Alexandre Petrenko
Adenilso Simão
José Carlos Maldonado (Eds.)

LNCS 6435

Testing Software and Systems

22nd IFIP WG 6.1 International Conference, ICTSS 2010
Natal, Brazil, November 2010
Proceedings

ifip

Lecture Notes in Computer Science 6435

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

Alexandre Petrenko Adenilso Simão
José Carlos Maldonado (Eds.)

Testing Software and Systems

22nd IFIP WG 6.1 International Conference, ICTSS 2010
Natal, Brazil, November 8-10, 2010
Proceedings

Springer

Volume Editors

Alexandre Petrenko
Centre de Recherche Informatique de Montréal (CRIM)
405 Ogilvy Avenue, Suite 101, Montréal, Québec H3N 1M3, Canada
E-mail: petrenko@crim.ca

Adenilso Simão
José Carlos Maldonado
University of São Paulo
Institute of Mathematics and Computer Sciences
Avenida Trabalhador São-carlense, 400, 13560-970 São Carlos, SP, Brazil
E-mail: {adenilso, jcmaldon}@icmc.usp.br

Library of Congress Control Number: 2010936701

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-16572-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-16572-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

Testing has steadily become more and more important within the development of software and systems, motivating an increasing amount of research aimed at trying to solve both new challenges imposed by the advancement in various areas of computer science and long-standing problems. Testing has evolved during the last decades from an ad-hoc and under-exposed area of systems development to an important and active research area.

The 22nd International Conference on Testing Software and Systems (ICTSS) involved the merger of two traditional and important events which have served the testing community as an important venue for discussing advancements in the area. Those events, namely, TestCom (the IFIP TC 6/WG 6.1 International Conference on Testing of Communicating Systems), and FATES (International Workshop on Formal Approaches to Testing of Software), together form a large event on testing, validation, and specification of software and systems. They have a long history. TestCom is an IFIP-sponsored series of international conferences, previously also called International Workshop on Protocol Test Systems (IWPTS) or International Workshop on Testing of Communicating Systems (IWTCS). It is devoted to testing of communicating systems, including testing of communication protocols, services, distributed platforms, and middleware. The previous events were held in Vancouver, Canada (1988); Berlin, Germany (1989); McLean, USA (1990); Leidschendam, The Netherlands (1991); Montreal, Canada (1992); Pau, France (1993); Tokyo, Japan (1994); Evry, France (1995); Darmstadt, Germany (1996); Cheju Island, South Korea (1997); Tomsk, Russia (1998); Budapest, Hungary (1999); Ottawa, Canada (2000); Berlin, Germany (2002); Sophia Antipolis, France (2003); Oxford, UK (2004); Montreal, Canada (2005); and New York, USA (2006). Fates — Formal Approaches to Testing of Software — is a series of workshops devoted to the use of formal methods in software testing. Previous events were held in Aalborg, Denmark (2001); Brno, Czech Republic (2002); Montreal, Canada (2003); Linz, Austria (2004); Edinburgh, UK (2005); and Seattle, USA (2006). From 2007 on, TestCom and Fates have been jointly held in Tallinn, Estonia (2007), Tokyo, Japan (2008) and Eindhoven, The Netherlands (2009).

The objective of ICTSS 2010 was to be a forum for researchers from academia as well as industry, developers, and testers to present, discuss, and learn about new approaches, theories, methods and tools in the field of testing software and systems. This volume contains the proceedings of ICTSS 2010. Out of 60 submitted papers the Program Committee selected 16 papers for publication as full papers and presentation at the conference. Together with the two invited presentations by Ina Schieferdecker of Fraunhofer Institut, Berlin, Germany and by Connie Heitmeyer of Naval Research Laboratory, Washington, DC, USA, they form the contents of these proceedings. The conference itself, in addition,

contained presentations of short papers, which were separately published as a technical report by CRIM.

We would like to thank the numerous people who contributed to the success of ICTSS 2010: the Steering Committee, the Program Committee and the additional reviewers for their support in selecting papers and composing the conference program, and the authors and the invited speakers for their contributions without which, of course, these proceedings would not exist. We thank CRIM, Conformiq and Microsoft Research for their financial support, and Springer for its support in producing these proceedings. We acknowledge the use of EasyChair for conference management and wish to thank its developers. Last, but not least, we thank the Department of Informatics and Applied Mathematics of Federal University of Rio Grande do Norte, in particular, Marcel Oliveira, Thais Batista and David Deharbe, for all matters regarding the local organization and for making ICTSS 2010 run smoothly.

August 2010

Alexandre Petrenko
Adenilso Simão
José Carlos Maldonado

Conference Organization

Program Chairs

Alexandre Petrenko	CRIM, Canada
Adenilso Simão	University of São Paulo, Brazil
José Carlos Maldonado	University of São Paulo, Brazil

Steering Committee

Paul Baker	Motorola, UK
Ana R. Cavalli	Telecom SudParis, France
John Derrick	University of Sheffield, UK, (Chair)
Wolfgang Grieskamp	Microsoft Research, USA
Roland Groz	Grenoble Institute of Technology, France
Toru Hasegawa	KDDI RandD Labs., Japan
Manuel Nunez	University Complutense de Madrid, Spain
Alexandre Petrenko	CRIM, Canada
Jan Tretmans	Embedded Systems Institute, The Netherlands
Andreas Ulrich	Siemens AG, Germany
Margus Veanes	Microsoft Research, USA

Program Committee

Paul Baker	Motorola, UK
Antonia Bertolino	ISTI-CNR, Italy
Roberto S. Bigonha	Federal University of Minas Gerais, Brazil
Gregor von Bochmann	University of Ottawa, Canada
Ana R. Cavalli	Telecom SudParis, France
John Derrick	University of Sheffield, UK
Sarolta Dibuz	Ericsson, Hungary
Khaled El-Fakih	American University of Sharjah, UAE
Gordon Fraser	Saarland University, Germany
Wolfgang Grieskamp	Microsoft Research, USA
Roland Groz	Grenoble Institute of Technology, France
Toru Hasegawa	KDDI R&D Labs, Japan
Klaus Havelund	Jet Propulsion Laboratory, USA
Rob Hierons	Brunel University, UK
Teruo Higashino	Osaka University, Japan

Dieter Hogrefe	University of Gottingen, Germany
Antti Huima	Conformiq Inc., USA
Thierry Jeron	IRISA Rennes, France
Ferhat Khendek	Concordia University, Canada
Myungchul Kim	ICU, Korea
Hartmut Konig	BTU Cottbus, Germany
Victor V. Kuliamin	ISP RAS, Russia
David Lee	Ohio State University, USA
Bruno Legeard	Smartesting, France
Patricia Machado	Federal University of Campina Grande, Brazil
Giulio Maggiore	Telecom Italia Mobile, Italy
José Carlos Maldonado	University of São Paulo, Brazil
Eliane Martins	University of Campinas, Brazil
Ana Cristina de Melo	University of São Paulo, Brazil
Brian Nielsen	University of Aalborg, Denmark
Daltro Jose Nunes	Federal University of Rio Grande do Sul, Brazil
Doron Peled	University of Bar-Ilan, Israel
Alexandre Petrenko	CRIM, Canada
S. Ramesh	General Motors India Science Lab, India
Augusto Sampaio	Federal University of Pernambuco, Brazil
Ina Schieferdecker	Fraunhofer FOKUS, Germany
Adenilso Simao	University of Sao Paulo, Brazil
Kenji Suzuki	University of Electro-Communications, Japan
Jan Tretmans	Embedded Systems Institute, The Netherlands
Andreas Ulrich	Siemens AG, Germany
Hasan Ural	University of Ottawa, Canada
M. Umit Uyar	City University of New York, USA
Margus Veanes	Microsoft Research, USA
Cesar Viho	IRISA Rennes, France
Carsten Weise	RWTH Aachen, Germany
Burkhart Wolff	University of Paris-Sud, France
Nina Yevtushenko	Tomsk State University, Russia
Xia Yin	Tsinghua University, China

Local Organization

Thais Batista	Federal University of Rio Grande do Norte, Brazil
David Deharbe	Federal University of Rio Grande do Norte, Brazil
Diego Oliveira	Federal University of Rio Grande do Norte, Brazil

External Reviewers

Fides Aarts
Bruno Abreu
Omar Alfandi
Erika Almeida
Wilkerson Andrade
Jongmoon Baik
Fayal Bessayah
Karine Birnfeld
Florent Bouchy
Marcelo d'Amorim
Arnaud Dury
Mazen El Maarabani
Jose Escobedo
Fabiano Cutigi Ferrari
Eduardo Figueiredo
Dominik Franke
Maxim Gromov
Yating Hsu
Iksoon Hwang
Guy-Vincent Jourdan
Prabhu Shankar Kaliappan
Sungwon Kang
Natalia Kushik
Martin Leucker
Ralf Mitsching
Swarup Mohalik
Sidney Nogueira
Ulrik Nyman
Petur Olsen
Paulo Salem da Silva
Gizelle Sandrini de Lemos
Rodolfo Srgio Ferreira de Resende
Abraham L.R. de Sousa
Ivan Tierno
Hirozumi Yamaguchi

Table of Contents

Test Automation with TTCN-3 - State of the Art and a Future Perspective (Invited Talk) 1
Ina Schieferdecker

A Model-Based Approach to Testing Software for Critical Behavior and Properties (Abstract of Invited Talk) 15
Constance Heitmeyer

A Pareto Ant Colony Algorithm Applied to the Class Integration and Test Order Problem 16
Rafael da Veiga Cabral, Aurora Pozo, and Silvia Regina Vergilio

More Testable Properties 30
Yliès Falcone, Jean-Claude Fernandez, Thierry Jéron, Hervé Marchand, and Laurent Mounier

Alternating Simulation and IOCO 47
Margus Veanes and Nikolaj Bjørner

Reducing the Cost of Model-Based Testing through Test Case Diversity 63
Hadi Hemmati, Andrea Arcuri, and Lionel Briand

Built-in Data-Flow Integration Testing in Large-Scale Component-Based Systems 79
Éric Piel, Alberto Gonzalez-Sanchez, and Hans-Gerhard Gross

Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing 95
Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand

Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage 111
Beatriz Pérez Lamancha and Macario Polo Usaola

Increasing Functional Coverage by Inductive Testing: A Case Study 126
Neil Walkinshaw, Kirill Bogdanov, John Derrick, and Javier Paris

FloPSy – Search-Based Floating Point Constraint Solving for Symbolic Execution 142
Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux

Test Data Generation for Programs with Quantified First-Order Logic Specifications 158
Christoph D. Gladisch

Efficient Distributed Test Architectures for Large-Scale Systems 174
Eduardo Cunha de Almeida, João Eugenio Marynowski, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez

Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction 188
Fides Aarts, Bengt Jonsson, and Johan Uijen

Practical End-to-End Performance Testing Tool for High Speed 3G-Based Networks 205
Hiroyuki Shinbo, Atsushi Tagami, Shigehiro Ano, Toru Hasegawa, and Kenji Suzuki

A Learning-Based Approach to Unit Testing of Numerical Software 221
Karl Meinke and Fei Niu

From Scenarios to Test Implementations via Promela 236
Andreas Ulrich, El-Hachemi Alikacem, Hesham H. Hallal, and Sergiy Boroday

Vidock: A Tool for Impact Analysis of Aspect Weaving on Test Cases 250
Romain Delamare, Freddy Munoz, Benoit Baudry, and Yves Le Traon

Author Index 267

Test Automation with TTCN-3 - State of the Art and a Future Perspective

Ina Schieferdecker

TU Berlin/Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
ina.schieferdecker@fokus.fraunhofer.de

Abstract. Test automation encompasses all activities to automate various steps in the overall testing process including automation of test management, test generation, or test execution. The standardized Testing and Test Control Notation (TTCN-3) addresses selected challenges by defining a test specification language and a test system architecture that enables the implementation and execution of TTCN-3 test suites. Over the years, the standard has continuously being maintained and evolved. For example, concepts for static test configurations or for advanced parameterization and typing have been defined. The paper reviews the history and current status of TTCN-3 and concludes by giving an overview on recent extensions of TTCN-3 and future plans.

Keywords: test automation, test specification, test framework, test execution, test management, TTCN-3.

1 Introduction

With the Testing and Test Control Notation (TTCN-3) the testing community obtained a generic technology, standardized by ETSI and ITU, which allows the development and design of systematic tests and their reuse independently from technologies, product lines and manufacturers. TTCN-3 is a living, widely established and continuously maintained testing technology, available in a version 4.2.1 from July 2010.

However, the access to this testing technology is not always easy; here, the prejudices of the testing automation are pairing with the acceptance threshold of a new technology. Thereby, research done by Motorola shows that the application of this technology is not only increasing the quality of tests, but also the efficiency of its development and the grade of its reutilization [1]. Moreover, studies in China [2] show that this technology is immediately applicable by the testing personnel and can be actively used by people being familiar with the needs, methods and solutions of testing already after a 1-2 day instruction.

TTCN-3 is a powerful testing technology – powerful enough to solve a big variety of testing problems – and with it at first glance also rather complex. However, please allow me to relate a mind game from a discussion with Mark Harman from Brunel University at a Dagstuhl seminar in 2009: Please imagine that the total loss of your

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 1–14, 2010.

system or the corresponding testing system is imminent. You will only be able to reduce the damage if you will decide or to save the system or the testing system. Whereas some years ago in its majority the system would have been saved (because this is the goal), nowadays the voices demanding the salvation of the testing system are increasing, since it is not only preserving the knowledge on the system to be developed, but also the knowledge about its safeguarding. The testing system allows us to reconstruct the system at any time and simultaneously to save the system in its evolution.

This more-of-knowledge is corresponding with the efforts of developing of every system: there is an increasing number of products where the same – if not more – efforts are being invested into the testing system as into the system itself. This shifting of efforts towards the testing system has been observed e.g. with Microsoft during the development of Windows XP, which was beneficial for all of us in terms of the stability of the operating system. At the same time, this additional investment is also reflected in technologies used for the development oft the testing systems. Who is not familiar with the huddle of testing tools, where every tool is already complex, but only the combination allows an adequate analysis and validation of a system. The main purpose of TTCN-3 is to thin out this huddle of tools – if not to abolish it. Thus, it is not amazing that this testing technology has been designed as powerful as it needs to be to consolidate the testing concepts, in the result achieving a simplification. And there is no end to be seen: continuously we are being asked to integrate additional concepts into TTCN-3. But we will recur to this later. In the following, the paper gives outline of the history, state of the art and possible future of TTCN-3.

2 TTCN-3 Yesterday: A Short Retrospection on the TTCN-3 History

TTCN-3 was worked out by ETSI as a new edition of the TTCN (Tree and Tabular Combined Notation), mainly developed for conformance, interoperability and performance tests of communication-based systems including protocols, services, interfaces, etc. TTCN-3 is a modern language for test specification and implementation, by means of which tests can be developed and specified textually or graphically, implemented and executed automatically.

TTCN-3 has first been presented in public in September 2000, and with the version v2.2.1 gained the necessary stability and maturity for tool development and its implementation in industry. The following versions have been amplified by new concepts, but designed with a backwards compatibility with former versions. The new versions for example included importable types defined in IDL or XML, dynamic templates and a logging-interface. In version v3.3.1, the addressing within the testing system has been improved. In version v3.3.2, a template-restriction was made possible, and in v3.4.1 user-defined attributes for testing judgment (verdicts) have been introduced. Recently for v4.1.1, a package concept has been created in order to allow the definition of specific concepts (e.g. for real-time) which are needed in dedicated domains or for selected applications of TTCN-3, but which would overload the

language if being defined for the core language. These packages are made optional and can be chosen depending on specific requirements of a given test solution. All the changes to TTCN-3 have been made transparent for everybody and can be tracked since 2005 by the change request (CR) procedure for TTCN-3.

