationship between tests and proofs is the question which part of a test affects which part of a proof or a theorem to be proven. Tests as well as proofs and the underlying theorems are highly structured entities with many important relationships, not all being relevant in a specific situation during development. For example, for proof counter-examples it is important to know which part of the expected proof or theorem is falsified by the counter-example, and it is important for the developer to find the respective parts of the test and the proof or theorem in an adequate way.

This paper discusses this general question with focus on a tool for UML and OCL models. Tests are thought of as UML object diagrams and theorems or properties which are to be checked are represented as OCL constraints, i.e., class invariants or operation pre- and postconditions. The paper shows for the UML and OCL tool USE (UML-based Specification Environment) [GKH09] how to trace and debug the validity of an expected theorem (an OCL constraint) within a given test case (a constructed state model in form of a UML object diagram). The technical realization in the tool is done by a so-called evaluation browser which allows the developer to debug the evaluation of a complex OCL expression and its subexpressions with respect to a given system state in a user-friendly way with the aim of better understanding, for example, invariant failure.

2 Basic Evaluation Browser Concepts by Example

Let us introduce the basic idea of our approach by means of an example. The USE screenshot in Fig. 1 shows in the upper row an OCL and UML model with

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 156–162, 2012.
© Springer-Verlag Berlin Heidelberg 2012

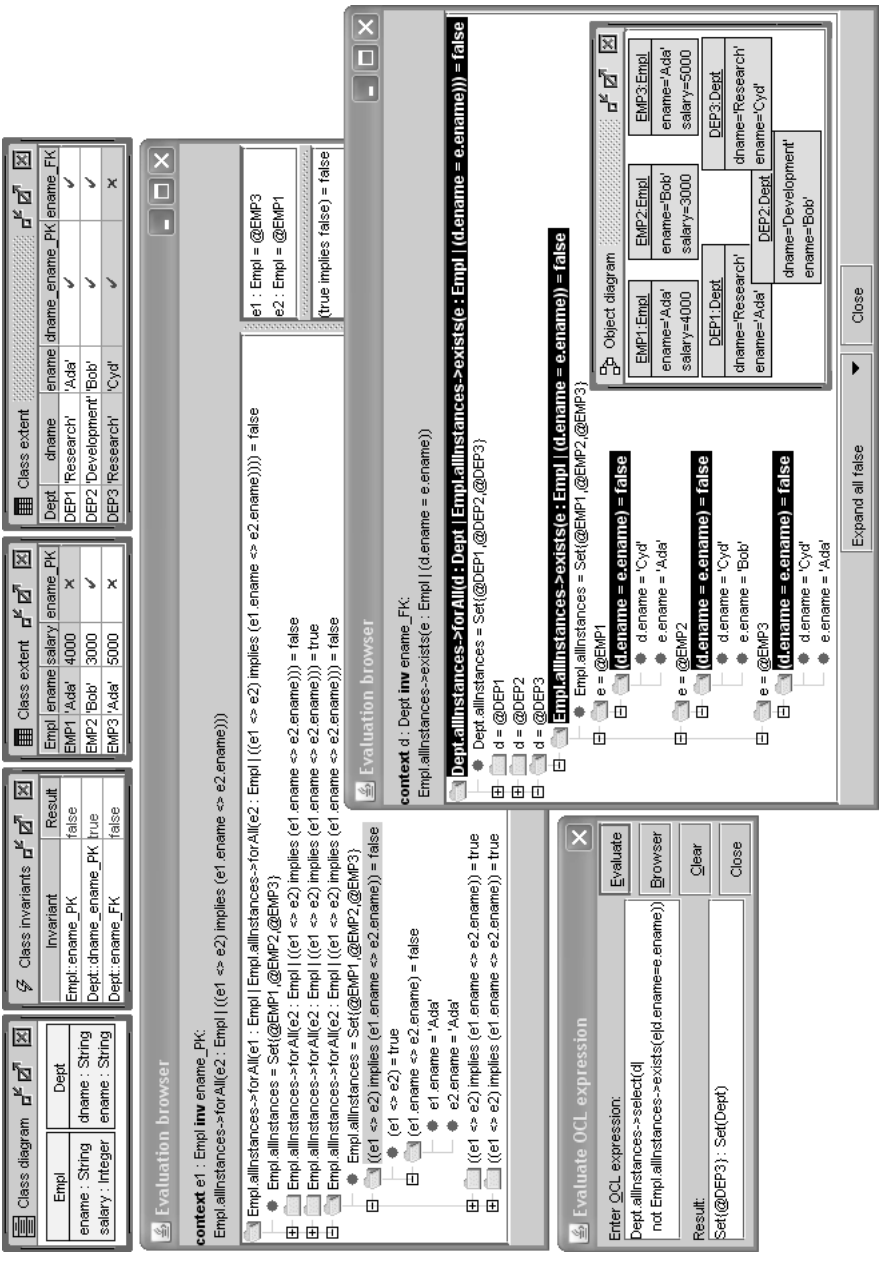


Fig. 1. Basic Use of Evaluation Browser

invariants and a corresponding state in form of 2 Class extent windows (these 2 windows determine a UML object diagram displayed in the lower right). The class diagram represents a simple relational database schema with two tables (`Empl[oyee]`, `Dep[artmen]t`), two primary key constraints ({`ename`} is primary key in `Empl`, {`dname, ename`} is primary key in `Dept`) and one foreign key constraint (`Dept.ename` references `Empl.ename`). The OCL details of one primary key constraint and the foreign key constraint are shown in the top part of the 2 Evaluation browser windows, respectively.

As shown in the Class invariants window, the (database) state represented in the Class extent windows does violate 2 of the specified constraints. In order to understand the reason for the violation, USE allows to open so-called Evaluation browser windows. In the screenshot we see one Evaluation browser window for the failing primary key constraint `ename_PK` and one for the failing foreign key constraint `ename_FK`. The windows have been configured in different ways to demonstrate the possibilities of our approach. For example, the first window shows variable substitutions in a subwindow in the very right, the second window shows the variable substitutions inside the main Evaluation browser window. OCL expressions evaluating to `false` are highlighted in the second window in a white-on-black style, whereas they are displayed without special indication in the first window. In this simple situation, the analysis could be done by simple inspection without the evaluation browser, but we want to demonstrate the approach with an easy understandable example. We will show below an involved situation hard to understand by simple inspection.