The parts that currently constitute TTCN-3 are shown in Fig. 1. A user typically works with TTCN-3 via the textual format (the core language), the graphical format (based on Message Sequence Charts) or the tabular format (resembling the old style TTCN tables), although the textual format is used in the majority of cases. The operational semantics and the execution interfaces are most often relevant for tool vendors, but help users also to understand TTCN-3 and to apply it precisely. Language mappings for ASN.1, IDL and XML schemata help to use TTCN-3 for systems under tests using these languages for their data, interfaces, services, or protocols. Specific language concepts (most often extending not only the core language and its operational semantics, but also the execution interfaces) are defined in extension packages for optional use in specific usage contexts or for specific test applications or target systems under tests.

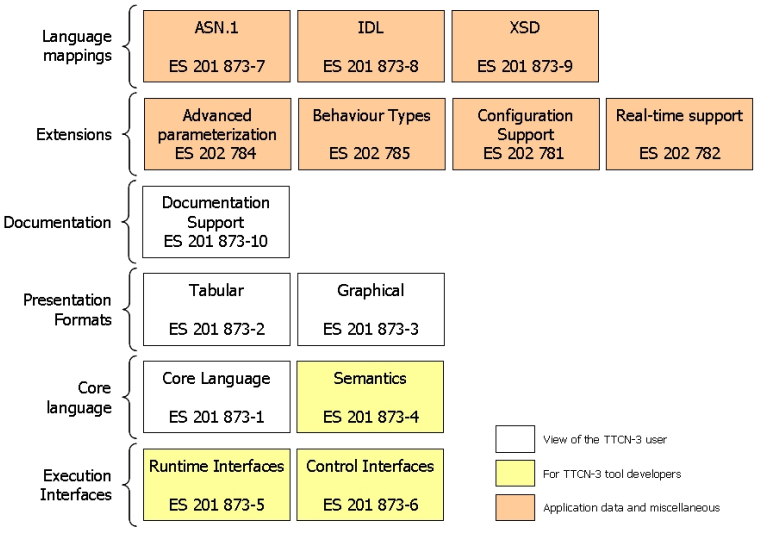


Fig. 1. The TTCN-3 Test Specification Language

In contrast to numerous testing and modeling languages, TTCN-3 does not only comprise a language for the specification of tests, but also a test system architecture and runtime and execution interfaces for TTCN-3 based testing systems (see Fig. 2). In the test system architecture, within the test engine (TE) the tests specified in TTCN-3 are executed – the TE is the runtime environment for the compiled TTCN-3 code of a set of TTCN-3 modules defining a test suite. The TE is wrapped by a number of adaptors that mediate between the TE and the system under test (SUT), the test

system, the test management, the logging, the encoding and decoding, and last but not least the (potentially distributed) handling of test components. The functionality of these adaptor components are defined via a number of operations specified in the Interface Definition Language (IDL) and mapped to programming languages for the test devices, i.e. to C. C++, C#, and Java.

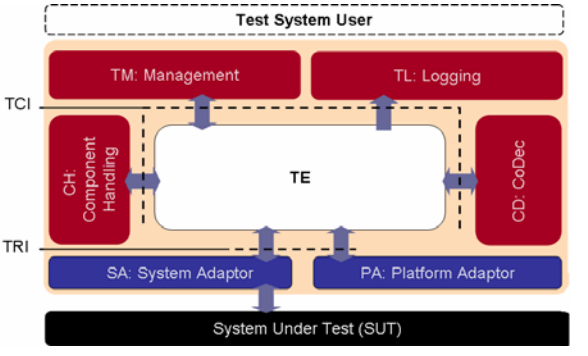


Fig. 2. The TTCN-3 Test Implementation Architecture

In the following, a small example for TTCN-3 is shown (see also Fig. 3) in order to give an insight into this technology: in a well known testing example of G.J. Meyers [3] for the classification of triangles (a triangle is defined by its side lengths und has to be classified), the question of typing is always posed. In TTCN-3, you are explicitly defining it:

```
type integer Triangle[3] (0..infinity);
// whole numbers to characterize a triangle
type enumerated Classification {
// properties of a triangle
   syntactICallyIncorrect,
   noTriangle,
   scaleneTriangle,
   isoscelesTriangle,
   equilateralTriangle
}
type port DetermineTriangle message
 { out Triangle; in Classification }
// Interface to SUT
type component TriangleTester
 { port DetermineTriangle b }
// Test Component
testcase Simple() runs on TriangleTester {
// a simple test case
    b.send(Triangle: {2,2,2});
    // the triangle to be checked
    b.receive(Classification: equilateralTriangle);
    // the expected response
    setverdict(pass); // a successful test
}
```

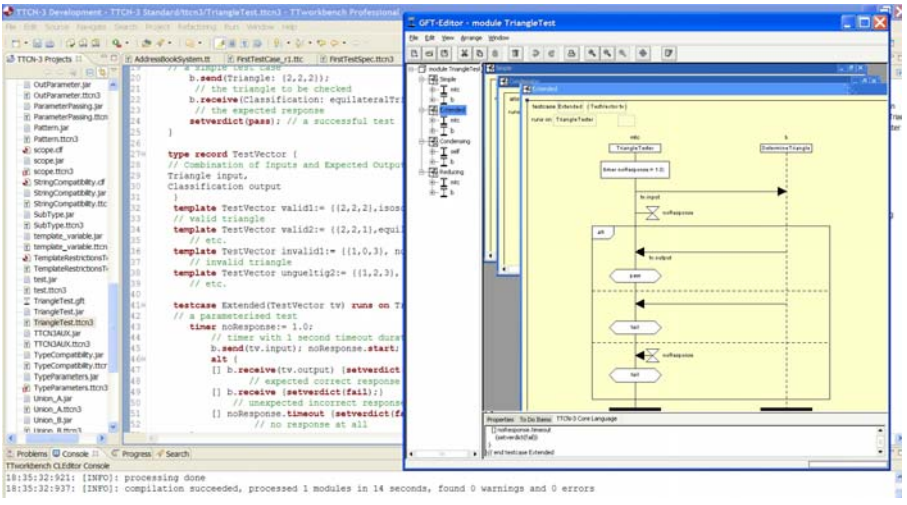


Fig. 3. A TTCN-3 Integrated Development Environment based on Eclipse

This test specification is wiring the testing data firmly with the testing activity, which is not very easy to maintain. Another method would be to separate the test data from the test behavior:

```
type record TestVector {
// combination of inputs and expected outputs
   Triangle input,
   Classification output
}
template TestVector valid1:= {{2,2,2}, isoscelesTriangle};
// valid triangle
template TestVector valid2:= {{2,2,1}, equilateralTriangle};
// etc.
template TestVector invalid1:= {{1,0,3}, noTriangle };
// invalid triangle
template TestVector invalid2:= {{1,2,3}, noTriangle };
// etc.

testcase Extended(TestVector tv) runs on TriangleTester {
// a parameterised test
   timer noResponse:= 1.0;
   // timer with 1 second timeout duration
   b.send(tv.input); noResponse.start;
   alt {
      []            b.receive(tv.output)            {setverdict(pass);}
         // expected correct response
      [] b.receive {setverdict(fail);}
         // unexpected incorrect response
      [] noResponse.timeout {setverdict(fail);}
         // no response at all
    }
}
```

In the extended test case given above, we are additionally taking into account that the SUT could also be failing or not answer at all. This could be followed up by you by e.g. starting queries with two parallel testing components to the SUT or by fixing two testing sequences, which first are checking all the valid, and afterwards, in dependence of positive testing results, all the non-valid triangles. Additional variations are possible, but cannot be demonstrated in this paper.

3 TTCN-3 Today: 10 Good Reasons to Use TTCN-3

This is leading us to a description of today's situation with TTCN-3, summarized in several "good reasons" for TTCN-3.

1. One is able to learn and apply TTCN-3 directly: TTCN-3 is a testing specific language, developed from testing persons for testing persons. A couple of key concepts, which one needs typically in the majority of the test cases like test cases, test verdicts or test data (called templates), allow one to design and specify tests directly and easily. But this is not the only advantage of TTCN-3: a great number of additional concepts allow one to develop also test for heterogeneous interfaces, distributed tests, dynamic tests etc. If this is not enough, one has the possibility to integrate further concepts into the language or to directly contribute to the TTCN-3 standards. I would like to draw your attention to the large number of books, magazines or articles dealing with this technology, its application and the respective tools (see also the TTCN-3 bibliography given in Fig. 4).
2. One is able to verify ones testing knowledge and get a certificate with successful attendance. On the basis of the certified Tester Schema of the International Software Testing Qualification Boards (ISTQB), the German Testing Board (GTB) has developed a TTCN-3 training scheme (see Fig. 4). Certified training providers are offering TTCN-3 courses training in the applications of TTCN-3. An inclusion of the TTCN-3 certificate into the Certified Tester Expert Level with ISTQB is in preparation.
3. One is able to contribute to the development of TTCN-3 by oneself. At ETSI, the Change Request (CR) process for TTCN-3 is administered and corrections, amendments and proposals for improvements can be placed (see Fig. 4). Whenever you will see the need of further development in your practical applications, you will be able to write this as CR and track the path to the solution in a transparent procedure. CRs can lead to corrections, clarifications, extensions or amendments. They can also be appointed as copies of an already existing CR or be closed well-founded without changes on the standard.
4. One is able to use a wide range of TTCN-3 tools. As TTCN-3 is a standard, it allows the development of off-the-shelf tools. Hence, it is possible to detach the extensive care and development of proprietary in-house tools by license and maintenance contracts and to concentrate on the design and specification of test cases. The development and maintenance of tool components for the test development, application, reporting or integration will be carried out by a third party. The availability of a large number of TTCN-3 tools requires no dead-end-decisions for one

provider – the investment in TTCN-3 remains secure independently from a concrete tool. The future spread of TTCN-3 is linked to a growing number of tool providers, like e.g. IBM, Elvior or Testing Technologies. A reference to commercial, free and open source tools is given in Fig. 4.

5. At present, TTCN-3 is used in a major extent. It is applied in the areas of e.g. telecommunication, Internet technologies, for control devices in automotive, avionics, transport and medical technologies, software and Web services of business applications and in eGovernment etc. Different bodies – not only ETSI or ITU – are voting for TTCN-3: 3GPP is using it for UMTS and LTE tests for wireless interfaces and backbones. Also, the WIMAX Forum for certification of hardware, the AUTOSAR consortium for basic software the automotive software middleware and the OMA for functional tests of enablers and mobile services are providing TTCN-3 tests. Thus, one is often be able to resort to existing libraries and TTCN-3 test suites and does not have to develop them right from the start. Further applications are emerging in the context of XÖV (XML-based exchange of data in the public administration), HL7 (communication stacks in integrated medical environments [4]) or MOST (entertainment application and components in the automobile).

6. One has the possibility to become a member of the permanently growing TTCN-3 community, meeting regularly at the annual TTCN-3 User Conference (T3UC, see Fig. 4) in order to share experience on applications, case studies, experiments, tools and new developments of TTCN-3. The 7th T3UC was taking place 2010 in Beijing. At T3UC, one has the chance to meet users, experts and tool manufacturers and to obtain new information about recent developments in the TTCN-3 tool and service area. Moreover, tutorials are held on a wide number of conferences such as SOFTEC, IQNITE, EuroStar or CONQUEST.

7. And most important: with TTCN-3 one has the possibility to tackle the testing tasks in an effective and efficient way. Thereby, TTCN-3 does not require a direct and absolute change of the current testing methods and environments, but allows a step by step migration to the new testing technology. One is able to combine existing solutions with new TTCN-3 testing solutions and thus use the established testing basis until one does not need it any more, when e.g. the tested systems are phased out.

8. Besides different programming languages like C, C++, Java or C#, which are supported by TTCN-3 tools, one gets a wide range of accesses to interface-technologies of the systems to test. This includes accesses for ASN.1, IDL and XML based systems – interfaces and formats which are elements of the TTCN-3 standard series. Additionally, tool manufacturers have been developing accesses for Java, C/C++, Tcl, Python, WSDL, BPEL, etc., which – although not (yet) standardized – are at disposition.

9. One is able to adequately comment the test cases and test suites with TTCN-3 by means of documentation annotations (documentation tags) and thus enhance the legibility, reutilization and maintainability of the tests. Additionally, one has the

possibility to use metric, samples, anti-samples and refactoring for TTCN-3, analyzing and optimizing the test suites.

10. TTCN-3 can be used for different kinds of testing and phases. The strengths of TTCN-3 are lying in the specification and application of tests, using interfaces or other communication means as test-accesses to the SUT. In some cases one has to deal with the heterogeneity of the system or part of the system with regards to interfaces, the used programming languages, the combination of SW and HW components, etc. It is exceptionally useful for integration, system and acceptance testing. Moreover, TTCN-3 can be used not only with functional testing, but also for non-functional testing like e.g. for performance tests (see e.g. [6]). In case one exploits the reuse potential of TTCN-3 test cases in different testing phases and types, one will not only win by a wide use of the testing technology, but also by reducing the input for instructions and training of the employees and the care of testing tools. At the same time, existing TTCN-3 test suites or libraries can be used more often.

Home Page:	www.ttcn-3.org
Quick Reference Guide:	http://www.blukaktus.com/
Bibliography:	www.ttcn-3.de
Mailing List:	ttcn3@list.etsi.org
Change Requests:	http://www.ttcn-3.org/ChangeRequest.htm
TTCN-3 Certificate:	http://www.german-testing-board.info/de/ttcn3_certificate.shtm
TTCN-3 User Conferences:	www.ttcn-3.org → Events
Tools:	www.ttcn-3.org → Tools
Tutorials:	www.ttcn-3.org → Courses and Tutorials

Fig. 4. TTCN-3 Sources

4 The Current TTCN-3 Version: TTCN-3 v4.2.1

In 2009, version 4.1.1 was prepared and completed in version 4.2.1 summer 2010. Here, the focus is on concepts for a more efficient application of testing configurations, on a better assistance of libraries and a further dynamization of testing specifications, e.g. by type-parameterization. Furthermore, concepts for the testing of real-time systems and for performance testing have been considered. In fact, such tests can already be formulated and applied by former TTCN-3 versions, but the objectives were to achieve a more efficient application of TTCN-3 by a more direct support of decided concepts, like those used with automotive engineering, automation, aeronautical and medical engineering.

Let us first however shortly review version v3.1.1 as several major extensions have been introduced. The following is not an exhaustive list of extensions but rather name the major ones only. The first group of extensions dealt with test components: test components can be **alive test components** meaning that several test behaviors can be executed on such a test component without termination of test component after termination of the test behavior. Hence, the test component state, i.e. the values of its port queues, local timers and variables, can be passed and reused between the tests behaviors started on an alive component. Furthermore, **inheritance for test component**

types has been introduced, so that test component types can extend other test component types – for example, to share a common set of ports.

The second group of extensions addressed the communication means. So far, unicast communication could be used only. In addition, **multicast communication**, i.e. communicating to or from a set of test components, and **broadcast communication**, i.e. communicating to or from all test components connected to the communicating test component, have been defined.

The third group of new concepts provided means for handling templates, i.e. test data, in TTCN-3 dynamically. So far, templates could be predefined in the module scope only. Parameters allowed to pass values into templates, however, neither the parameterization of templates with templates was not possible nor the construction of templates at runtime – in particular the dynamic combination of matching mechanisms according to the responses from the SUT. Therefore, v3.1.1 added the concepts of **template variables**, **template parameters**, **template returning functions**, and **local templates**, so that templates became as flexible as values.

A fourth group of extension added **nested type definitions**: so far, every type used within a structured type – for example the type of a record field – had to be explicitly defined before being referenced in the structured type definition. Now, type definitions can be made at the place where they are used – without giving them an explicit type name. This eases the mapping of external type systems like e.g. XML to TTCN-3.

Furthermore, the **logging** has been extended. The log statement has been enabled to log the "status" of any object in TTCN-3, i.e. of values and expressions, test components, timers, ports, and defaults. Test components can be named in the create operation, which eases the reading of test executions logs. Furthermore, the TCI was extended with an explicit logging interface – the TTCN-3 logging interface TLI [10] via which a test execution can be traced in various details – providing the various events during test execution with its timing and additional information.

Last but not least, **documentation support** has been added as a new part [14]: documentation comments with predefined documentation tags such as "@author", "@remark", "@desc", etc. can be used to document TTCN-3 modules on the basis of which documentation artifacts like html-pages can be generated. An example documentation for the triangle tests presented above is given in Fig. 5.

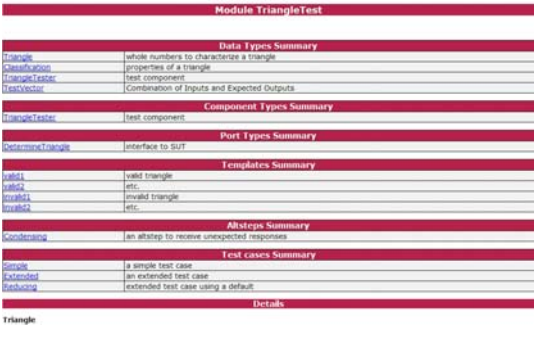


Fig. 5. Selected Documentation Elements for the Triangle Tests

Major extensions in version 4.1.1 and 4.2.1 have been the definition of **visibility** rules to control the reuse of definitions, i.e. public, private and friend, and the **import of import** statements. However, above all, **TTCN-3 packages** have been defined. The intention of TTCN-3 packages is to support additional concepts required by specific TTCN-3 targets such as application domains, without making them mandatory for all tool environments. Rather, these packages are optional and can be freely combined with the core language depending on specific requirements of a given application context. For example, a tool environment for automotive electronic control units may require the real-time package of TTCN-3, but not the behavior types package.

The Advanced Parameterization Package [15] defines static value parameterization of types, static type parameterization of types, templates, functions, altsteps and testcases, and default types for type parameters like default values/templates for value parameters. This allows e.g. to define message types with fields of "open" type or functions performing actions on data of "open" types as shown below:

```
type record Data <in type p_PayloadType>
{ Header hdr, p_PayloadType payload}
// a data record with header and payload field
// the hdr field is statically typed
// the payload field receives the type via the type parameter


type record of p_myType MyList <in type p_myType>;
// a list of p_MyType elements

function f_addElem <in type p_myType >
// adding an element to a list
( in MyList<p_myType> p_list, in p_myType p_elem)
 return MyList<p_myType>
{ p_list[lengthof(p_list)]:= p_elem; return p_list; }

f_addElem <integer> ({1,2,3,4}, 5);
// returning {1,2,3,4,5}
```

The Behaviour Types Package [16] allows to define types of functions, altsteps and testcases (FAT) as "prototypes" and to have module parameters, constants, variables and templates of FAT types and to store references to "real" FATs in them. References to FATs can be stored, passed as parameters and sent to other components. FATs can be called via their references.

```
type function MyFuncType ( in integer p1 ) return integer;
// definition of a function type
// with an integer parameter and an integer return
function f_myFunc ( in integer p_int ) return integer
{ return 2*p_int };
// definition of a concrete function compatible to MyFuncType
:
var MyFuncType v_func;
// definition of a variable of function type
v_func := f_myFunc;
// assignment of function f_myFunc to v_func
:
var integer x:= apply(v_func(10));
// execution of the function assigned to v_func, returning 20
```

The Configuration and Deployment Support Package [17] allows to have predefined static test components/configurations which exist across test cases. These "static" test configurations can be created and destroyed in the control part and used for several test cases sequentially. The test cases may add "dynamic" test components and connections to the test configuration but cannot destroy parts of the static test configuration.

```
configuration f_StaticConfig()
// the static test configuration f_StaticConfig
runs on MyMtcType
// having a main test component of MyMtcType
system MySystemType
// testing system under tests of MySystemType
{
     myStaticPTC:= MyPTCType.create static;
     // creation of the static parallel test component myStaticPTC
     map (myStaticPTC:PCO,system:PCO) static;
     // mapping the PTC port to the system port
}
:
testcase tc_test1 () execute on f_StaticConfig
// test case tc_test1 executing on f_StaticConfig
// having main test component, test system interface and parallel
// test component as defined above
{
     :
     myDynamicPTC:= MyPTCType.create;
     // creation of an additional parallel test component myDynamicPTC
     // which ceases to exist when the test case terminates
    :
}
testcase tc_test2 () execute on f_StaticConfig {…}
// test case tc_test1 executing on f_StaticConfig as well

control {
     var configuration myStaticConfig;
     // definition of the configuration variable myStaticConfig
     myStaticConfig := f_StaticConfig();     // configuration setup
     // setup of the static configuration
     execute(tc_test1());
     // execution of tc_test1 on the static configuration
     // the status of the static main and parallel test component
     // and of the statically mapped ports are kept for
     // test case tc_test2
     execute(tc_test2());
     // execution of tc_test2 on the static configuration
     // having the status when tc_test1 was completed
}
```

Last but not least, the Real-Time and Performance Testing Package [18] introduces system time progress (since the start of a test case), delays and timestamps. It allows specifying the required precision of time in the test system, to get the actual time, to suspend the execution of a test component until a given point in time, to specify ports with real-time requirements at which entering messages in the test system interface can be time stamped. The time stamps can be accessed and further used in the TTCN-3 specification.

```
module MyModule {
    :
    type port MyPortType message realtime
      // the message-based real time port type MyPortType
      // the received messages of ports of that type are time stamped
      {...}
      :
      port MyPortType myPort;
      // the message-based real time port myPort
      :
    var float v_specifiedSendTime:=1.0, v_sendTime, v_receiveTime;
      // time variables to store time stamps
    :
    wait (v_specifiedSendTime);
      // wait one second
    myPort.send(m_out);
      // send message m_out immediately
    v_sendTime:= now;
      // assign the current time to v_sendTime
      if (v_sendTime - v_specifiedSendTime > 0.01)
      // too late sending by the test system
      { … } // react accordingly
      :
    myPort.receive(t)-> timestamp v_receiveTime;
      // assign receive time to v_receiveTime
      if (v_receiveTime - v_sendTime > 10.0)
      // too late response from the system under test
      { … } // react accordingly
      :
} with {stepsize "0.001"};
// defines the time precision of that module to be millisecond
```

This paper allows providing impressions of the recent extensions to TTCN-3 only. The interested reader is asked to refer to the packages their selves for further details.

5 TTCN-3 Tomorrow: An Outlook

TTCN-3 has progressed a lot in the past. It is successfully used across various domains. A plethora of tools is available that support TTCN-3 in different scale. Further test experts are looking into the adoption of TTCN-3 in additional domains such as scientific computing or cloud computing. Still, more needs to be done. In particular people need to be trained in order to enable an efficient use and adoption of this test technology. The established TTCN-3 Certificate provides a good basis – however more reading and training material and training courses should be provided. Also, TTCN-3 is rarely lectured at universities – although free tools help in this. In particular, it is not enough to spread the knowledge about TTCN-3 and its success stories, but also a thorough methodology including guidelines and best practices for the application of TTCN-3 should be provided.

A strong basis will be the further maintenance and evolution of TTCN-3. Although already a quite exhaustive and powerful set of testing concepts is supported by TTCN-3, still more requirements appear – for example to have native support for object-oriented data types. However, like in the past every new concept is very critically reviewed and discussed before being added to the core language or to one – if

not to a new – extension package of TTCN-3 in order to keep TTCN-3 handy for the users and tool vendors.

Thinking further, our focus is on establishing TTCN-3 as a middleware for automated tests, like we know it with SQL for databases or with IDL for object-oriented systems. Even if software-based systems differ in its area of application and in its tests, a common number of key concepts for the development of testing solutions can be identified. On this common number of key concepts, implemented in TTCN-3, a corresponding test middleware, dedicated methods and tools for the different application areas can be efficiently and effectively developed. In my view, testing should more and more be seen as an engineering discipline, using common automated methods with educated personnel. The times of manual and proprietary testing solutions should be replaced sooner or later.

For this purpose, the methodology in the application of TTCN-3 should be better mediated and the tools should be oriented even stronger on the needs of the testing personnel. Up-to-date methods of software development should also be offered for the development of testing systems. This includes, among others, refactoring, metrics, verifications, debugging, and even simulations and tests. A major development is the model-based testing methodology [5], which in combination with the TTCN-3 middleware is exploiting its potential at a maximum.

References

1. Rao, G.G., Weigert, T.: Network Element Testing using TTCN-3: Benefits and Comparison. In: SDL Forum 2005, Grimstadt, Norway (June 2005)
2. Ji, W.: TTCN-3 test technique evaluation report, Technical Report, Joint Sino-German Institute, Beijing, China (Mai 2006)
3. Myers, G.J.: The Art of Software Testing. Wiley, Chichester (2004)
4. EUREKA TestNGMed Project: Test automation for next generation medical systems (2010), `http://www.testngmed.org` (Last access August 4, 2010)
5. ITEA D-MINT Project: Deployment of Model-Based Technologies to Industrial Testing (2010), `http://www.d-mint.org` (Last access August 4, 2010)
6. Din, G.: A Workload Realization Methodology for Performance Testing of Telecommunication Services, PhD Thesis, TU Berlin, Faculty Electrical Engineering and Computer Science (September 2008)
7. ETSI ES 201 873-1: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language (July 2010)
8. ETSI ES 201 873-4: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Operational Semantics (July 2010)
9. ETSI ES 201 873-5: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI) (July 2010)
10. ETSI ES 201 873-6: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI) (July 2010)
11. ETSI ES 201 873-7: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3 (July 2010)
12. ETSI ES 201 873-8: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 8: The IDL to TTCN-3 Mapping (July 2010)

13. ETSI ES 201 873-9: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Use XML with TTCN-3 (July 2010)
14. ETSI ES 201 873-10: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 10: TTCN-3 Documentation Comment Specification (July 2010)
15. ETSI ES 202 781: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Configuration and Deployment Support (July 2010)
16. ETSI ES 202 784: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Advanced Parameterization (January 2010)
17. ETSI ES 202 785: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Behaviour Types (January 2010)
18. ETSI ES 202 782: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: TTCN-3 Performance and Real Time Testing (July 2010)

A Model-Based Approach to Testing Software for Critical Behavior and Properties

Constance Heitmeyer

Naval Research Laboratory
Washington, DC, United States
heitmeye@itd.nrl.navy.mil

Abstract. To integrate the theoretical concepts of composition and refinement with the engineering notions of software models and components, the Naval Research Laboratory has formulated a set of practical composition-based methods, with associated modeling and proof techniques, for developing critical software systems. The general approach is to develop a set of software components and to use various forms of composition to combine the components in a manner that guarantees properties of the composite system. An assumption underlying this research is that much of the software code can be generated automatically from models using automatic code generators. A problem is that the code generated by such tools still requires testing to ensure that the software delivers its critical services correctly and that the software behavior satisfies critical properties, such as safety properties. The need for testing arises in part because only some of the required code is generated automatically: Stubs are provided for code that cannot be generated automatically (for example, certain algorithms), and such code must be constructed manually. This talk describes model-based methods for developing software, and how the models and properties developed using these methods can be used as the basis for automatically constructing tests for evaluating the correctness of software code. These tests are designed to satisfy various coverage criteria, such as branch coverage. An example is presented showing how our model-based method can be used to construct a suite of tests for evaluating the software code controlling the behavior of an autonomous system.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, p. 15, 2010.

A Pareto Ant Colony Algorithm Applied to the Class Integration and Test Order Problem

Rafael da Veiga Cabral, Aurora Pozo, and Silvia Regina Vergilio*

Computer Science Department, Federal University of Paraná
CP: 19081, CEP 19031-970, Curitiba, Brazil
{rafaelv,aurora,silvia}@inf.ufpr.br

Abstract. In the context of Object-Oriented software, many works have investigated the Class Integration and Test Order (CITO) problem, proposing solutions to determine test orders for the integration test of the program classes. The existing approaches based on graphs can generate solutions that are sub-optimal, and do not consider the different factors and measures that can affect the stubbing process. To overcome this limitation, solutions based on Genetic Algorithms (GA) have presented promising results. However, the determination of a cost function, which is able to generate the best solutions, is not always a trivial task, mainly for complex systems with a great number of measures. Therefore, we introduce, in this paper, a multi-objective optimization approach to better represent the CITO problem. The approach generates a set of good solutions that achieve a balanced compromise between the different measures (objectives). It was implemented by a Pareto Ant Colony (P-ACO) algorithm, which is described in detail. The algorithm was used in a set of real programs and the obtained results are compared to the GA results. The results allow discussing the difference between single and multi-objective approaches especially for complex systems with a greater number of dependencies among the classes.

Keywords: Integration testing, object-oriented software, multi-objective, ant colony algorithm.

1 Introduction

Software test is considered a fundamental activity to ensure software quality. It should be conducted in an incremental strategy [16]. In the context of Object Oriented (OO) software, this strategy includes different levels [10,2]: method, class, cluster and system levels. In general terms, the base code (or component) should be developed, unit-tested, integrated and tested. A common problem in the integration phase is to determine the order in which classes are integrated and tested. This order is generally referred to the inter-class test order and is important because it affects [17]: the order in which classes are developed; the design of test cases; the number of created stubs for classes; the order in which inter-class faults are detected.

Sometimes, the class under test requires another class to be available before it can be executed. This kind of relationship is named dependency between classes, and can

* This work is partially supported by CNPq.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 16–29, 2010.

require the creation of a stub to emulate the functionality that is used by the first class. The creation of stubs is an expensive and error-prone operation, and for that reason, the minimization of the number of stubs created during the integration testing is an important problem to be solved, called the Class Integration and Test Order (CITO) problem [1].

When there are no dependency cycles, the CITO problem can be solved by a simple reverse topological ordering of classes considering their dependency [12]. However, recent studies with real Java systems show that the presence of complex cycles is very common [13], and solutions to find an optimal order to minimize the stubbing effort are essential. Given such importance, we find in the literature, many works that address this subject. These works propose solutions based on graphs [17,1,12,18,6]. In general, the solution is obtained by removing, from the graph, dependencies that maximize the number of broken cycles. However, many times, these solutions are sub-optimal. Other disadvantage of the graph based approaches is that the cost to construct a stub may depend on many factors and can not be completely measured or estimated [4]. For example, number of attributes of a class, number of calls or distinct methods invoked, constraints related to organizational or contractual reasons, etc. To adapt the most graph-based solutions to consider these factors seems difficult or even impossible.

Due to the computational complexity, the CITO problem has been subject of the Search Based Software Engineering (SBSE), a recent research field that explores the use of meta-heuristic techniques to the Software Engineering problems [9]. A solution based on Genetic Algorithms (GA) was proposed in [4]. The authors use coupling measures to determine the stubbing complexity. The implemented GA is evaluated in real programs by using different four functions: number of broken dependencies, number of attributes, number of methods, and a geometric average of attributes and methods [3]. The obtained results are very promising when compared with the graph based solutions, which consider only the dependency. This solution allows considering different factors to establish the test orders. However, the choice of the more adequate evaluation function for the GA is not always a trivial task. The authors propose a procedure to find weights for all coupling measures, which is based on subjective steps and can be very expensive and labor-intensive for complex cases. This makes difficult the use of the solution based on GA in practice.

To overcome this limitation and to obtain solutions more adequate that consider the real constraints and diverse factors that may influence the CITO problem, we propose, in this paper, the use of a multi-objective search based approach. Such approach treats the CITO problem as a combinatorial constrained multi-objective optimization problem, more precisely, a problem where the goal is to find a set of test orders which satifies constraints and optimizes different factors.

To implement and evaluate the introduced approach, we can find in the literature many multi-objective algorithms. In this work, a multi-objective algorithm based on Ant Colony Optimization [8,7] is explored. This algorithm is a meta-heuristic and has been successfully applied to solve many combinatorial optimization problems, such as, traveling salesman problem, sequential ordering problem, set covering problem, etc. To apply Ant Colony Optimization (ACO), the optimization problem is transformed into

the problem of finding the best path on a weighted graph. For that reason, we consider ACO one of the most suitable algorithm to the CITO problem.

The solutions returned by the multi-objective algorithm are evaluated according to Pareto dominance concepts [14] and represent a good trade off between the coupling measures: number of methods and attributes. To allow comparison with the GA approach, we conducted an experiment using the same benchmark used by Briand et al [3]. This benchmark is composed by real systems with varying complexities, what permits good insights about the use of both approaches. The results point out that the multi-objective approach presents a variety of good solutions even for complex cases.