In the first Evaluation browser window, which can be opened by double-clicking in the Class invariants window the failing primary key invariant `ename_PK`, one subformula which evaluates to `false` is highlighted in grey. Three lines below the highlighted grey line, we see that the OCL terms `e1.ename` and `e2.ename` both evaluate to `'Ada'`. The variable substitutions responsible for this evaluation are stated in the right subwindow (basically stating `e1=EMP3, e2=EMP1`) and the evaluation of the selected and highlighted subexpression is displayed below the substitutions on the right (`(true implies false)=false`). Thus, this Evaluation browser window displays one concrete counter-proof for the expected primary key property `ename_PK`: `EMP3` and `EMP1` are distinct, but their ename values coincide, and this violates the primary key requirement.

In the second Evaluation browser window, the foreign key constraint is analyzed. Only those subformulas evaluating to `false` are displayed and are pictured in a white-on-black style. Below the central highlighted exists subformula it is shown that for the `Dept` object `DEP3` having ename attribute value `'Cyd'` no corresponding `Empl` object exists having the ename attribute value `'Cyd'`. Thus, the Evaluation browser window again displays one concrete counter-proof for the expected foreign key property `ename_FK`.

3 General Features Available in the Evaluation Browser

Our so-called evaluation browser pictures the evaluation of an OCL term in a graphical style as a tree. The tree nodes show OCL terms or subterms of the

original term together with values of subterms and substitutions for occurring variables. Tree subbranches may be opened or closed interactively through the user or by setting particular configuration parameters. Particular tree parts may be highlighted in color or in a white-on-black style. The aim of the evaluation browser is to offer an intuitive, highly configurable, and flexible tool for analyzing the evaluation of complex OCL terms. As indicated in Fig. 2, there are basically six central configuration parameters for the OCL evaluation browser (basically available by right-clicking into the Evaluation browser's pane):

(A) Determination of opened subbranches of the tree.
(B) Turning the extended OCL formula evaluation on or off.
(C) Turning an additional variable assignment subwindow on or off.
(D) Turning an additional subexpression evaluation subwindow on or off.
(E) Positioning of variable assignments in the main evaluation term.
(F) Determining the highlighting of subformulas evaluating to particular values.

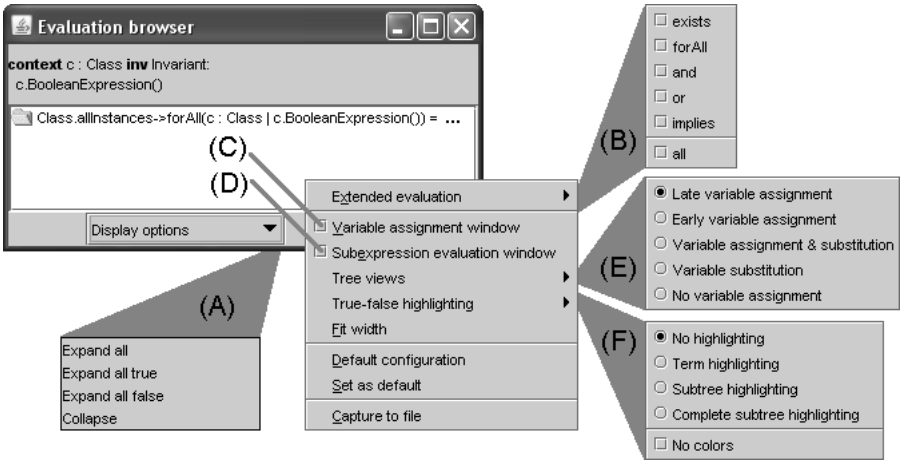


Fig. 2. General Features Available in Evaluation Browser

In (A) the developer determines the basic structure of opened or closed tree subbranches. Either all subbranches, all subbranches evaluating to `true`, all subbranches evaluating to FALSE are opened or no subbranch is opened. In (B) an extended evaluation of OCL subformulas is configured. In the standard evaluation of OCL for the exists quantifier the evaluation stops with `true`, if the first satisfying element is found. However, one frequently wants to know all elements satisfying the exists predicate. This can be accomplished by turning on the extended evaluation for exists. Analogous possibilities are provided for the other logical operations. In (C) an explicit subwindow for the variable assignments is opened. In (D) an explicit subwindow for the subexpression evaluation is opened. In (E) the position of variable assignments in the tree is fixed.

The variable assignments may be placed at the tree leafs ('Late') or inside the tree as early as they appear ('Early'). More options are available. In (F) the highlighting of `false` subformulas resp. `true` subformulas is determined.

4 Further Features Available in the Evaluation Browser

The second USE screenshot in Fig. 3 shows the evaluation browser being used for an automatically generated object diagram (test case) which is the result of an ASSL procedure [GKH09]. The randomly generated object diagram represented by 2 Class extent windows in the left upper part of the screenshot involves 16 objects with respective attribute values. As the class invariants window in the right shows, all invariants fail. The Evaluation browser window has been opened through double-clicking the failing invariant `Dept::ename_FK`. This window shows all details, i.e., all reasons, why this invariant fails. The complete evaluation tree has 3 subbranches evaluating to `false` and exactly these 3 subbranches have been opened and are displayed as white-on-black. The 3 subbranches indicate that the 3 objects `Dept2`, `Dept4`, and `Dept8` are the reason for