The paper is organized as follows. Section 2 presents a review of works related to the CITO problem including the GA approach. Section 3 introduces our multi-objective approach and describes the implemented Ant Colony algorithm. Section 4 discusses the experimental results obtained. Section 5 concludes the paper and contains future research works.

2 The Class and Integration Test Order Problem

The integration test has the objective to detect faults associated to the interfaces between the modules when these are integrated to build the structure of the software, established in the project phase. It checks if the integrated system components work as desired. In the Object Oriented (OO) context the modules are classes that need to be integrated one at a time or, in some case in small groups. Inside this context, the CITO problem can be described as the identification of a priority order for integrating the classes. Once, an order of integration of the components has been assumed, this order can cause the implementation of components called stubs needed to simulate the behavior of tested and not yet integrated classes. The stubs represent additional costs to the project and its number must be reduced to the possible minimum. The minimization of the number of stubs created during the integration testing is an important problem to be solved, called the Class Integration and Test Order (CITO) problem [1].

Most approaches to the CITO problem are based on directed graphs, named ORD (Object Relation Diagrams), where the nodes represent classes and the edges represent their relationships. When there are no dependency cycles between the classes, the CITO problem can be solved by a simple reverse topological ordering of classes considering their dependencies [12]. However, this is not always the case, because most systems contain cycles. The approach proposed by Kung et al [12] was the first one to address this problem. The idea of this work is to identify strongly connected components (SCCs) in the graph and removing associations until no cycles remain. When there are more than one candidate associations for cycle breaking, a random selection is performed.

Tai and Daniels [17] define two class levels. Major-level numbers are assigned to classes based on inheritance and aggregation dependencies only. Then within each major level, minor-level numbers are assigned, based on association dependencies only. SCCs are identified in the major level and each edge of the SCCs receives a weight based on the related incoming and outgoing dependencies. Edges with higher weights

are selected to break cycles because it is supposed to be related with more cycles. However, according to Briand et al [6] this assumption is not always true. There are cases where class associations are not involved in cycles, and this solution is suboptimal in terms of the required number of test stubs.

In the work of Le Traon et al [18] the weights are assigned by the sum of incoming and outgoing frond dependencies for a given class within the SCC identified by Tarjan's algorithm. The frond dependency is a kind of edge, which is defined as going from a vertex (class) to one of its ancestors (a vertex that is traversed before it in a depth-first search that is the class depends on it, directly or indirectly). For each nontrivial SCC (with more than one vertex), the procedure above is then called recursively. The approach is non-deterministic because different sets of edges can be labeled as frond depending on the starting node, and when two or more nodes have the same weight, the selection is arbitrary.

A graph-based approach that combines the works of Le Traon et al and Tai and Daniels was proposed by Briand et al [6]. They also use Tarjan's algorithm to identify SCCs. The association edges in the SCCs are assigned with weights corresponding to the estimated number of involved cycles. This number is calculated according to the number of incoming and outgoing dependencies. The edge with highest weight is removed. The process is repeated until no SCC remains. The main advantages of this approach are that it does not break inheritance and aggregation edges, and computes the weights in a more precise way.

The works mentioned above present several limitations [4]. The most graph-based solutions consist in recursively identifying SCCs and in each SCC removing one dependency that maximizes the number of broken cycles. They optimize the decision without determining the consequences on the ultimate results. There are situations where breaking two dependencies has a lower cost than breaking only one that would make the graph acyclic in one step. Other disadvantage pointed out by Briand et al [4] is that the cost to construct a stub may be depend on many factors and can not be completely measured or estimated. For example, number of attributes of a class, number of calls or distinct methods invoked, constraints related to organizational or contractual reasons, and etc. To adapt the most graph-based solutions to consider these factors seems difficult or even impossible.

The work of Abdurazik and Offutt [1] consider more information in the minimization of the stubbing effort. The weights are derived from quantitative analysis of nine introduced kind of couplings, and are assigned to the edges and nodes. The coupling measures use number of parameters, number of return value types, number of variables and number of methods. The node weight is related to the estimated cost of removing it. If a class is used by multiple classes, then all or part of the same stub for that class may be shared among all classes that use it, thus reducing the cost of stubbing. The weight of a node is at least as high as the maximal weight of all incoming edges (assuming total sharing of the stub), and no higher than the sum of the weights of all incoming edges (assuming no sharing of the stub). The evaluation of this approach present positive results when compared with the approaches described before.

With this same objective, to allow the use of different kind of constraints and coupling measures, the approach based on Genetic Algorithms [4] has presented the most

Table 1. Solutions found by GA algorithm

ATM System										
Function	1	2	3	4	5	6	7	8	9	10
Dependencies	(52,19)	(54,19)	(67,13)	(67,19)	(52,19)	(46,19)	(46,19)	(45,19)	(59,19)	(47,13)
Attributes	**(39,13)**	(39,19)	**(39,13)**	(39,19)	(39,19)	(39,19)	**(39,13)**	**(39,13)**	(39,19)	**(39,13)**
Methods	**(39,13)**	(67,13)	(59,13)	(67,13)	(46,13)	(46,13)	(60,13)	(61,13)	**(39,13)**	(67,13)
Average	**(39,13)**	**(39,13)**	**(39,13)**	**(39,13)**	**(39,13)**	**(39,13)**	**(39,13)**	**(39,13)**	**(39,13)**	**(39,13)**

ANT System										
	1	2	3	4	5	6	7	8	9	10
Dependencies	(187,26)	(187,26)	**(157,26)**	(187,26)	(187,26)	(213,22)	(187,26)	**(157,26)**	**(157,26)**	**(157,26)**
Attributes	**(131,33)**	**(131,33)**	**(131,33)**	**(131,33)**	**(131,33)**	**(131,33)**	**(131,33)**	**(131,33)**	**(131,33)**	**(131,33)**
Methods	**(178,19)**	(184,19)	(184,19)	(197,22)	(227,22)	(227,22)	(197,22)	(229,22)	(226,22)	(197,22)
Average	**(136,29)**	**(136,29)**	**(136,29)**	**(136,29)**	**(136,29)**	**(136,29)**	**(136,29)**	**(136,29)**	**(136,29)**	**(136,29)**

BCEL System										
Functions	1	2	3	4	5	6	7	8	9	10
Dependencies	(128,73)	(128,70)	(125,72)	(125,75)	(128,73)	(125,72)	(127,73)	(127,73)	(125,72)	(125,72)
Attributes	(47,86)	(47,87)	(47,85)	(46,84)	(46,85)	(46,84)	(46,76)	(46,83)	(46,85)	(46,84)
Methods	(131,67)	(125,70)	(131,70)	(131,70)	(131,67)	(134,69)	(133,69)	(134,69)	(138,70)	
Average	(56,70)	(55,72)	(58,72)	**(48,73)**	(54,73)	(53,73)	(59,73)	**(47,74)**	(59,73)	(50,74)

DNS System										
Functions	1	2	3	4	5	6	7	8	9	10
Dependencies	(22,11)	(28,11)	(28,11)	**(19,11)**	(22,11)	**(19,11)**	(22,11)	(28,11)	(28,11)	**(19,11)**
Attributes	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**
Methods	(28,11)	(28,11)	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	(22,11)	(28,11)	(28,11)	(22,11)
OCplx	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**	**(19,11)**

SPM System										
Functions	1	2	3	4	5	6	7	8	9	10
Dependencies	(149,28)	(149,28)	(149,28)	(149,28)	**(146,27)**	**(146,27)**	(149,28)	(149,28)	(149,28)	(149,28)
Attributes	**(146,27)**	**(146,27)**	**(146,27)**	**(146,27)**	**(146,27)**	**(146,27)**	**(146,27)**	**(146,27)**	**(146,27)**	**(146,27)**
Methods	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**
Average	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**	**(148,26)**

promising results. The authors used a fitness cost function based on different coupling measures. Number of attributes and methods necessary for the stubbing procedure are considered besides the dependency factor. Briand et al [3,5] conducted an experiment with a set of real programs and studied four different fitness functions: 1) number of dependencies; 2) method coupling; 3) attributes coupling; and 4) an aggregation of attribute and method coupling given by a geometric average. For each function the GA was executed 10 times. Thus, they have generated 40 solutions for each of the mentioned system. The solutions reported by Briand et al [3,5] are presented in Table 1. For example, we can observe that for ATM System, the GA approach found the best solutions (highlighted entries in the table), with respectively 39 attributes and 13 methods, by using an average as fitness function.

The aggregation function is in fact a multiple objectives cost function, and is an alternative method to use a mono-objective algorithm to solve a multi-objective problem. However, Briand et al [4,3] conclude that a practical issue when using a multiple objectives cost function is to determine appropriate weights. For that reason, Briand et al [4,3] suggest a set of steps to select the best weights to be used. This set involves subjective aspects, such as the choice of minimal values for the measures. In complex cases the best solution can never be reached and the proposed procedure can be very difficult to apply for a great number of coupling measures or for complex cases.

Therefore, we introduce in next section a multi-objective approach, which obtains a set of good solutions and achieves a balanced compromise between the measures.

3 A Multi-objective Approach

We observe in the last section that the CITO problem is a constrained multi-objective optimization problem as it may involve a number of different objectives; different coupling measures can be used as well as diverse factors that may influence the stubbing process. Therefore, we introduce, in this section, a multi-objective approach. We describe a representation for the solutions, objectives to be evaluated and implemented algorithm - a Multi-objective Ant Colony algorithm.

3.1 Multi-objective Optimization

Optimization problems with two or more objective functions are called multi-objective. In such problems, the objectives to be optimized are usually in conflict, which means that they do not have a single solution. In this way, the goal is to find a good "trade-off" of solutions that better represent the possible compromise among them.

The general multi-objective maximization problem (with no restrictions) can be stated as to maximize Equation 1.

$$\overrightarrow{f}(\overrightarrow{x}) = (f_1(\overrightarrow{x}), ..., f_Q(\overrightarrow{x})) \tag{1}$$

subjected to $\overrightarrow{x} \in \Pi$, where: $\overrightarrow{x}$ is a vector of decision variables and Π is a finite set of feasible solutions.

Let $\overrightarrow{x} \in \Pi$ and $\overrightarrow{y} \in \Pi$ be two solutions. For a maximization problem, the solution $\overrightarrow{x}$ dominates $\overrightarrow{y}$ if:

$$\forall f_i \in \overrightarrow{f}, i = 1...Q, f_i(\overrightarrow{x}) \geq f_i(\overrightarrow{y}), and\ \exists f_i \in \overrightarrow{f}, f_i(\overrightarrow{x}) > f_i(\overrightarrow{y})$$

$\overrightarrow{x}$ is a non-dominated solution if there is no solution $\overrightarrow{y}$ that dominates $\overrightarrow{x}$.

The goal is to discover solutions that are not dominated by any other in the objective space. A set of non-dominated objective vectors is called Pareto optimal and the set of all non-dominated vectors is called Pareto Front.

The Pareto optimal set is helpful for real problems, for example, engineering problems. It provides valuable information about the underlying problem [11]. In most applications, the search for the Pareto optimal is NP-hard [11], then the optimization problem focuses on finding an approximation set, as close as possible to the Pareto optimal.

3.2 Representing the CITO Problem as Multi-objective

An important issue in the implementation of a meta-heuristic algorithm is the chosen representation to describe the solutions of the problem. This choice will influence on the implementation of all the stages of the algorithm. In this case, the chosen representation for the problem is simple, with the solution being represented by a vector whose positions assume an integer number in the interval [1, N], being N the number of classes. Thus, being each class represented by a number, an example of valid solution for a

problem with 10 class would be (2,8,1,3,10,4,5,6,7,9). In this example, the first class to be tested and integrated would be the class represented by number '2'. A solution for the CITO problem is a permutation of the classes.

Other issue related to multi-objective algorithms is the choice of the objective functions. As mentioned before, many possible measures and factors can be used to the CITO problem: for example, coupling, cohesion, fault-proneness, contractual, process and time constraints, etc. Different meta-heuristics can be used. In this work, we chose an ACO algorithm and to allow comparison with the GA approach, we use two functions based on the same coupling measures used in the works of Briand et al [4,3].

The stubbing complexity of an order o is based on its attribute and method coupling. Two complexities are then calculated in the following way:

- $ACplx(o)$ (attribute complexity): The attribute complexity counts the maximum number of attributes that would have to be handled in the stub if the dependency were broken (the number of attributes locally declared in the target class when references/pointers to the target class appear in the argument list of some methods in the source class).This information is an input for the algorithm and is represented by a matrix $A(i, j)$, where rows and columns are classes and i depends on j. Then, for a given test order o and a set of d dependencies to be broken, the attribute complexity $ACplx$ is calculated according to Equation 2.

$$ACplx(o) = \sum_{i=1,n} \sum_{j=1,n} A(i, j); j \neq k \tag{2}$$

  Where n is the total number of classes and k is any class included before the class i, in test order o.
- $MCplx(o)$ (method complexity): The method complexity counts the number of methods that would have to be emulated in the stub if the dependency were broken (the number of methods locally declared in the target class which are invoked by the source class methods). This information is an input for the algorithm and is represented by a matrix $M(i, j)$, where rows and columns are classes and i depends on j. Then, for a given test order o and a set of d dependencies to be broken, the method complexity $MCplx$ is computed as defined by Equation 3.

$$MCplx(o) = \sum_{i=1,n} \sum_{j=1,n} M(i, j); j \neq k \tag{3}$$

  Where n is the total number of classes and k is any class included before the class i, in test order o.
- Constraints: In this work, following Briand et al [4,3] work, Inheritance and Composition dependencies cannot be broken. This means the base/container classes must precede child/contained classes in any order test order o. The dependencies that cannot be broken are inputs for the algorithm, provided by a precedence table.

Based on the measures and constraints presented above, the problem is the search for an order that minimizes two objectives: the method and attribute complexities.

3.3 Multi-objective Ant Colony Optimization Algorithm

The Ant Colony Optimization Algorithm (ACO) was introduced by [8]. ACO is inspired by the behavior of real ant colonies, in particular, by their foraging behavior. One of its main ideas is the indirect communication among the individuals of a colony or agents, called (artificial) ants, based on an analogy with trails of a chemical substance, called pheromone, which real ants use for communication. The (artificial) pheromone trails are a kind of distributed numeric information which is modified by the ants to reflect their experience accumulated while solving a particular problem.

The basic idea of ACO algorithms come from the ability of ants to find shortest paths from their nest to food locations. Considering a combinatorial optimization problem, an ant iteratively builds a solution. This constructive procedure is conducted using at each step a probability distribution, which corresponds to the pheromone trails in real ants. Once a solution is completed, pheromone trails are updated according to the quality of the best solution built. Hence, cooperation between ants is performed by a common structure that is the shared pheromone matrix. In addition to this, the algorithm discussed in this paper is based on the Pareto Ant Colony (P-ACO) algorithm, which is based on the Ant Colony System algorithm and was originally proposed to solve the Multiobjective Portfolio Selection problem [7].

P-ACO works with k pheromone matrices, where k is the number of objectives, and uses an aggregation heuristic function computed from the k objectives. The transition rule used to choose the next class j to be included in a test order o is given by Equation 4.

$$j = \begin{cases} \text{argmax}_{j \in U} \left[\sum_{k=1}^{K} p_k \cdot \tau_{ij}^k \right]^{\alpha} \cdot \eta_{ij}^{\beta} & \text{if } q \leq q_0 \\ p\,(j) & \text{otherwise} \end{cases} \tag{4}$$

where $p(j)$ is given by the probability, represented in Equation 5.

$$p(j) = \begin{cases} \dfrac{\left[\sum_{h=1}^{K} p_k \cdot \tau_{ij}^k \right]^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{h \in U} \left[\sum_{h=1}^{K} p_k \cdot \tau_{ij}^k \right]^{\alpha}} & \text{if } j \in U \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where the pheromone matrices are represented by τ^k.

According to Pasia et al [15] this rule is a straightforward extension for multiple pheromone matrices of the rule used in the Ant Colony System [8]. The parameters α and β determine the relative influence of the pheromone and heuristic information, respectively; η_{ij}^k is the heuristic information of the objective k, $p^k \in [0, 1]$ are weights, which are uniformly distributed such that $\sum_{k=1}^{k} = 1$. In each iteration, a weight vector is assigned to each ant. The parameter q is a random number uniformly distributed in the interval [0, 1], and $q_0 \in [0, 1]$ is a parameter that defines the intensification and diversification properties of the algorithm.

The local pheromone update for all matrices is performed whenever an ant chooses a sequence of classes (i, j) and uses the following rule: $\tau_{ij}^k = (1 - \rho) \cdot \tau_{ij}^k + \rho \cdot \tau_0$, where ρ is the rate of evaporation and τ_0 is the initial pheromone value.

The global pheromone update is performed after all ants of the population have built a test order. For each objective k best and second-best solution is determined and then,

Algorithm 1. Pseudocode of P-ACO

```
/* Let C the number of classes */
/* Let N the number of iterations and Ants the number of ants */
F1, F2 are the objective functions
Initialize pheromone (F1, F2, τ0)
while nriter ≤ N do
   for all ant in Ants do
      p1 = rand (0, 1)
      p2 = 1 − p1
      TakeInitialCandidates = ()
      s = GenerateInitialPath ()
      BuildPath s = (s, q, q0, p1, p2, F1, F2)
      LocalPheromoneUpdate (s, F1, F2)
      s1 = LocalSearch (s, F1)
      s2 = LocalSearch (s, F2)
   end for
   for all objective k do
      Determine best b and second-best b'
      GlobalPheromoneUpdate (b, b', Fk)
   end for
   ParetoSetUpdate (P, s1, s2)
   nriter += 1
end while
```

the following rule is used: $\tau_{ij}^k = (1-\rho)\cdot\tau_{ij}^k + \rho\cdot\Delta\tau_{ij}^k$, where $\Delta\tau_{ij}^k$ receives as value: 15 if the (i,j) component belongs to the best and second best solutions; 10 if (i,j) belongs only to the best path; 5 if it belongs just to the second best path; and 0 otherwise.

Algorithm 1 describes the main steps of the P-ACO algorithm. First, an initial procedure is executed where the pheromone matrices are set to τ_0. The next step is an iterative loop, at each iteration, all the ants built a set, s, a test order solution.

The *Build_Path* procedure is used for each ant to build a test order. The ant chooses, based on a probabilistic decision (Equation 4), a class that has not yet been included in the test order s. The ant uses a candidate list composed by classes which do not have any precedence constraint, that is, either the class does not have any precedence by itself or all its precedence's classes had already been placed on the test order vector (precedence table). When the ant obtains a test order vector a local update is performed(`LocalPheronomeUpdate`).

After then, local searches are performed to achieve a strong exploitation of the search space. For each objective, a local search is performed on each ant (`LocalSearch`).

Finally, all ants are evaluated and a global pheromone update is performed (`GlobalPheromoneUpdate`) based on the best solutions of the iteration. The archive of the best solutions found so far (that is, the approximation of the Pareto front), is also updated (`ParetoSetUpdate`). At the end of the iterative loop, the solutions presented in the archive are the solutions obtained by the algorithm.

4 Experimental Results

The goal of this section is to describe the application of the P-ACO algorithm for the CITO problem in real systems and to compare its results with the GA approach. To do this, we used the same benchmark from the work of Briand et al [3]. This benchmark is composed of five real systems: ATM, ANT, SPM, BCEL and DNS. According to [3] these systems are deemed to be of sufficient size and of varying complexity. This allows a better evaluation considering different characteristics of the systems, which are described in Table 2. The systems ATM, ANT, SPM, and BCEL have class diagrams of reasonable (and comparable) sizes (between 19 and 45), but with very different numbers of cycles (from 30 for ATM to 416,091 for BCEL). On the other hand, the DNS system has the greatest number of classes and almost the same number of relationships of BCEL system, but the smallest number of cycles (fewer number than ATM, ANT, and SPM).

The P-ACO algorithm was executed with the following parameters: $q_0 = 0.4$, $\alpha = 1$, $\beta = 1$, $\tau_{min} = 0.0001$ and $\tau_0 = 1.0$. These values were got from the original P-ACO [7]. The number of ants is equal to the number of classes of each system, and the number of iterations is equal to twice this number except for the DNS system, where a lower number was used. These parameters are presented in Table 3. Observe that the population (number of ants) and iterations are lower than the values used by Briand et al [3]. The GA used a population size of 100 and a number of generation of 500.

As explained on Section 3.3, the P-ACO algorithm uses two local searches, on this implementation, a neighbor of 20 is explored and the stopping criterion is a number of 100 iterations without improvement on the best solution. The P-ACO algorithm was executed five times for each system. A performance evaluation between the proposed algorithm and the Briand et al [3] algorithm is not possible because their algorithm is not available. Then, this work tries to understand the benefit of one approach or the other based on the differences and similarities of the results.

Table 2. Detailed Information about the Systems

System	Classes	Uses	Associations	Compositions	Inheritance	Cycles	LOC
ATM	21	39	9	15	4	30	1390
ANT	25	54	16	2	11	654	4093
SPM	19	24	34	10	4	1178	1198
BCEL	45	18	226	4	46	416,091	3033
DNS	61	211	23	12	30	16	6710

Table 3. Number of Ants and Iterations

System	ATM	ANT	SPM	BCEL	DNS
Ants	20	20	20	40	60
Iterations	40	40	40	80	80

4.1 Results and Analysis

The P-ACO results are presented for each system in Table 4. Each line presents the attribute and method complexities of the solutions for one independent run of the P-ACO algorithm. The non-dominated solutions, considering all the executions are highlighted. In this section, the obtained results are compared to the GA results presented in Table 1, considering the approximation of the Pareto front.

For ATM system the P-ACO algorithm found solutions with attribute complexity of 13 and method complexity of 39. These values are the best known approximation of the Pareto Front. For the GA algorithm, the same solutions are found when the used function is the average of the attribute and method complexities, as mentioned before. On the other hand, when the function uses only one complexity measure, the solutions are only good with respect to the metric employed.

In the ANT system, the P-ACO algorithm found solutions with better balance between the attribute and method complexities, forming the best approximation of the Pareto Front. The GA algorithm also found some of these solutions when the used function is the average of the attribute and method complexities. However, P-ACO has found two more solutions in the approximation of the Pareto Front. Again, when the GA algorithm uses a function based only on one complexity metric, the solutions are only good with respect to the metric employed, but the GA has more problems to find the best solutions and some runs found sub-optimal solutions.

For SPM, the approximation of the Pareto Front contains two non-dominated solutions with (146,27) and (148,26) respectively for attribute and method complexities. The P-ACO and GA algorithms found these solutions. But, the GA has found one solution when the attribute complexity function is used and the other one when the method

Table 4. Solutions found by P-ACO algorithm

ATM System	
1	**(39,13)**
2	**(39,13)**
3	**(39,13)**
4	**(39,13)**
5	**(39,13)**

ANT System						
1	**(157,26)**	**(136,29)**	(184,19)	**(162,25)**	(183,22)	**(131,33)**
2	**(136,29)**	**(157,26)**	(183,22)	(168,25)	(184,19)	
3	**(157,26)**	**(136,29)**	(184,19)	(183,22)	(170,25)	**(131,33)**
4	**(157,26)**	**(136,29)**	**(178,19)**	**(163,22)**		
5	**(157,26)**	(162,25)	**(136,29)**	(183,22)	(184,19)	**(131,33)**

BCEL System								
1	(45,77)	(130,66)	(57,69)	**(78,68)**	(55,71)	(54,73)		
2	(79,69)	(128,68)	(129,67)	(45,77)	**(49,72)**	(131,66)	(54,71)	
3	(129,68)	(45,77)	(98,70)	**(46,75)**	(130,67)	(133,66)	(52,72)	(56,71) (126,69)
4	(45,77)	**(130,66)**	**(54,69)**	(53,75)	(50,76)			
5	(105,68)	(57,70)	**(45,76)**	(54,71)	(134,66)	**(127,67)**		

DNS System	
1	**(19,11)**
2	**(19,11)**
3	**(19,11)**
4	**(19,11)**
5	**(19,11)**

SPM System		
1	**(146,27)**	**(148,26)**
2	**(146,27)**	**(148,26)**
3	**(148,26)**	**(146,27)**
4	**(146,27)**	**(148,26)**
5	**(148,26)**	**(146,27)**

complexity function is used. Moreover, these solutions were not found when the aggregated function is used.

For BCEL, note that there are only two solutions from the GA algorithm in the approximation of the Pareto Front. These solutions have (47,74) and (48,73) for attribute and method complexity respectively, and they were found by the function that aggregates both complexity measures. Besides these solutions, the P-ACO algorithm found another six solutions that are on the approximation of the Pareto Front. The BCEL System has the greatest number of solutions in the approximation of the Pareto Front, and this fact reveals a difference between the comparative study between GA and P-ACO algorithms.

The DNS System has the greatest number of classes, however, its complexity seems like to the ATM system since for all P-ACO executions only one solution was found with (19,11) as attribute and method complexities, respectively. These values are the best known approximation of the Pareto Front. For the GA algorithm, the same solutions are found when the used function is the aggregation of the attribute and method complexity. On the other hand, when the function uses only one complexity metric, the solutions are only good with respect to the metric employed.

Table 5 presents a comparison between the number of solutions found in the approximation of the Pareto Front by the compared algorithms. It is possible to observe that GA and P-ACO present similar behaviour for some systems. This behaviour can be explained because some systems have similar measures for attribute and method complexities, that is, when the attribute complexity grows the method complexity also grows. On the other hand, systems like BCEL exhibit different behaviour for attribute and method complexities. In these cases the P-ACO algorithm is the most suitable.

It is important to remark that the GA used four different complexity functions and the solutions were not always found by the same function. Consequently, some effort must be spent to determine the best function for the GA approach. It seems that this

Table 5. Number of Solutions on the Approximation of the Pareto Front

Systems	ATM	ANT	SPM	BCEL	DNS
P-ACO	1	6	2	6	1
GA	1	4	2	2	1
Total	1	6	2	8	1

Table 6. Attribute and Method Complexities of BCEL system

Solutions	Algorithms	Attribute Complexity	Method Complexity
1	P-ACO	45	76
2	P-ACO	46	75
3	GA	47	74
4	GA	48	73
5	P-ACO	49	72
6	P-ACO	54	69
7	P-ACO	68	78
8	P-ACO	130	66

effort is greater for complex systems, with a great number of dependencies between classes, such as BCEL. This does not happen with the P-ACO approach.

Table 6 details the solutions found by P-ACO and GA algorithms for BCEL system. In this table it is possible to note different solutions that have in common a good trade off between the two complexity metrics. Remark that all these solutions are non dominated, hence no one can be considered to be better than any other with respect to both complexity metrics.

When the P-ACO approach is used, the tester can take advantage from the variety of these solutions according to his (or her) preferences (needs). He (or she) can conduct the integration test by prioritizing either attribute or method complexities. On the other hand, the tester can use other preference information about the classes, such as constraint related to contractual aspects to select an order from a larger range of good solutions than if the GA approach is used.

5 Conclusions

In this paper we introduce a new approach based on multi-objective optimization to the CITO problem. The approach uses a Pareto Ant Colony algorithm that was adapted to produce test orders that represent a good tradeoff between the number of attributes and methods in the stubbing process.

The algorithm was evaluated in a benchmark of five real programs and its performance was compared to the GA performance by considering the approximation of the Pareto front. In this evaluation we observe that the multi-objective is very advantageous because different factors can be considered and a set of good solutions that achieve a balanced compromise between the considered measures is obtained without human intervention. In addition to this, the approach is applicable and present better results in complex cases, when the system being tested contains a large number of dependency cycles.

It is possible to argue that the greater the number of solutions found in the approximation of the Pareto Front and their distributions, the greater the ways to satisfy integration test plans considering the real world needs. This fact points out that the P-ACO algorithm and the research in multi objective combinatorial problems are more suitable to solve software engineering complex problems, such as the CITO.

Now, we intend to conduct experiments with other multi-objective algorithms, such as Non-dominated Sorting Genetic Algorithm (NSGA-II). Besides this, other objectives can be included in the CITO problem, as mentioned in this text. The performance of the P-ACO with more than two objectives should be evaluated in further studies, with other benchmarks.

References

1. Abdurazik, A., Offutt, J.: Coupling-based class integration and test order. In: International Workshop on Automation of Software Test. ACM, Shanghai (May 2006)
2. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, Reading (2000)

3. Briand, L.C., Feng, J., Labiche, Y.: Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders. Carleton University, Technical Report SCE-02-03 (October 2002)
4. Briand, L.C., Feng, J., Labiche, Y.: Using genetic algorithms and coupling measures to devise optimal integration test orders. In: 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy (July 2002)
5. Briand, L.C., Feng, J., Labiche, Y.: Experimenting with genetic algorithms and coupling measures to devise optimal integration test orders. In: Proceedings of Software Engineering with Computational Intelligence, pp. 204–234. Kluwer Academic Publishers, Dordrecht (2003)
6. Briand, L.C., Labiche, Y.: An investigation of graph-based class integration test order strategies. IEEE Transactions on Software Engineering 29(7), 594–607 (2003)
7. Doerner, K., Gutjahr, W.J., Hartl, R.F., Strauss, C., Stummer, C.: Pareto ant colony optimization: A metaheuristic approach to multiobjective portfolio selection. Annals of Operation Research (131), 79–99 (2004)
8. Dorigom, M., Socha, K.: An Introduction to Ant Colony Optimization. No. TR/IRIDIA/2006-010., Technical Report - IRIDIA (April 2006)
9. Harman, M.: The current state and future of search based software engineering. In: Proceedings of International Conference on Software Engineering / Future of Software Engineering 2007 (ICSE/FOSE 2007), May 20-26, pp. 342–357. IEEE Computer Society, Minneapolis (2007)
10. Harrold, M.J., McGregor, J.D., Fitzpatrick, K.J.: Incremental testing of object-oriented class structures. In: 14th International Conference on Software Engineering, pp. 68–80. IEEE Computer Society, Melbourne (May 1992)
11. Knowles, J., Thiele, L., Zitzler, E.: A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizer. 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland (February 2006)
12. Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C.: A test strategy for object-oriented programs. In: 19th International Computer Software and Applications Conference. IEEE Computer Society, Los Alamitos (August 1995)
13. Melton, H., Tempero, E.: An empirical study of cycles among classes in Java. Empirical Software Engineering 12, 389–415 (2007)
14. Pareto, V.: Manuel D'Economie Politique. Ams Press, Paris (1927)
15. Pasia, J.M., Hart, R., Doerner, K.F.: Solving a bi-objective flowshop scheduling problem by Pareto-ant colony optimization. In: Dorigo, M., Gambardella, L.M., Birattari, M., Martinoli, A., Poli, R., Stützle, T. (eds.) ANTS 2006. LNCS, vol. 4150, pp. 294–305. Springer, Heidelberg (2006)
16. Pressman, R.: Software Engineering: A Practitioner's Approach. McGraw-Hill, New York (2006)
17. Tai, K.C., Daniels, F.J.: Test order for inter-class integration testing of object-oriented software. In: 21st International Computer Software and Applications Conference, pp. 602–607. IEEE Computer Society, Los Alamitos (August 1997)
18. Traon, Y.L., Jéron, T., Jézéquel, J.M., Morel, P.: Efficient object-oriented integration and regression testing. IEEE Transactions on Reliability, 12–25 (2000)

More Testable Properties*

Yliès Falcone[1], Jean-Claude Fernandez[2], Thierry Jéron[1], Hervé Marchand[1], and Laurent Mounier[2]

[1] INRIA, Rennes - Bretagne Atlantique, France
[2] Verimag, Université Grenoble I, France
Firstname.Lastname@inria.fr, Firstname.Lastname@imag.fr

Abstract. In this paper, we explore the set of testable properties within the *Safety-Progress* classification where testability means to establish by testing that a relation, between the tested system and the property under scrutiny, holds. We characterize testable properties wrt. several relations of interest. For each relation, we give a sufficient condition for a property to be testable. Then, we study and delineate, for each Safety-Progress class, the subset of testable properties and their corresponding test oracle producing verdicts for the possible test executions. Finally, we address automatic test generation for the proposed framework.