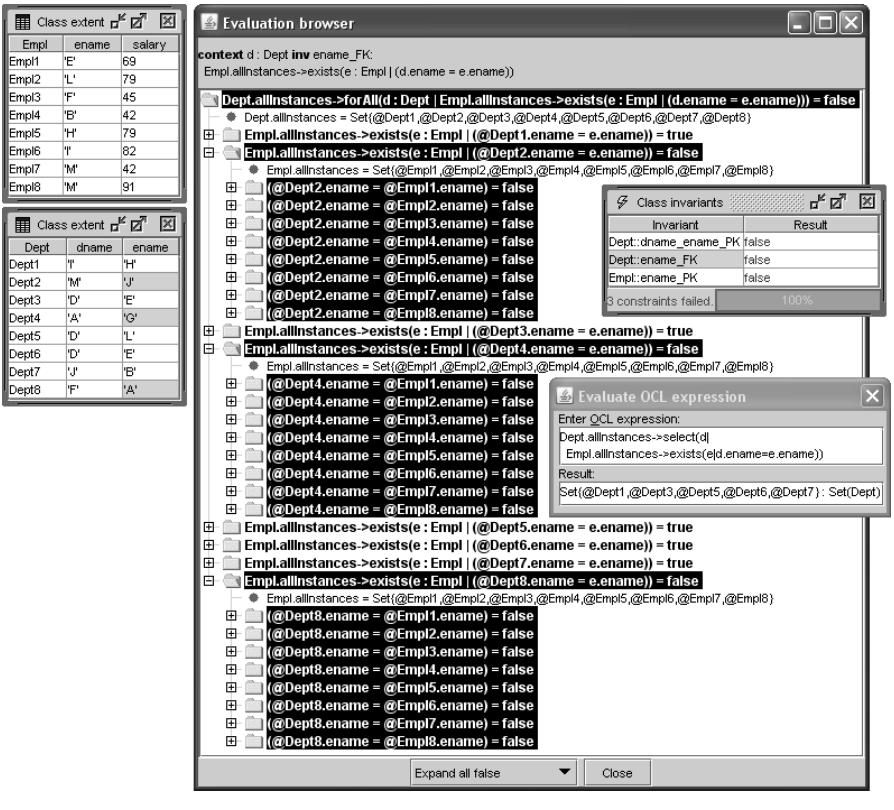


Fig. 3. Further Use of Evaluation Browser

the invariant failure. Checking these objects against all Dept objects in the second Class extent window one learns that the respective ename values (`'J', 'G', 'A'`) indeed cannot be found as ename values in the first Class extent window for Empl objects. In this evaluation browser configuration, variable substitutions are not displayed, but variables have been substituted by their values.

The Evaluate OCL expression window on the right is a cross-check against the found result. This OCL expression retrieves all Dept objects which possess a corresponding Empl object having the same name. It returns the complement `Dept` object set `Dept1`, `Dept3`, `Dept5`, `Dept6`, and `Dept7`. This screenshot is an explanation why the 'theorem' (i.e., the invariant `Dept::ename_FK`) fails in this test case (i.e., in this object diagram). Such a detailed analysis is needed during development when unexpected results in form of failing constraints occur in order to understand the reason for constraint failure.

5 Related Work

We only point to a few works on debugging in the context of model-based development and declarative languages. In [SSJ+03] counter-example generation is understood as debugging. [GHMGB07] discusses model-level debugging for software architectures. Initial ideas towards model-based debugging are proposed in [MS08]. [KSWR09] discusses a connection between debugging and QVT. The work in [RVM10] proposes a debugger for the specification language Maude.

6 Conclusion

This paper has made a proposal for debugging OCL invariants in UML models. Debugging works by means of term evaluation. The display of the evaluation term may be adjusted by the developer in various ways. The ideas of the proposal could be used for other declarative languages as well. Up to now there are too few proposals for user-friendly debugging in the context of theorem proving or theorem checking. Our approach currently works for invariants only and has to be extended for pre- and postconditions. Further options for configuring the evaluation tree are imaginable, for example, grouping of subbranches with similar results. Larger case studies must give feedback on the usability of the proposal.

References

[GHMGB07] Graf, P., Hübner, M., Müller-Glaser, K.D., Becker, J.: A Graphical Model-Level Debugger for Heterogenous Reconfigurable Architectures. In: Bertels, K., Najjar, W.A., van Genderen, A.J., Vassiliadis, S. (eds.) FPL, pp. 722–725. IEEE (2007)

[GKH09] Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 90–104. Springer, Heidelberg (2009)

[KSWR09] Kusel, A., Schwinger, W., Wimmer, M., Retschitzegger, W.: Common Pitfalls of using QVT Relations - Graphical Debugging as Remedy. In: ICECCS, pp. 329–334. IEEE Computer Society (2009)

[MS08] Mayer, W., Stumptner, M.: Evaluating Models for Model-Based Debugging. In: ASE, pp. 128–137. IEEE (2008)

[RVM10] Riesco, A., Verdejo, A., Martí-Oliet, N.: A Complete Declarative Debugger for Maude. In: Johnson, M., Pavlovic, D. (eds.) AMAST 2010. LNCS, vol. 6486, pp. 216–225. Springer, Heidelberg (2011)

[SSJ+03] Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging Overconstrained Declarative Models using Unsatisfiable Cores. In: ASE, pp. 94–105. IEEE Computer Society (2003)

A Framework for the Specification of Random SAT and QSAT Formulas*

Nadia Creignou[1], Uwe Egly[2], and Martina Seidl[3,4]

[1] Laboratoire d'Informatique Fondamentale CNRS UMR 7279, Aix-Marseille Université, France
[2] Institut für Informationssysteme 184/3, Technische Universität Wien, Austria
[3] Institute for Formal Models and Verification, Johannes Kepler University, Austria
[4] Institut für Interaktive Systeme 188/3, Technische Universität Wien, Austria

Abstract. We present the framework [q]bfGen which allows the declarative specification of random models for generating SAT and QSAT formulas not necessarily in (prenex) conjunctive normal form. To this end, [q]bfGen realizes a generic formula generator which creates formula instances by interpreting the random model specification expressed in XML. Consequently, the implementation of specific random formula generators becomes obsolete, because our framework subsumes their functionality.

1 Motivation

Over the last years, tools for solving the satisfiability problem of propositional logic (SAT) showed to be powerful backend engines for various hardware and software verification problems [14]. The same hope is pined on extensions like QSAT, where quantifiers are introduced over the propositional variables allowing more succinct encodings of verification problems [2]. So far, QBF solvers have not reached the same maturity as SAT solvers in terms of efficiency and stability. Techniques which showed to be useful in SAT can often not directly be transferred to QBF. Whereas in SAT conjunctive normal form is the canonical input format, the pendant for QSAT, the prenex conjunctive normal form (PCNF), is not the commonly accepted representation format. In fact, the transformation to PCNF might negatively influence the behavior of a solver [6]. Consequently, QSAT solvers have been developed which process non-PCNF formulas, i.e., formulas of less restricted structure [7,10,11].

In SAT as well as in QSAT, random formulas find their *raison d'être* justified in two different use cases. From a theoretical point of view, random formulas provide the basis for investigations on properties like the phase transition phenomenon [9,8,4,5]. From a practical point of view, they are an important tool

* This work was partially funded by the Vienna Science and Technology Fund (WWTF) through project ICT10-018, by the Austrian Science Fund (FWF) under grant S11409-N23 and by the Agence Nationale de la Recherche under grant ANR-09-BLAN-0011-01.

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 163–168, 2012.
© Springer-Verlag Berlin Heidelberg 2012

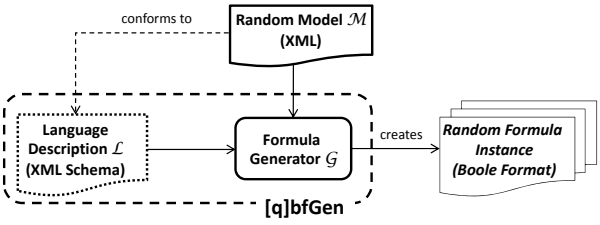


Fig. 1. The basic architecture of [q]bfGen

for testing and evaluating solvers. In particular, random formulas are used for fuzz testing which supports the automatic detection of various kinds of defects in solvers [3] by random inputs.

Random models provide control mechanisms for randomly generating formulas of a certain structure and size. The regularity of the formula structure allows for a characterization by statistical and combinatorial means, which in turn allows for a prediction of properties such as satisfiability and computational difficulty. Since most state-of-the-art solvers process formulas in (prenex) conjunctive normal form only, also most random models describe and generate formulas in PCNF. This restricted structure allows only a small set of parameters like number and distribution of variables, clause sizes, and the probability that a variable occurrence is negated to vary in the random model.

With the advent of non-PCNF solvers, also random models are required which generate formula instances of less restricted structure and which, consequently, introduce additional degrees of variability like the nesting depth of the formula tree. To support the specification of such random models, we introduce the framework [q]bfGen which provides a dedicated language for the description of random models as well as a generic formula generator which creates random formula instances according to such descriptions. By using [q]bfGen, the functionality of specific random formula generators like [13,9,4] can be realized by giving simple declarative descriptions of the according random model. For demonstration purposes, we extend the shape model of Navarro and Voronkov [13] for quantified Boolean formulas (QBF).

2 The Architecture of [q]bfGen

Our framework [q]bfGen allows the description of SAT and QSAT random models in XML from which formula instances are directly created. The current prototype uses the Boole format[1] for the the representation, in future implementations we consider to support also other output formats. As illustrated in Fig. 1, [q]bfGen consists of two main components: (i) the *language specification* $\mathcal{L}$ and (ii) the *formula generator* $\mathcal{G}$. Within [q]bfGen a random model must be expressed in conformance to $\mathcal{L}$. The resulting random model description $\mathcal{M}$ is then passed

[1] http://www.qbflib.org/boole.html

to $\mathcal{G}$, which generates random formula instances according to $\mathcal{M}$. Finally, these formula instances are provided to a SAT/QBF solver and evaluated. In the following, both $\mathcal{L}$ and $\mathcal{G}$ are presented in detail.

The Language Specification $\mathcal{L}$. In our implementation, $\mathcal{L}$ is realized as XML Schema. For ease of presentation, we use the notation of the UML Class Diagram to visualize a selection of concepts provided by $\mathcal{L}$. In Fig. 2, we show a simplified Class Diagram of the language specification $\mathcal{L}$. Each random model has one single element Root, which contains an arbitrary number of parameters and the actual formula. A Parameter element has a unique name within a random model and is characterized by a minimum value attribute, a maximum value attribute, and a step width attribute. With parameters it is possible to specify iterations for generating multiple formula instances with different settings. A Formula is either a Quantified Formula or a Connective Formula. A Quantified Formula has a unique name and introduces a new quantifier scope of a specified size which is either of existential, of universal, or of random type. In the case of random type, the quantifier is randomly selected for each instance. The size is either a fixed number or it may be assigned by a parameter. For example, a QBF of the form $\forall x_1 x_2 x_3\ \phi$ may be described by a Quantified Formula where the name of the scope is x, the size is 3, the type is universal, and ϕ is a Formula. A Connective Formula is translated to a conjunction or a disjunction. These connectives are of arbitrary arity. For example $(\neg x \wedge y \wedge \phi)$ could be an instantiation of a conjunction where x and y are variables and ϕ may be a complex formula. The variables occurring in a Connective Formula are specified by a VarSet element which states the probability for a variable being negated as well as from which quantifier block how many variables shall be selected. When a random formula instance is created, it can be assumed that within all instantiations of a VarSet, each variable occurs at most once. To specify that the variables shall occur in all branches of the subformula where the VarSet is defined, the position attribute must be set to a positive integer (the relative distance from the current position in the formula tree). In this way, it is possible to ensure that a variable is only instantiated once within the subformula. An example for this feature follows in the next section. The Formula element contains an attribute duplicates. For example, if a conjunction shall contain three clauses of a certain size, then the clause is specified only once and the duplication attribute is set to 3.