1 Introduction

Due to its ability to scale up well and its practical aspect, testing remains one of the most effective and widely used validation technique for software systems. However, due to recent needs in the software industry (for instance in terms of security), it is important to reconsider the classes of requirements this technique allows to validate or invalidate. The aim of a testing stage may be either to find defects or to witness expected behaviors on an implementation under test (IUT). From a practical point of view, a test campaign consists in producing a test suite (*test generation*) from some initial system description, and executing it on the system implementation (*test execution*). The test suite consists in a set of test cases, where each test case is a set of interaction sequences to be executed by an external tester (performed on the points of control and observation, PCOs). Any execution of a test case should lead to a *test verdict*, indicating if the system succeeded or not on this particular test (or if the test was not conclusive).

One way to improve the practical feasibility of a test campaign is to use a property to drive the test execution. In this case, the property is used to generate the so-called test purposes [2,3] so as to select the most relevant test case behaviors. A property may also represent the desired behavior of the system. In this setting, the property may be a formalization of a security policy describing prohibited behaviors and expectations from the users, as considered in [4,5]. Several approaches (*e.g.*, [6]) combine classical testing techniques and property verification so as to improve the test activity. Most of these approaches used

* An extended version of this paper with complete proofs can be found in [1].

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 30–46, 2010.

safety and co-safety properties. A natural question is the existence of other kinds of properties that can be "tested", *i.e.,* to define a precise notion of *testability.*

In [7,8], Nahm, Grabowski, and Hogrefe addressed this issue by discussing the set of temporal properties that can be tested on an implementation. A property is said to be *testable* if it is possible to determine if a given relation (*e.g.*, inclusion) holds between the sequences described by a property and the set of execution sequences that can be produced by interacting with the IUT, after the execution of a finite sequence on the IUT. In their work, testability of properties is studied wrt. the *Safety-Progress* classification ([9] and Section 3) for infinitary properties. The announced classes of testable properties are the safety and guarantee[1] classes. Then, it is not too surprising that most of the previously depicted approaches used safety and co-safety properties during testing.

Context. In this paper, we shall use the same notion of testability. We consider a generic approach, where an underlying property is compared to the possibly infinite execution sequences of the IUT by a tester. This property expresses finite and infinite[2] observable behaviors (which may be desired or not). Usually, IUT's execution sequences are expressed in a different alphabet than the one used to describe the property and have thus to be interpreted. However, testability and the test oracle problem (*i.e.,* the problem of deciding verdicts) can be studied while abstracting this alphabet discrepancy. A second characteristic is that we do not require the existence of an executable specification to generate the test cases. This allows to encompass several conformance testing approaches by viewing the specification as a special property.

Motivations and contributions. The main motivation of this paper is to leverage the use of an extended version of the *Safety-Progress* classification of properties dedicated to runtime techniques. We give a precise characterization of testable properties and provide a formal basis for several previous testing activities. We extend the results of [7] by showing that lots of interesting properties (neither safety nor guarantee) are also testable. Moreover, this framework allows to simply obtain test oracles producing verdicts according to the test execution.

Paper organization. The remainder of this paper is organized as follows. In Section 2, some preliminary concepts and notations are introduced. A quick overview of the *Safety-Progress* classification of properties for runtime validation techniques is given in Section 3. Section 4 introduces the notion of testability considered in this paper. In Section 5, testable properties are characterized. Automatic test generation is addressed in Section 6. Next, in Section 7, we overview the related work and propose a discussion on the results provided by this paper. Finally, Section 8 gives some concluding remarks and raised perspectives.

[1] In the *Safety-Progress* classification the guarantee class is the co-safety class in the *Safety-Liveness* classification.

[2] The tester observes a finite sequence of the IUT and should state a verdict about all potential continuations of this execution sequence (finite and infinite ones).

2 Preliminaries

Given an alphabet of actions Σ, a sequence σ on Σ is a total function $\sigma : I \to \Sigma$ where I is either the interval $[0, n]$ for some $n \in \mathbb{N}$, or $\mathbb{N}$ itself. The empty sequence is denoted by ϵ. We denote by Σ^* the set of finite sequences over Σ and by Σ^ω the set of infinite sequences over Σ. $\Sigma^* \cup \Sigma^\omega$ is noted Σ^∞. The length (number of elements) of a finite sequence σ is noted $|\sigma|$ and the $(i+1)$-th element of σ is denoted by σ_i. For $\sigma \in \Sigma^*, \sigma' \in \Sigma^\infty$, $\sigma \cdot \sigma'$ is the concatenation of σ and σ'. The sequence $\sigma \in \Sigma^*$ is a *strict prefix* of $\sigma' \in \Sigma^\infty$ (equivalently σ' is a *strict continuation* of σ), noted $\sigma \prec \sigma'$, when $\forall i \in [0, |\sigma| - 1] : \sigma_i = \sigma'_i$ and $|\sigma| < |\sigma'|$. When $\sigma' \in \Sigma^*$, we note $\sigma \preceq \sigma' \stackrel{\text{def}}{=} \sigma \prec \sigma' \vee \sigma = \sigma'$. For $\sigma \in \Sigma^\infty$ and $n \in \mathbb{N}$, $\sigma_{\dots n}$ is the sub-sequence containing the $n+1$ first elements of σ. The set of prefixes of $\sigma \in \Sigma^\infty$ is $pref(\sigma) \stackrel{\text{def}}{=} \{\sigma' \in \Sigma^* \mid \sigma' \preceq \sigma\}$. For a finite sequence $\sigma \in \Sigma^*$, the set of finite continuations is $cont^*(\sigma) \stackrel{\text{def}}{=} \{\sigma' \in \Sigma^* \mid \exists \sigma'' \in \Sigma^* : \sigma' = \sigma \cdot \sigma''\}$.

The IUT is a program $\mathcal{P}$ abstracted as a generator of execution sequences. We are interested in a restricted set of operations that influence the truth value of tested properties and are made on PCOs. We abstract these operations by an alphabet Σ. We denote by $\mathcal{P}_\Sigma$ a program with alphabet Σ. The set of execution sequences of $\mathcal{P}_\Sigma$ is denoted by $Exec(\mathcal{P}_\Sigma) \subseteq \Sigma^\infty$. This set is *prefix-closed*, that is $\forall \sigma \in Exec(\mathcal{P}_\Sigma) : pref(\sigma) \subseteq Exec(\mathcal{P}_\Sigma)$. We will use $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$ (resp. $Exec_\omega(\mathcal{P}_\Sigma)$) to refer to the finite (resp. infinite) execution sequences of $\mathcal{P}_\Sigma$, that is $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \stackrel{\text{def}}{=} Exec(\mathcal{P}_\Sigma) \cap \Sigma^*$ and $Exec_\omega(\mathcal{P}_\Sigma) \stackrel{\text{def}}{=} Exec(\mathcal{P}_\Sigma) \cap \Sigma^\omega$.

Properties as sets of execution sequences. A *finitary property* (resp. an *infinitary property*, a *property*) is a subset of execution sequences of Σ^* (resp. Σ^ω, Σ^∞). Given a finite (resp. infinite) execution sequence σ and a property ϕ (resp. φ), we say that σ *satisfies* ϕ (resp. φ) when $\sigma \in \phi$, noted $\phi(\sigma)$ (resp. $\sigma \in \varphi$, noted $\varphi(\sigma)$). A consequence of this definition is that properties we will consider are restricted to *linear time* execution sequences, excluding specific properties defined on powersets of execution sequences and branching properties.

Runtime properties [10]. Runtime properties should characterize satisfaction for both kinds of sequences (finite and infinite) in a uniform way. To do so, we define *r-properties* as pairs $\Pi = (\phi, \varphi) \subseteq \Sigma^* \times \Sigma^\omega$. We say that $\sigma \in Exec(\mathcal{P}_\Sigma)$ satisfies (ϕ, φ) (noted $\Pi(\sigma)$) when $\sigma \in \Sigma^* \wedge \phi(\sigma) \vee \sigma \in \Sigma^\omega \wedge \varphi(\sigma)$. The definition of the negation of an *r-property* follows from definition of the negation for finitary and infinitary properties. Boolean combinations of *r-properties* are defined in a natural way. For $* \in \{\vee, \wedge\}$, $(\phi_1, \varphi_1) * (\phi_2, \varphi_2) \stackrel{\text{def}}{=} (\phi_1 * \phi_2, \varphi_1 * \varphi_2)$.

An *r-property* $\Pi \subseteq \Sigma^* \times \Sigma^\omega$ is said to be negatively (resp. positively) determined [11] by $\sigma \in \Sigma^*$ if $\neg\Pi(\sigma) \wedge \forall \mu \in \Sigma^\infty : \neg\Pi(\sigma \cdot \mu)$ (resp. $\Pi(\sigma) \wedge \forall \mu \in \Sigma^\infty : \Pi(\sigma \cdot \mu)$), denoted $\ominus\text{–}determined(\sigma, \Pi)$ (resp. $\oplus\text{–}determined(\sigma, \Pi)$).

3 A Safety-Progress Classification for Runtime Techniques

The *Safety-Progress* (SP) classification of properties [12,9] introduced a hierarchy between regular (linear time) properties[3] defined as sets of *infinite* execution sequences. In [10], we extended the classification to deal also with finite-length execution sequences by revisiting it using runtime properties (*r-properties*). The *Safety-Progress* classification is an alternative to the classical *Safety-Liveness* [13,14] dichotomy. Unlike this later, the *Safety-Progress* classification is a hierarchy and not a partition, and provides a finer-grain classification of properties in a uniform way according to 4 views [15]: a language-theoretic view (seeing properties as sets of sequences), a logical view (seeing properties as LTL formulas), a topological view (seeing properties as open or closed sets), and an automata view (seeing properties as accepted words of Streett automata [16]).

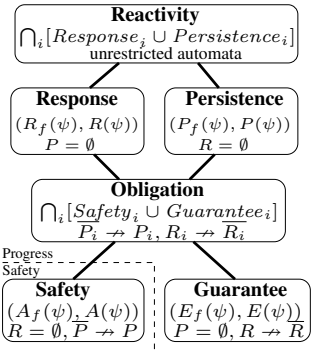


Fig. 1. SP classification

A graphical representation of the *Safety-Progress* classification of properties is depicted in Fig. 1. Further details and results can be found in [17]. Here, we consider only the language and the automata views.

The language-theoretic view of r-properties. The language-theoretic view of the SP classification is based on the construction of infinitary properties and finitary properties from finitary ones. It relies on the use of four operators A, E, R, P (building infinitary properties) and four operators A_f, E_f, R_f, P_f (building finitary properties) applied to finitary properties. Formal definitions can be found in [17]. In the following ψ is a finitary property.

$A(\psi)$ consists of all infinite words σ s.t. *all* prefixes of σ belong to ψ. $E(\psi)$ consists of all infinite words σ s.t. *some* prefixes of σ belong to ψ. $R(\psi)$ consists of all infinite words σ s.t. *infinitely many* prefixes of σ belong to ψ. $P(\psi)$ consists of all infinite words σ s.t. *all but finitely many* prefixes of σ belong to ψ.

$A_f(\psi)$ consists of all finite words σ s.t. *all* prefixes of σ belong to ψ. One can observe that $A_f(\psi)$ is the largest prefix-closed subset of ψ. $E_f(\psi)$ consists of all finite words σ s.t. *some* prefixes of σ belong to ψ. One can observe that $E_f(\psi) = \psi \cdot \Sigma^*$. $R_f(\psi)$ consists of all finite words σ s.t. $\psi(\sigma)$ and there exists an infinite number of continuations σ' of σ also belonging to ψ. $P_f(\psi)$ consists of all finite words σ belonging to ψ s.t. there exists a continuation σ' of σ s.t. σ' persistently has continuations staying in ψ (*i.e.,* σ'' s.t. $\sigma' \cdot \sigma''$ belongs to ψ).

The automata view of r-properties [10]. We define a variant of deterministic and complete Streett automata (introduced in [16] and used in [15]). We add to original Streett automata an acceptance condition for finite sequences in such a way that these automata uniformly recognize *r-properties*.

[3] In the remainder of this paper, the term property will stand for regular property.

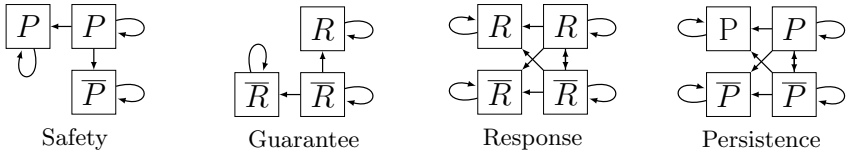


Fig. 2. Schematic illustrations of the shapes of Streett automata for basic classes

Definition 1 (Streett automaton). *A deterministic Streett automaton* $\mathcal{A}$ *is a tuple* $(Q^{\mathcal{A}}, q^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$. *The set* $Q^{\mathcal{A}}$ *is the set of states,* $q^{\mathcal{A}}_{\text{init}} \in Q^{\mathcal{A}}$ *is the initial state.* $\longrightarrow_{\mathcal{A}}$: $Q^{\mathcal{A}} \times \Sigma \to Q^{\mathcal{A}}$ *is the (complete) transition function.* $\{(R_1, P_1), \ldots, (R_m, P_m)\}$ *is the set of accepting pairs, for all* $i \leq n$, $R_i \subseteq Q^{\mathcal{A}}$ *and* $P_i \subseteq Q^{\mathcal{A}}$ *are the sets of recurrent and persistent states.*

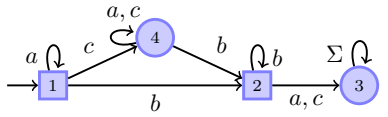
We refer to an automaton with m accepting pairs as an m-automaton. A *plain*-automaton is a 1-automaton, and we refer to R_1 and P_1 as R and P. Moreover, for $\sigma = \sigma_0 \cdots \sigma_{n-1} \in \Sigma^*$ and $q, q' \in Q^{\mathcal{A}}$, we note $q \xrightarrow{\sigma} q'$ when $\exists q_1, \ldots, q_{n-2} \in Q^{\mathcal{A}} : q \xrightarrow{\sigma_0} q_1 \wedge \ldots \wedge q_{n-2} \xrightarrow{\sigma_{n-2}} q'$. For $q \in Q^{\mathcal{A}}$, $Reach_{\mathcal{A}}(q) = \{q' \in Q^{\mathcal{A}} \mid \exists \sigma \in \Sigma^* \setminus \{\epsilon\} : q \xrightarrow{\sigma}_{\mathcal{A}} q'\} \cup \{q\}$ is the set of reachable states from q. For $\sigma \in \Sigma^\infty$, the *run* of σ on $\mathcal{A}$ is the sequence of states involved by the execution of σ on $\mathcal{A}$. It is formally defined as $run(\sigma, \mathcal{A}) = q_0 \cdot q_1 \cdots$ where $\forall i : (q_i \in Q^{\mathcal{A}} \cap Reach_{\mathcal{A}}(q^{\mathcal{A}}_{\text{init}}) \wedge q_i \xrightarrow{\sigma_i}_{\mathcal{A}} q_{i+1}) \wedge q_0 = q^{\mathcal{A}}_{\text{init}}$. For an execution sequence $\sigma \in \Sigma^\omega$ on a Streett automaton $\mathcal{A}$, we define $vinf(\sigma, \mathcal{A})$ as the set of states appearing infinitely often in $run(\sigma, \mathcal{A})$.