The Formula Generator $\mathcal{G}$. With the language specification $\mathcal{L}$ formulated in XML Schema, specifications of random models are expressed in XML. For the creation of formula instances out of random model specifications, we provide the formula generator $\mathcal{G}$, which is implemented as a command line tool in Java using Apache XMLBeans. When $\mathcal{G}$ is started it requires arguments like the random model, a set name for the formulas, and the number of formula instances to be generated. First $\mathcal{G}$ parses the provided random model and checks if it is conformant to $\mathcal{L}$ and if no constraints are violated. Such constraints assert that no parameter name is used which has not been specified, that no minimum value is greater than a maximum value, etc. When the random model passed

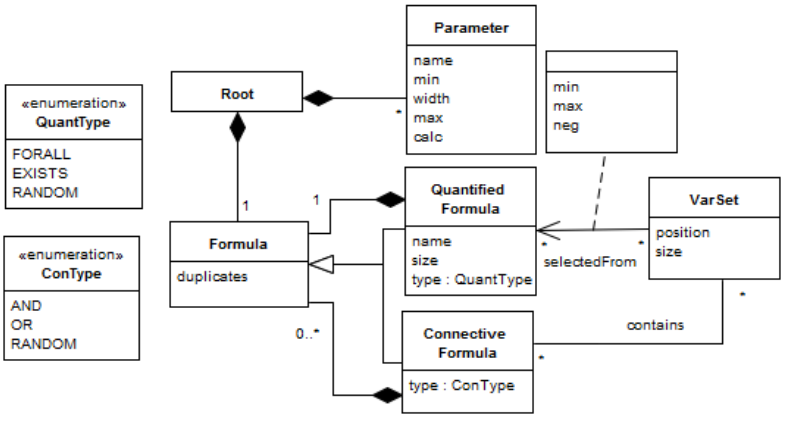


Fig. 2. Language specification

these tests, the formula instances are generated. Each formula instance is stored in an individual file. In the current version of $\mathcal{G}$, the output format is Boole, a standard format for QBF. If the `-p` flag is set, then no quantifiers are printed, i.e., the generated formulas are propositional formulas.

3 Tool Demo: The Fixed-Shape Model for QBF

In this section, we first present an extension of the fixed-shape model by Navarro and Voronkov [13] to QBF and then show how it is specified within our framework. For pedagogical reasons we focus on the specific $\langle 2,2,3\rangle$-*shape*. The method can then easily be extended to any *balanced shape* as defined in [13].

The Fixed-Shape Model for SAT and QBF. A $\langle 2,2,3\rangle$-*shape* is an alternating-$\{\vee,\wedge\}$-formula tree where the root node is a disjunction having two conjunctions as subformulas. Each of these conjunctions contains two clauses of size three. A $\langle 2,2,3\rangle$-*constraint* over the set of variables X is any instantiation of the above tree obtained by replacing the variables in the tree by possibly negated distinct variables from X. A $\langle 2,2,3\rangle$-*formula* is a conjunction of $\langle 2,2,3\rangle$-constraints. Observe that such formulas are in negation normal form

We are interested in creating random QBF instances of the form $\forall X \exists Y \phi$, where X and Y are sets of variables and ϕ is a $\langle 2,2,3\rangle$-formula. This extension of the fixed-shape random model introduced in [13] to quantified formulas requires the following additional parameters:

- The first parameter is the pair (m,n) specifying the number of variables in each quantifier block (size of X, size of Y).
- The second parameter is a pair (u,e), which fixes the number u of universal variables and the number e of existential variables that occur in each deepest, non-leaf subtree of every $\langle 2,2,3\rangle$-constraint of ϕ. Thus, $u+e=3$. Here we fix $u=1$ and $e=2$.
- The third parameter L is the number of $\langle 2,2,3\rangle$-constraints in ϕ.

Thus, we obtain a random (m, n)-$(1, 2)$-L-$\langle 2, 2, 3\rangle$-formula in choosing uniformly, independently and with replacement L constraints among all the possible ones that fulfill the above requirements. Note that the random model we propose is inspired by [8] and [4] for QBF in PCNF.

Realization in [q]bfGen. For the specification of the random model described above, the following steps are necessary:

1. To vary the ranges of X and Y, we specify two parameters m and n.
2. As we are interested in the probability that a formula instance is satisfiable when the ratio number of existential variables to number of constraints increases, we range our experiments over $L = rn$ where r is a real value. Therefore we introduce another parameter called width.
3. In the next step, we introduce a Quantified Formula element of universal type for X of size m which itself contains a Quantified Formula element of existential type for Y of size n.
4. Then we specify the outmost conjunction by a Connective Element which contains a description of the $\langle 2, 2, 3\rangle$-constraints. Note that it suffices to specify such a constraint only once, as the duplication may be achieved by the duplicates attribute which is set to the value of the parameter width.
5. The $\langle 2, 2, 3\rangle$-formula starts with a disjunction. Here we also specify a VarSet consisting of one variable selected from X and two variables selected from Y. The position attribute of this VarSet is set to two, stating that the variables are not inserted immediately, but in the clauses occurring two levels below in the formula tree. So we can insure, that these clauses do not share any variables.
6. Finally, we specify a conjunction containing a disjunction which are both duplicated twice realizing the two "2" in $\langle 2, 2, x\rangle$. The last disjunction is of arity three due to the literals obtained from the VarSet specified above.

We kindly refer to our project site [1] where we discuss the random model in detail and where we also show first experiments. Further (P)CNF random models from literature like [4,5,8,12] can also easily be handled.

4 Conclusion and Future Work

We introduced the framework [q]bfGen which provides a language for specifying SAT and QSAT random models and a formula generator which interprets such specifications and which creates formula instances accordingly. [q]bfGen is not only valuable in the context of empirical investigations of the properties of randomly generated formulas, but it is also a valuable tool within the solver development process. The randomly generated formulas may then serve as input data for fuzz testing [3], which is a powerful testing technique for such complex tools as solvers. So conceptual as well as programming bugs can be tracked down automatically and the robustness of the solvers increases. When used as backend engine for verification tasks, buggy solvers are worthless. Thus, [q]bfGen might become very helpful for the development of stable software.

We demonstrated our approach by describing a shape model for prenex QBF. The XML Schema, the prototypical implementation of the formula generator, and the detailed specification of the random model presented above are available at our project site [1].

In future work and driven by practical needs, we will include additional language elements like XOR and realize more advanced duplication and iteration mechanisms as provided at the moment.

References

1. qbfGen Project Site, http://fmv.jku.at/qbfgen/
2. Benedetti, M., Mangassarian, H.: QBF-Based Formal Verification: Experience and Perspectives. JSAT 5(1-4), 133–191 (2008)
3. Brummayer, R., Lonsing, F., Biere, A.: Automated Testing and Debugging of SAT and QBF Solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 44–57. Springer, Heidelberg (2010)
4. Chen, H., Interian, Y.: A Model for Generating Random Quantified Boolean Formulas. In: Proc. of IJCAI 2005, pp. 66–71. Professional Book Center (2005)
5. Creignou, N., Daudé, H., Egly, U., Rossignol, R.: (1,2)-QSAT: A Good Candidate for Understanding Phase Transitions Mechanisms. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 363–376. Springer, Heidelberg (2009)
6. Egly, U., Seidl, M., Tompits, H., Woltran, S., Zolda, M.: Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 214–228. Springer, Heidelberg (2004)
7. Egly, U., Seidl, M., Woltran, S.: A solver for QBFs in negation normal form. Constraints 14(1), 38–79 (2009)
8. Gent, I., Walsh, T.: Beyond NP: The QSAT Phase Transition. In: Proc. of AAAI/IAAI 1999, pp. 648–653 (1999)
9. Gent, I.P., Walsh, T.: The SAT Phase Transition. In: Proc. of ECAI 1994, pp. 105–109 (1994)
10. Goultiaeva, A., Iverson, V., Bacchus, F.: Beyond CNF: A Circuit-Based QBF Solver. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 412–426. Springer, Heidelberg (2009)
11. Klieber, W., Sapra, S., Gao, S., Clarke, E.: A Non-Prenex, Non-Clausal QBF Solver with Game-State Learning. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 128–142. Springer, Heidelberg (2010)
12. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: 2+p-sat: Relation of typical-case complexity to the nature of the phase transition. Random Struct. Algorithms 15(3-4), 414–435 (1999)
13. Navarro, J., Voronkov, A.: Generation of Hard Non-Clausal Random Satisfiability Problems. In: Proc. AAAI/IAAI 2005, pp. 436–442 (2005)
14. Prasad, M.R., Biere, A., Gupta, A.: A survey of recent advances in SAT-based formal verification. STTT 7(2), 156–173 (2005)

A Lesson on Structural Testing with PathCrawler-online.com*

Nikolai Kosmatov[1], Nicky Williams[1], Bernard Botella[1], Muriel Roger[1], and Omar Chebaro[2]

[1] CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] ASCOLA (EMN-INRIA, LINA), École des Mines de Nantes, 44307 Nantes France
firstname.lastname@mines-nantes.fr

Abstract. PathCrawler is a test generation tool developed at CEA LIST for structural testing of C programs. The new version of PathCrawler is developed in an entirely novel form: that of a test-case generation web service which is freely accessible at PathCrawler-online.com. This service allows many test-case generation sessions to be run in parallel in a completely robust and secure way. This tool demo and teaching experience paper presents PathCrawler-online.com in the form of a lesson on structural software testing, showing its benefits, limitations and illustrating the usage of the tool on simple examples.

1 Introduction

Structural testing assures that the test set has thoroughly exercised the program with respect to a given coverage criterion. PathCrawler [1, 2] is a concolic test generation tool enumerating all program paths developed at CEA LIST for structural testing of C programs. This paper presents a new version of the tool developed in a novel form: that of a test-case generation web service freely accessible online [3]. This form is ideal for discovering the tool, its evaluation and teaching. We have used it in courses taught in several French universities for groups of 30 students working in parallel.

In our opinion, the benefits of automatic structural testing remain underestimated in the industry. To improve the situation, structural testing tools must be more widely taught at verification and validation courses during higher education. That was our motivation to write a teaching experience paper and to demonstrate the PathCrawler tool in the form of a small lessson where the students manipulate the tool and answer the questions. Sec. 2,3 present our experience feedback and the lesson. Sec. 4 provides some related work and concludes.

2 Teaching Feedback and Discussion

This lesson assumes the students have learned basic notions related to structural testing e.g. control-flow graphs (CFG), execution paths, branches and oracle. Our

* This work has been partially funded by several ANR projects.

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, pp. 169–175, 2012.
© Springer-Verlag Berlin Heidelberg 2012

a)

```
1  int Bsearch(int *A, int x, int n){
2    int low=0, high=n-1, mid, ret=0;
3    while( high > low ){
4      mid = (low + high) / 2 ;
5      if( x == A[mid] )
6        ret = 1;
7      if( x > A[mid] )
8        low = mid + 1 ;
9      else
10       high = mid - 1;
11     }
12   mid = (low + high) / 2 ;
13   if( ret == 0 && x == A[mid] )
14     ret = 0;
15   return ret;
16 }
```

b)

```
1  int Bsearch(int *A, int x, int n){
2    int low=1, high=n-1, mid, ret=0;
3    while( high > low ){
4      mid = (low + high) / 2 ;
5      if( x == A[mid] )
6        ret = 1;
7      if( x > A[mid] )
8        low = mid + 1 ;
9      else
10       high = mid - 1;
11     }
12   mid = (low + high) / 2 ;
13   if( ret == 0 && x == A[mid] )
14     ret = 1;
15   return ret ;
16 }
```

Fig. 1. Two erroneous versions of binary search of element `x` in sorted array `A` of size `n`

experience shows that theoretical courses are insufficient for learning software testing for the majority of students. The selected questions of this lesson correspond exactly to the difficult points that should be thoroughly exercised in practice. Testing with a wrong (incomplete or too strong) precondition, or without a precondition is a very common error that may be revealed by runtime-errors or wrong test results, but may remain completely unnoticed. Another common difficulty is to understand the role of an oracle. Many students do not see how to check the results of a function under test f without necessarily using the same algorithm again. Almost all students check the return values and forget to check that f does not modify variables when it does not supposed to do so. An incorrect oracle may work perfectly in an exercise and remain unnoticed, so the teacher should check the oracle of each student even if the final results seem correct. The three final questions help the students to acquire a deeper understanding of the subtleties of structural testing. Drawing the CFG helps the students to visualize and analyze test generation results, especially in the last question involving several functions.

3 The Lesson

The C function `Bsearch` implements the well-known binary (or dichotomic) search algorithm. Given an ordered array of integers, `A`, an integer value to search for, `x`, and the number of elements in `A`, `n`, it should return 1 if `x` is an element of `A` and 0 if not. Let us investigate how PathCrawler can be used to test two different implementations of `Bsearch` of Fig. 1.