Definition 2 (Acceptance conditions). *For* $\sigma \in \Sigma^\omega$, $\mathcal{A}$ *accepts* σ *if* $\forall i \in [1, m] : vinf(\sigma, \mathcal{A}) \cap R_i \neq \emptyset \vee vinf(\sigma, \mathcal{A}) \subseteq P_i$. *For* $\sigma \in \Sigma^*$ *s.t.* $|\sigma| = n$, $\mathcal{A}$ *accepts* σ *if* $(\exists q_0, \ldots, q_{n-1} \in Q^{\mathcal{A}} : run(\sigma, \mathcal{A}) = q_0 \cdots q_{n-1} \wedge q_0 = q^{\mathcal{A}}_{\text{init}}$ *and* $\forall i \in [1, m] : q_{n-1} \in P_i \cup R_i)$. $\mathcal{A}$ defines *an r-property* $(\phi, \varphi) \in 2^{\Sigma^* \times \Sigma^\omega}$ *iff the set of finite (resp. infinite) sequences accepted by* $\mathcal{A}$ *is equal to* ϕ *(resp.* φ*).*

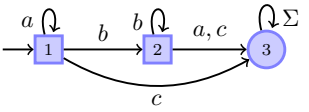
The hierarchy of r-properties. The hierarchical organization of *r-properties* can be seen in the language view using the operators and in the automata view using syntactic restrictions on Streett automata (illustrated in Fig. 2 for basic classes).

Definition 3 (Safety-Progress classes). *An r-property* Π *defined by* $(Q^{\mathcal{A}_\Pi}, q^{\mathcal{A}_\Pi}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$, Π *is said to be*

- *A* safety *r-property if* $\Pi = (A_f(\psi), A(\psi))$ *for some* $\psi \subseteq \Sigma^*$ *or equivalently* $\mathcal{A}_\Pi$ *is a plain-automaton s.t.* $R = \emptyset$ *and there is no transition from* $\overline{P}$ *to* P.
- *A* guarantee *r-property if* $\Pi = (E_f(\psi), E(\psi))$ *for some* $\psi \subseteq \Sigma^*$ *or equivalently* $\mathcal{A}_\Pi$ *is a plain-automaton s.t.* $P = \emptyset$ *and there is no transition from* R *to* $\overline{R}$.
- *An* m-obligation *r-property if* $\Pi = \bigcap_{i=1}^{m}(S_i(\psi_i) \cup G_i(\psi'_i))$ *or* $\Pi = \bigcup_{i=1}^{m}(S_i(\psi_i) \cap G_i(\psi'_i))$ *where* $S(\psi_i)$ *(resp.* $G(\psi'_i)$*) are safety (resp. guarantee) r-properties defined over the* ψ_i *and the* ψ'_i*; or equivalently* $\mathcal{A}_\Pi$ *is an* m*-automaton s.t. for* $i \in [1, m]$ *there is no transition from* $\overline{P_i}$ *to* P_i *and from* R_i *to* $\overline{R_i}$.
- *A* response *r-property if* $\Pi = (R_f(\psi), R(\psi))$ *for some* $\psi \subseteq \Sigma^*$ *or equivalently* $\mathcal{A}_\Pi$ *is a plain-automaton s.t.* $P = \emptyset$.

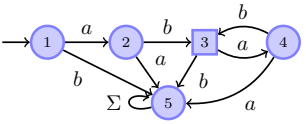


(a) $\psi_1 = a^* \cdot \left(b^* + c \cdot (c + a)^* \cdot b^+\right)$

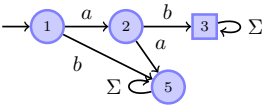


(b) Streett safety, $P = \{1, 2\}$

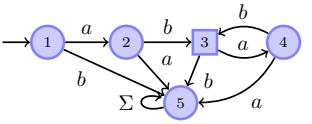
Fig. 3. DFA for ψ_1 and Streett for $(A_f(\psi_1), A(\psi_1))$



(a) $\psi_2 = (a \cdot b)^+$



(b) Streett guarantee



(c) Streett response

Fig. 4. DFA for ψ_2 and Streett for $(E_f(\psi_2), E(\psi_2))$, $(R_f(\psi_2), R(\psi_2))$, $R = \{3\}$

– *A* persistence *r-property if* $\Pi = (P_f(\psi), P(\psi))$ *for some* $\psi \subseteq \Sigma^*$ *or equivalently* $\mathcal{A}_\Pi$ *is a plain-automaton s.t.* $R = \emptyset$.
– *A* reactivity *r-property if* Π *is obtained by finite boolean combinations of response and persistence r-properties or equivalently* $\mathcal{A}_\Pi$ *is unrestricted.*

An r-property of a given class is pure *when not belonging to any other sub-class.*

Example 1 (r-properties). Let us consider $\Sigma_1 = \{a, b, c\}$ and $\psi_1 = a^* \cdot \left(b^* + c \cdot (c + a)^* \cdot b^+\right)$ defined by the deterministic finite-state automaton (DFA) in Fig. 3a with accepting states $1, 2$. The Streett automaton in Fig. 3b defines $(A_f(\psi_1), A(\psi_1))$. Let $\Sigma_2 = \{a, b\}$, and the finitary property $\psi_2 = (a \cdot b)^+$ recognized by the DFA depicted in Fig. 4a. The Streett automaton in Fig. 4b (resp. Fig. 4c) represents the guarantee (resp. response) *r-property* built upon ψ_2.

4 Some Notions of Testability

From its *finite* interaction with the underlying IUT, the tester produces a sequence of events in Σ^*. We study the conditions for a tester, using the produced sequence of events, to determine whether a given relation holds between the set of *all* (finite and infinite) execution sequences that can be produced by the IUT ($Exec(\mathcal{P}_\Sigma)$), and the set of sequences described by the *r-property* Π. Roughly speaking, the challenge addressed by a tester is thus to determine a verdict between Π and $Exec(\mathcal{P}_\Sigma)$, from a finite sequence extracted from $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$[4].

Let us recall that the *r-property* is a pair made of two sets: a set of finite sequences and a set of infinite sequences. In the sequel, we shall compare this pair to the set of execution sequences of the IUT which is a set constituted of finite and infinite sequences. As noticed in [7], one may consider several possible relations between the execution sequences produced by the program and those

[4] Or from a finite set of finite sequences, as a straightforward extension.

described by the property. Those relations are recalled here in the context of *r-properties*. In [1], further relations are studied.

Definition 4 (Relations between IUT sequences and an *r-property* [7]). *The possible relations of interest between $Exec(\mathcal{P}_\Sigma)$ and Π are:*

- $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \subseteq \Pi \cap \Sigma^*$ *and* $Exec_{\omega}(\mathcal{P}_\Sigma) \subseteq \Pi \cap \Sigma^\omega$ *(noted* $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$*).*
- $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \cap (\Pi \cap \Sigma^*) \neq \emptyset$ *and* $Exec_{\omega}(\mathcal{P}_\Sigma) \cap (\Pi \cap \Sigma^\omega) \neq \emptyset$ *(noted* $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$*).*

The test verdict is thus determined according to the conclusions that one can obtain for the considered relation. In essence, a tester can and must only determine a verdict from a *finite interaction* $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$. In Section 5, we will study the conditions to state weaker verdicts on a single execution sequence.

Definition 5 (Verdicts [7]). *Given a relation $\mathcal{R}$ between $Exec(\mathcal{P}_\Sigma)$ and Π and a test execution σ, the tester produces verdicts as follows:*

- *pass if σ allows to determine that $\mathcal{R}$ holds;*
- *fail if σ allows to determine that $\mathcal{R}$ does not hold;*
- *unknown otherwise.*

We note $verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi))$ the verdict that the observation of σ allows to determine. Let us remark the two following practical problems:

- In general, the IUT may be a program exhibiting infinite-length execution sequences. Obviously these sequences cannot be evaluated by a tester wrt. Π.
- Moreover, finite execution sequences contained in the *r-property* cannot be processed easily. For instance, if the test execution exhibits a sequence $\sigma \notin \Pi$, deciding to stop the test is a critical issue. Actually, nothing allows to claim that a continuation of the test execution would not exhibit a new sequence belonging to the *r-property*, *i.e.*, $\sigma' \in \Sigma^\infty$ s.t. $\sigma \cdot \sigma' \in \Pi$.

Thus, the test should be stopped only when there is no doubt regarding the verdict to be established. Following [7], we propose a notion of testability, that takes into account the aforementioned practical limitations, and that is set in the context of the Safety-Progress classification. We suppose the existence of a tester that can interpret the execution sequences with the IUT $\mathcal{P}_\Sigma$ on $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$.

Definition 6 (Testability). *An r-property Π is said to be* testable *on $\mathcal{P}_\Sigma$ wrt. the relation $\mathcal{R}$ if there exists an execution sequence $\sigma \in \Sigma^*$ s.t.:*

$$\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \Rightarrow verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)) \in \{pass, fail\}$$

Intuitively, this condition compels the existence of a sequence which, if played on the IUT, allows to determine for sure, whether the relation holds or not. Let us note that this definition entails to synthesize a test oracle which allows to determine $\mathcal{R}\big(Exec(\mathcal{P}_\Sigma), \Pi\big)$ from the observation of a sequence $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$.

A test oracle is a finite state machine (FSM) parametrized by a test relation as shown in Definition 4. It reads incrementally an interaction sequence $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$ and produces verdicts in $\{pass, fail, unknown\}$.

Definition 7 (Test Oracle). *A* test oracle $\mathcal{O}$ *for an IUT* $\mathcal{P}_\Sigma$*, a relation* $\mathcal{R}$ *and an r-property* Π *is a 4-tuple* $(Q^{\mathcal{O}}, q^{\mathcal{O}}_{\text{init}}, \longrightarrow_{\mathcal{O}}, \Gamma^{\mathcal{O}})$*. The finite set* $Q^{\mathcal{O}}$ *denotes the control states and* $q^{\mathcal{O}}_{\text{init}} \in Q^{\mathcal{O}}$ *is the initial state. The complete function* $\longrightarrow_{\mathcal{O}}$*:* $Q^{\mathcal{O}} \times \Sigma \to Q^{\mathcal{O}}$ *is the transition function. The output function* $\Gamma^{\mathcal{O}} : Q^{\mathcal{O}} \to \{pass, fail, unknown\}$ *produces verdicts with the following constraints:*

- *all states emitting a pass or a fail verdict are final (sink states),*
- $\exists \sigma \in Exec_{\text{f}}(\mathcal{P}_\Sigma) : q^{\mathcal{O}}_{\text{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \wedge \Gamma(q) = pass \Rightarrow \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)$,
- $\exists \sigma \in Exec_{\text{f}}(\mathcal{P}_\Sigma) : q^{\mathcal{O}}_{\text{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \wedge \Gamma(q) = fail \Rightarrow \neg\mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)$.

5 Testable Properties without Executable Specification

The framework of *r-properties* (Section 3) allows to determine the testability of the different classes of properties using positive and negative determinacy. Moreover, this framework provides a computable oracle, which is a sufficient condition for testing. Furthermore, we will be able to characterize which test sequences allow to establish sought verdicts. Then, we will determine which verdict has to be produced in accordance with the played test sequence.

In this paper, we focus on the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. Characterizations for the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ (by duality) and others relations are in [1].

Obtainable verdicts and sufficient conditions. For this relation, the unique verdicts that may be produced are *fail* and *unknown*. We explicit this below.

A *pass* verdict means that all execution sequences of $\mathcal{P}_\Sigma$ belong to Π. The unique case where it is possible to establish a *pass* verdict is in the trivial case where $\Pi = (\Sigma^*, \Sigma^\omega)$, *i.e.,* the *r-property* Π is always verified. Obviously, every implementation with alphabet Σ satisfies this relation. In other cases, it is impossible to obtain such a verdict (whatever is the property class under consideration), since the whole set $\mathcal{P}_\Sigma$ is usually unknown from the tester. In Section 5, we will study the conditions under which it is possible to state *weak pass* verdicts, when reasoning on a *single* execution sequence of the IUT.

A *fail* verdict means that there exists some sequences of the program which are not in Π. In order to produce this verdict, a sufficient condition is to exhibit an execution sequence of $\mathcal{P}_\Sigma$ s.t. Π is *negatively determined* by this sequence:

$$\exists \sigma \in Exec_{\text{f}}(\mathcal{P}_\Sigma) : \ominus\text{–}determined(\sigma, \Pi) \;\Rightarrow\; verdict(\sigma, Exec(\mathcal{P}_\Sigma) \subseteq \Pi) = fail$$

Testability of this relation in the Safety-Progress classification. For each SP class, we state the conditions under which the properties of this class are testable.

Theorem 1 (Testability of $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$**).** *For* $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q^{\mathcal{A}_\Pi}_{\text{init}}, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ *recognizing an r-property* Π*, according to the class of* Π*, the testability conditions expressed both in the language-theoretic and automata views are given in Table 1.*

Table 1. Summary of testability results wrt. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$

$Exec(\mathcal{P}_\Sigma) \subseteq \Pi$	Testability Condition (language view)	Testability Condition (automata view)
Safety $(A_f(\psi), A(\psi))\ \|R = \emptyset, \overline{P} \nrightarrow P$	$\overline{\psi} \neq \emptyset$	$\overline{P} \neq \emptyset$
Guarantee $(E_f(\psi), E(\psi))\|P = \emptyset, R \nrightarrow \overline{R}$	$\{\sigma \in \overline{\psi} \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$	$\{q \in \overline{R} \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \overline{R}\} \neq \emptyset$
Obligation $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi'_i)) \bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi'_i))$ $\overline{P_i} \nrightarrow P_i, R_i \nrightarrow \overline{R_i}$	$\bigcup_{i=1}^{k}(\overline{\psi_i} \cap \{\sigma \in \overline{\psi'_i} \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \overline{\psi'_i}\}) \neq \emptyset$ $\bigcap_{i=1}^{k}(\overline{\psi_i} \cup \{\sigma \in \overline{\psi'_i} \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \overline{\psi_i}\}) \neq \emptyset$	$\bigcup_{i=1}^{k}(\overline{P_i} \cap \{q \in \overline{R_i} \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \overline{R_i}\}) \neq \emptyset$
Response $(R_f(\psi), R(\psi))\ \|P = \emptyset$	$\{\sigma \in \overline{\psi} \mid cont^*(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$	$\{q \in \overline{R} \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \overline{R}\} \neq \emptyset$
Persistence $(P_f(\psi), P(\psi))\ \|R = \emptyset$	$\{\sigma \in \overline{\psi} \mid cont^*(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$	$\{q \in \overline{P} \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \overline{P}\} \neq \emptyset$

Verdicts to deliver. We now state the verdicts that should be produced by a tester for the possibly infinite sequences of the IUT. Each testability condition in the language view is in the form $f(\{\psi_i\}_i) \neq \emptyset$ where the $\psi_i \subseteq \Sigma^*$ ($i \in [1, n]$) are used to build the *r-property* and f is a composition of set operations on ψ_i. When $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \cap f(\{\psi_i\}_i)$, the test oracle should deliver *fail* since the underlying *r-property* is negatively determined. Conversely, when $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \setminus f(\{\psi_i\}_i)$, the test oracle can deliver *unknown*. In practice, those verdicts are determined by a computable function, reading an interaction sequence, *i.e.,* a test oracle. In our framework, the test oracle is obtained from a Streett automaton[5]:

Property 1 (Test oracle for the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$). Given $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q^{\mathcal{A}_\Pi}_{\mathrm{init}}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ defining Π, the test oracle $(Q^{\mathcal{O}}, q^{\mathcal{O}}_{\mathrm{init}}, \longrightarrow_{\mathcal{O}}, \Gamma^{\mathcal{O}})$ for the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ is defined as follows. $Q^{\mathcal{O}}$ is the smallest subset of $Q^{\mathcal{A}_\Pi}$, reachable from $q^{\mathcal{O}}_{\mathrm{init}}$ by $\longrightarrow_{\mathcal{O}}$ (defined below) with $q^{\mathcal{O}}_{\mathrm{init}} = q^{\mathcal{A}_\Pi}_{\mathrm{init}}$.