3.1 Testing without a Precondition (Test Parameters)

Question 1. We start with the first implementation shown in Fig. 1a and behave as an inexperienced tester might, by just uploading this source code into

```
1  void oracle_Bsearch(int *Pre_A, int *A,
2    int Pre_x, int x, int Pre_n, int n, int result_implementation){
3    int i, present = 0;
4    for(i=0;i<n;i++){
5      if(A[i] != Pre_A[i])
6        { pathcrawler_verdict_failure(); return; } /* A modified */
7      if(Pre_A[i] == Pre_x)
8        present = 1;
9    }
10   if(present==0 && present != result_implementation)
11     { pathcrawler_verdict_failure(); return; } /* x wrongly found in A */
12   else if(present==1 && present != result_implementation)
13     { pathcrawler_verdict_failure(); return; } /* x wrongly not found in A */
14   else {pathcrawler_verdict_success(); return; }
15 }
```

Fig. 2. Oracle for the functions of Fig. 1

PathCrawler-online.com via Test Your Code page and running test generation of function Bsearch with the default test parameters. Look at the Test Session Results and explain the errors.

Answer. The Test Session Summary shows that two test-cases provoked a run-time error. The Test-Case pages provide the input array sizes, input values and covered path of each test-case. In one of them, with n=1873679323, the segmentation fault occurred because although the array was empty in this test-case, the loop body was executed (the path contains +3, i.e. the true branch at line 3 of the source code), including an out-of-bound array access at line 5. The implementation assumes that n is no greater than the array dimension, so this must be true for each test-case. The other test-case, with a negative value of n, provoked a similar out-of-bound array access at line 13.

3.2 Definition of a Precondition

Question 2. Restart test generation of Bsearch again, but this time customize the test parameters so that the values of the elements of A and of x are restricted to the interval 0..10, the dimension of A, denoted by dim(A), can be any value from 1 to 8 and add an unquantified precondition stating that n is equal to dim(A). Look at the Test Session Results. Are there runtime errors? Complete the precondition if necessary. Explain what purpose a precondition serves in testing.

Answer. This time, the generated tests do not cause execution errors. However, we see that in these cases A is not always sorted so binary search does not necessarily work. We add to the precondition the requirement that A is sorted in the following quantified precondition:

$$\text{for all INDEX such that INDEX} < \text{n - 1}, \quad \text{A[INDEX]} <= \text{A[INDEX + 1]}.$$

The preconditions ensure that the automatically generated test-cases will all respect the input domain of the implemented function. This avoids test failures due to test-cases which provoke execution errors or give the wrong result because, even if correctly implemented, the algorithm is not supposed to work on the inputs of such test-cases.

3.3 Role of an Oracle in Testing

Question 3. What purpose does the oracle fulfill in testing? What should a complete oracle for `Bsearch` check?

Answer. The oracle examines the inputs and outputs of each test-case and decides whether the implementation has given the expected outputs for the given inputs. A complete oracle for `Bsearch` should check that

- the array `A` is not changed by the implementation,
- the implementation returns the correct result, i.e. if `x` is really present in `A` then the implementation returns 1 and if not, it returns 0.

3.4 Using Structural Testing to Detect a Bug

Question 4. Go back to the test parameters used in Question 2 and change the default oracle, calling `pathcrawler_failure()` if the outputs are as expected and `pathcrawler_success()` if not. Rerun generation and check the verdicts and paths covered by the different test-cases. How do these help locate the bug in this implementation? Correct the bug and re-run generation with the same test parameters.

Answer. We replace the default oracle by the function of Fig. 2. The new test session results contain 8 test-cases with verdict `failure` and 15 with verdict `success`. Looking at the paths of the test cases, we see that the second condition on line 13 is only satisfied by the test-cases which failed (their path contains `+13, +13b`). This indicates that the bug is in the single statement (line 14) which is executed if this condition is satisfied. We replace it by `ret=1;` and re-run generation. All test-cases have now verdict `success`.

3.5 Limits of Structural Testing

Question 5. With the same test parameters as in Question 4, now generate tests for the second implementation of `Bsearch` in Fig. 1b. Are the same number of cases generated as in Question 4? What are the verdicts? The bug is in line 2. Try generation a few times to check whether the verdicts are always the same. Explain your results.

Answer. Fewer cases are generated this time and they almost always all have verdict `success` (although a run may occasionally happen to generate a test with a `failure` verdict). The bug in this implementation just causes the first element of the array not to be checked in some cases and this is why fewer tests are generated. This example shows the limits of classical structural testing. All paths in the code may be covered without revealing the error. This sort of error can only be found by taking the intended functionality of the implementation into account when generating the tests.

```
/* copy the function Bsearch above */
int spec_Bsearch(int *Pre_A, int *A,
  int Pre_x, int x, int Pre_n, int n, int result_implementation){
  int i, present = 0;
  for(i=0;i<n;i++){
    if(A[i] != Pre_A[i])
      return 0; /* A modified */
    if(Pre_A[i] == Pre_x)
      present = 1;
  }
  if(present==0 && present != result_implementation)
    return 0; /* x wrongly found in A */
  else if(present==1 && present != result_implementation)
    return 0; /* x wrongly not found in A */
  else return 1;
}

int CompareBsearchSpec(int *A, int x, int n){
  int *Pre_A = (int *)malloc(n * sizeof(int));
  int i;
  for (i = 0; i < n; i++)
    Pre_A[i] = A[i];
  int ret=Bsearch(A,x,n);
  return spec_Bsearch(Pre_A, A, x, x, n, n, ret);
}
```

Fig. 3. Specification for `Bsearch` and function `CompareBsearchSpec` to be tested in Qu. 6

```
void oracle_CompareBsearchSpec(int *Pre_A, int *A,
  int Pre_x, int x, int Pre_n, int n, int result_compare){
  if (result_compare)
    { pathcrawler_verdict_success(); return; }
  else
    { pathcrawler_verdict_failure(); return; }
}
```

Fig. 4. Oracle for function `CompareBsearchSpec` of Fig. 3

3.6 Testing with a Specification

Question 6. Create a file with the function `Bsearch` of Fig. 1b and the functions shown in Fig. 3. Upload this new file into `PathCrawler-online.com` and generate tests for the function `CompareBsearchSpec` using the oracle of Fig. 4 and, otherwise, the same test parameters as in Question 5. Explain the significance of the test-case with a `failure` verdict (usually obtained before test generation is interrupted because of the limit on the number of partial paths in this evaluation version).

Answer. The function `spec_Bsearch` provides a specification similar to the oracle of Fig. 2, while `CompareBsearchSpec` calls `Bsearch` and `spec_Bsearch` to compare the result with the specification. There are therefore execution paths in `CompareBearchSpec` in which the result returned by the implementation is not accepted by `spec_Bsearch`. In trying to generate tests to cover these paths, PathCrawler is searching for inputs which cause the implementation of `Bsearch` to give an unexpected result. In the test-case with the failure verdict, `x` is the first element of `A` and this case reveals the bug that was not detected by structural testing of the implementation of `Bsearch` alone.

4 Related Work and Conclusion

The PathCrawler method is related to other test generation tools combining symbolic and concrete execution of the program under test, for example, DART [4], CUTE [5], EXE [6], PEX [7], YOGI [8]. Although the idea to make tools available online for evaluation and use is very attractive, most tools are available only for download and require installation on the user's platform.

In the domain of software verification, few research teams provide an online (evaluation) version for their tool to allow (potential) users to quickly run it and to familiarize themselves with its concepts. The AgitarOne tool [9] (for unit testing of Java) and Euclide [10] (for property verification or proving reachability in C code) did have online versions allowing to try the tools. PEX for Fun [11] gives access to a limited version of the PEX [7] test generation tool in a recreational way, inviting the user to try to solve little puzzles. The online version of the Interproc static analyzer [12] illustrates static analysis for programs in a small imperative programming language accepted by the tool. In constraint programming, WebCHR [13] provides a service for solving CHR (Constraint Handling Rules) constraints online. No other software verification tool provides a testing web service for C software similar to `PathCrawler-online.com`.

Automatic structural test generators may also be used for other purposes, for example, to find execution errors [4–6], to verify conformity to specifications [7, 8, 14] or to verify non-functional properties [15]. The PathCrawler method can be efficiently combined with static analysis techniques, for example, for program debugging [16, 17]. We are currently studying uses of PathCrawler which go beyond traditional structural test-case generation, as illustrated by Question 6 above, and its novel combinations with static analysis and proof tools.

In this paper, we demonstrated `PathCrawler-online.com`, the new online version of the PathCrawler test generation tool, and illustrated by a small practical session how it can be used for teaching. We hope that this work will be helpful in teaching software testing at university level and will contribute to the introduction of automatic structural testing techniques in industry.

References

1. 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)
2. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST 2009 (2009)
3. Kosmatov, N.: PathCrawler online (2010-2012), `http://pathcrawler-online.com/`
4. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI 2005 (2005)
5. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/FSE 2005 (2005)

6. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: CCS 2006 (2006)
7. 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)
8. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: ISSTA 2008 (2008)
9. AgitarOne Test Generator (2012), http://www.agitar.com/
10. Gotlieb, A.: Euclide: a constraint-based testing platform for critical C programs. In: ICST 2009 (2009), http://euclide.gforge.inria.fr/
11. Pex for fun: Online evaluation version of PEX (2011), http://pexforfun.com/
12. Interproc online (2012), http://pop-art.inrialpes.fr/interproc/interprocweb.cgi
13. WebCHR online (2012), http://dtai.cs.kuleuven.be/CHR/webchr.shtml
14. Rueher, M.: Exploration of the Capabilities of Constraint Programming for Software Verification. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 182–196. Springer, Heidelberg (2006)
15. Williams, N.: WCET measurement using modified path testing. In: WCET 2005 (2005)
16. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: The SANTE Tool: Value Analysis, Program Slicing and Test Generation for C Program Debugging. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 78–83. Springer, Heidelberg (2011)
17. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC 2012 (2012)

Tutorial on Automated Structural Testing with PathCrawler*

(Extended Abstract)

Nikolai Kosmatov and Nicky Williams

CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Introduction. Automation of test-case generation brings obvious benefits. In critical systems processes where structural testing is required by the development norm, manually creating tests from the specification fails to achieve complete satisfaction of the coverage criterion. In this case, automatic methods help to reach the objectives which are not covered and provide corresponding path conditions that may be used to refine the specification if needed. They may also determine whether the objectives which are not yet covered are really infeasible. Even when the development process does not impose any structural testing activity, the use of a structural test generation tool is a way to increase the quality of the software with a very low cost overhead. PathCrawler is a concolic test generation tool developed at CEA LIST for structural testing of C programs. It aims to cover all feasible program paths. The new version of PathCrawler is developed in an entirely novel form: that of a test-case generation web service which is freely accessible at `PathCrawler-online.com`.

The Tutorial. The first aim of the tutorial is to show how *C code can quickly and easily be debugged using automatic structural unit testing.* The second aim is to show that tools such as PathCrawler can help to *respect the code coverage required by many development norms,* and report on what cannot be covered. Finally, we will show how structural testing may be used in *combination with static analysis techniques* and enhance their results.

This tutorial is aimed mainly at software engineering professionals and students. They will learn more about the state of the art in automated software testing and how it could help them in their future career in software development or validation. Software engineering lecturers may also be interested in how a tool such as `PathCrawler-online.com` can help in teaching software testing. The necessary background is some knowledge of the C language.

After a brief introduction to structural testing and the concolic method underlying the PathCrawler tool, the notions of precondition, coverage criterion, and oracle will be explained and illustrated interactively on simple examples. The tool outputs (test cases, results, infeasible paths,...) will be explained and the tutorial students will be guided in the use of these outputs to discover the source of different bugs. Some limits of structural testing and advantages of its combination with static analysis tools will also be illustrated. The tutorial will use the online version of the PathCrawler tool: `PathCrawler-online.com`.

* This work has been partially funded by several ANR projects.

A.D. Brucker and J. Julliand (Eds.): TAP 2012, LNCS 7305, p. 176, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Author Index