- $\Gamma^{\mathcal{O}}$ is defined as follows:
  - If Π is a pure safety, guarantee, obligation, or response property $\Gamma^{\mathcal{O}}(q) =$ *fail* if $q \in \bigcup_{i=1}^{k}(\overline{P_i} \cap \{q \in \overline{R_i} \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \overline{R_i}\}$ and *unknown* otherwise,
  - If Π is a pure persistence property $\Gamma^{\mathcal{O}}(q) =$ *fail* if $q \in \{q \in \overline{P} \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \overline{P}\}$ and *unknown* otherwise;
- $\longrightarrow_{\mathcal{O}}$ is defined as the smallest relation verifying:
  - $q \xrightarrow{e}_{\mathcal{O}} q$ if $\exists e \in \Sigma, \exists q' \in Q^{\mathcal{O}} : q \xrightarrow{e}_{\mathcal{A}_\Pi} q'$ and $\Gamma^{\mathcal{O}}(q) =$ *fail*,
  - $\longrightarrow_{\mathcal{O}} = \longrightarrow_{\mathcal{A}_\Pi}$ otherwise.

The proof of this property follows from Theorem 1 and Definition 7.

Example 2 (Testability of some r-properties wrt. $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$). We present the testability of three r-properties introduced in Example 1. The safety r-property Π_1 is testable wrt. the relation $Exec(\mathcal{P}_{\Sigma_1}) \subseteq \Pi_1$. Indeed in the language view, there are sequences belonging to $\overline{\psi_1}$ (the corresponding DFA has a non accepting state). In the automata view, we have sink $\in \overline{P}$ (reachable from the initial state). The guarantee r-property Π_2 is testable wrt. the relation $Exec(\mathcal{P}_{\Sigma_2}) \subseteq \Pi_2$.

[5] The test oracle can be also obtained from the *r-properties* described in others views (language, logic). Indeed, in [17] we describe how to express an *r-property* in the automata view from its expression in the language or the logic view.

Indeed, there are sequences belonging to $\overline{\psi_2}$ *s.t. all prefixes of theses sequences and all its continuations are also in* $\overline{\psi_2}$. *In the automata view, there is a (reachable) state in* $\overline{R}$ *from which all reachable states are in* $\overline{R}$. *The response r-property* Π_3 *is testable wrt. the relation* $Exec(\mathcal{P}_{\Sigma_2}) \subseteq \Pi_3$. *Indeed, there are sequences belonging to* $\overline{\psi_2}$ *s.t. all continuations of these sequences belong to* $\overline{\psi_2}$. *In the automata view, there is a (reachable) state in* $\overline{R}$ *from which all reachable states are in* $\overline{R}$. Thus, we have clarified and extended some results of [7]. First, we have shown that the safety *r-property* $(\Sigma^*, \Sigma^\omega)$ always lead to a *pass* verdict and is vacuously testable. Moreover, we exhibited some *r-properties* of other classes which are testable, *i.e.,* some obligation, response, and persistence *r-properties*.

Refining verdicts. Similarly to the introduction of weak truth values in runtime verification [18,10,17], it is possible to introduce *weak* verdicts in testing. In this respect, stopping the test and producing a weak verdict consists in stating that the test interaction sequence produced so far belongs (or not) to the property. The idea of satisfaction "if the program stops here" in runtime verification [18,10] corresponds to the idea of "the test has shown enough on the implementation" in testing. In this case, testing would be similar to a kind of "active runtime verification": one is interested in the satisfaction of one execution of the program which is steered externally by a tester. Basically, it amounts to not seeing testing as a destructive activity, but as a way to enhance confidence in the implementation compliance wrt. a property.

Under some conditions, it is possible to determine *weak verdicts* for some classes of properties in the following sense: the verdict is expressed on *one single execution sequence* σ, and it does not afford any conclusion on the set $Exec(\mathcal{P}_\Sigma)$.

We have seen that, for $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$, the only verdicts that can be produced were *fail* and *unknown*. Clearly, *fail* verdicts can still be produced. Furthermore, *unknown* verdicts can be refined into weak *pass* verdicts when the sequence σ *positively determines* the *r-property*. In this case, the test can be stopped since whatever is the future behavior of the IUT, it will exhibit behaviors that will satisfy the *r-property*. In this case, it seems reasonable to produce a weak pass verdict and consider new test executions in order to gain in confidence.

We revisit, for each *Safety-Progress* class, the situations when weak *pass* verdicts can be produced for this relation.

For safety *r-properties*. Let Π be a safety *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(A_f(\psi), A(\psi))$. When the produced sequence belongs to $\{\sigma \in \psi \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi\}$, the tester can produce a weak *pass* verdict.

For guarantee *r-properties*. Let Π be a guarantee *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(E_f(\psi), E(\psi))$. It is possible to produce a weak *pass* verdict if the set ψ is not empty: guarantee *r-properties* are always positively determined when they are satisfied.

For obligation *r-properties*. Let Π be an m-obligation *r-property*.

- If for $m \in \mathbb{N}^*$, Π is expressed $\bigcap_{i=1}^{m}(S_i(\psi_i) \cup G_i(\psi'_i))$ where $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) is a safety (resp. guarantee) *r-property* built upon ψ_i (resp. ψ'_i), $i \in [1, m]$. The tester can produce a weak *pass* verdict when the interaction sequence belongs to $\bigcap_{i=1}^{m} \psi'_i$.

- If for $m \in \mathbb{N}^*$, Π is expressed $\bigcup_{i=1}^{m}(S_i(\psi_i) \cap G_i(\psi_i'))$ where $S_i(\psi_i)$ (resp. $G_i(\psi_i')$) is a safety (resp. guarantee) *r-property* built upon ψ_i (resp. ψ_i'), $i \in [1, m]$. The tester can produce a weak *pass* verdict when the interaction sequence produced by the program belongs to $\bigcup_{i=1}^{m}(\{\sigma \in \psi_i \mid \mathit{pref}(\sigma) \cup \mathit{cont}^*(\sigma) \subseteq \psi_i\} \cap \psi_i')$.

For response and persistence *r-properties*. The reasoning is similar to the one used for safety *r-properties*. Let Π be a response (resp. persistence) *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(R_f(\psi), R(\psi))$ (resp. $(P_f(\psi), P(\psi))$). When the interaction sequence belongs to $\{\sigma \in \psi \mid \mathit{pref}(\sigma) \cup \mathit{cont}^*(\sigma) \subseteq \psi\}$, the tester can produce a weak *pass* verdict.

6 Automatic Test Generation

In this section, we address test generation for the testing framework introduced in this paper. Here, test generation is based on *r-properties*, and the purpose of the test campaign is to detect verdicts for a relation between an *r-property* and an IUT. Before entering into the details of test generation, we first discuss informally some practical constraints that have to be taken into account for test generation. After that, we are able to compute the canonical tester, discuss test selection, and show how quiescence can be taken into account in our framework.

Which sequences should be played? The sequences of interest to play on the IUT are naturally those leading to a *fail* or a *weak pass* verdict and these can be used to generate test cases. In the language view (resp. automata view), these sequences are those belonging to the exhibited sets (resp. leading to the exhibited set of states) in testability conditions. For instance, for a safety *r-property* $\Pi_S = (A_f(\psi), A(\psi))$ built upon ψ, and defined by a safety automaton $\mathcal{A}_{\Pi_S}$, one should play sequences in $\overline{\psi}$ or equivalently those leading to $\overline{P}$ in $\mathcal{A}_{\Pi_S}$.

When to stop the test? When the tested program produces an execution sequence $\sigma \in \Sigma^*$, a raised question is when to safely stop the test. Obviously, a first answer is when a *fail* or *weak pass* verdict has been issued since this verdict is definitive. Although in other cases, when the test interactions produced some test sequences leading so far to *unknown* evaluations, the question prevails. It remains to the tester appraisal to decide when the test should be stopped (see Section 6.2).

Vocabularies and test architecture. In order to address test generation, we will need to distinguish inputs and outputs and the vocabularies of the IUT and the *r-property*. The alphabet Σ of the property is now partitioned into $\Sigma_?$ (input actions) and $\Sigma_!$ (output actions). The alphabet of the IUT becomes Σ^{IUT} and is partitioned into $\Sigma_?^{IUT}$ (input actions) and $\Sigma_!^{IUT}$ (output actions) with $\Sigma_? = \Sigma_?^{IUT}$ and $\Sigma_! = \Sigma_!^{IUT}$. As usual, we also suppose that the behavior of the IUT can be modeled by an IOLTS $\mathcal{I} = (Q^{\mathcal{I}}, q_{\text{init}}^{\mathcal{I}}, \Sigma^{IUT}, \longrightarrow_{\mathcal{I}})$.

6.1 Computation of the Canonical Tester

We adapt the classical construction of the canonical tester for our framework. The canonical tester that we build for a relation $\mathcal{R}$ between an IUT $\mathcal{P}_\Sigma$ and

a *r-property* Π is purposed to detect all verdicts for the relation between the *r-property* and all possible interactions that can be produced with $\mathcal{P}_\Sigma$.

We define canonical testers from Streett automata. To do so, we will use a set of subsets of Streett automaton states that we introduced in [10] for runtime verification. For a Streett automaton $\mathcal{A}_\Pi$, the sets $G^{\mathcal{A}_\Pi}, G_c^{\mathcal{A}_\Pi}, B_c^{\mathcal{A}_\Pi}, B^{\mathcal{A}_\Pi}$ form a partition of $Q^{\mathcal{A}_\Pi}$ and designate respectively the good (resp. currently good, currently bad, bad) states:

- $G^{\mathcal{A}_\Pi} = \{q \in Q^{\mathcal{A}_\Pi} \cap \bigcap_{i=1}^{m}(R_i \cup P_i) \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \bigcap_{i=1}^{m}(R_i \cup P_i)\}$
- $G_c^{\mathcal{A}_\Pi} = \{q \in Q^{\mathcal{A}_\Pi} \cap \bigcap_{i=1}^{m}(R_i \cup P_i) \mid Reach_{\mathcal{A}_\Pi}(q) \not\subseteq \bigcap_{i=1}^{m}(R_i \cup P_i)\}$
- $B_c^{\mathcal{A}_\Pi} = \{q \in Q^{\mathcal{A}_\Pi} \cap \bigcup_{i=1}^{m}(\overline{R_i} \cap \overline{P_i}) \mid Reach_{\mathcal{A}_\Pi}(q) \not\subseteq \bigcup_{i=1}^{m}(\overline{R_i} \cap \overline{P_i})\}$
- $B^{\mathcal{A}_\Pi} = \{q \in Q^{\mathcal{A}_\Pi} \cap \bigcup_{i=1}^{m}(\overline{R_i} \cap \overline{P_i}) \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \bigcup_{i=1}^{m}(\overline{R_i} \cap \overline{P_i})\}$

It is possible to show [10] that if a sequence σ reaches a state in $B^{\mathcal{A}_\Pi}$ (resp. $G^{\mathcal{A}_\Pi}$), then the underlying property Π is negatively (resp. positively) determined by σ.

The canonical tester is defined as follows.

Definition 8 (Canonical Tester). *From a Streett automaton* $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}^{\mathcal{A}_\Pi}, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ *defining a* testable *r-property* Π*, the canonical tester is the IOLTS* $T = (Q^T, q_{\text{init}}^T, \Sigma, \longrightarrow_T)$ *defined as follows:*

- $Q^T = B_c^{\mathcal{A}_\Pi} \cup G_c^{\mathcal{A}_\Pi} \cup \{Fail\} \cup \{WeakPass\}$ *with* $q_{\text{init}}^T = q_{\text{init}}^{\mathcal{A}_\Pi}$*;*
- $\longrightarrow_T$ *is defined as follows:*
  - $\forall e \in \Sigma : Fail \xrightarrow{e}_T Fail \wedge WeakPass \xrightarrow{e}_T WeakPass$,
  - $q \xrightarrow{e}_T Fail$ *if* $q \xrightarrow{e}_{\mathcal{A}_\Pi} q' \wedge q' \in B^{\mathcal{A}_\Pi}$*, for any* $e \in \Sigma$,
  - $q \xrightarrow{e}_T WeakPass$ *if* $q \xrightarrow{e}_{\mathcal{A}_\Pi} q' \wedge q' \in G^{\mathcal{A}_\Pi}$*, for any* $e \in \Sigma$,
  - $q \xrightarrow{e}_T q'$ *if* $q \xrightarrow{e}_{\mathcal{A}_\Pi} q' \wedge q, q' \in G_c^{\mathcal{A}_\Pi} \cup B_c^{\mathcal{A}_\Pi}$*, for any* $e \in \Sigma$.

A Streett automaton is transformed as follows. Transitions leading to a bad (resp. good) state are redirected to *Fail* (resp. *WeakPass*). Those latest states are terminal: the test can be stopped and the verdict produced.

6.2 Test Selection

For a given *r-property*, the set of potential sequences to be played is infinite. In practice, one may use the underlying Streett automaton to constrain the states that should be visited during a test. Furthermore, as usual, one needs to select a test case that is *controllable* [3]. It can be done on the canonical tester by first disabling input actions that do not permit to reach sought verdicts. Second, for a state in which several input actions are possible, one needs to generate different test cases with one input per state. More details can be found in [1].

Test selection plays also a role to state *weak pass* verdicts. Indeed, when dealing with sequences satisfying a *r-property so far* and not positively determining it, test selection should plan the moment for stopping the test. It can be, for instance, when the test lasted more than a given expected duration or when the number of interactions with the IUT is greater or equal than an expected number. However, one should not forget that there might exist a continuation, that can be produced by letting the test execution continue, not satisfying the *r-property* or even negatively determining it. Here, it thus remains to the tester expertise to state the halting criterion (possibly using quiescence, see Section 6.3).

6.3 Introducing Quiescence

Quiescence [19,3] was introduced in conformance testing in order to represent IUT's inactivity. In practice, several kinds of quiescence may happen (see [3] for instance). Here we distinguish two kinds of quiescence. Outputlocks (denoted δ_o) represent the situations where the IUT is waiting for an input and produces no outputs. Deadlocks (denoted δ_d) represent the situations where the IUT cannot interact anymore, *e.g.*, its execution is terminated or it is deadlocked. Thus, we introduce those two events in the output alphabet of the IUT. We have now the following additional alphabets: $\Sigma^{IUT}_{!,\delta} = \Sigma^{IUT}_{!} \cup \{\delta_o, \delta_d\}$, $\Sigma^{IUT}_{\delta} = \Sigma^{IUT}_{!,\delta} \cup \Sigma^{IUT}_{?}$.

We also have to distinguish the set of traces of the IUT from the set of potential interactions with the IUT. This latest is based on the observable behavior of the IUT and potential choices of the tester. The set of executions of the IUT is now $Exec(\mathcal{P}_{\Sigma^{IUT}}) \subseteq (\Sigma^{IUT}_{\delta})^{\infty}$. The set of interactions of the tester with the IUT is $Inter(\Sigma^{IUT}) \subseteq (\Sigma^{IUT} + \delta_o)^* \cdot (\delta_d + \epsilon)$, *i.e.,* the tester can observe IUT's outputlocks and finishes by the observation of a deadlock or program termination. When considering quiescence, characterizing testable properties now consists in comparing the set of interactions to the set of sequences described by the *r-property*. The intuitive ideas are the following:

- the tester can observe self-terminated executions of the IUT with δ_d,
- the tester can decide to terminate the program when observing an outputlock.

The notion of negative determinacy is now modified in the context of quiescence as follows. We say that the *r-property* Π is negatively determined upon quiescence by the sequence $\sigma \in Inter(\mathcal{P}_{\Sigma^{IUT}})$ (denoted $\ominus\text{–}determined\text{–}q(\sigma, \Pi)$) if $\ominus\text{–}determined(\sigma_{\downarrow \Sigma^{IUT}}, \Pi) \vee (|\sigma| > 1 \wedge last(\sigma) \in \{\delta_d, \delta_o\} \wedge \neg \Pi((\sigma_{\ldots|\sigma|-2})_{\downarrow \Sigma^{IUT}})$, where $\sigma_{\downarrow \Sigma^{IUT}}$ is the projection of σ on Σ^{IUT}.

For the proposed approach, the usefulness of quiescence lies in the fact that the current test sequence does not have any continuation. Consequently, testability conditions may be weakened. Indeed, when one has determined that the current interaction with the IUT is over, it is not necessary that the *r-property* should be evaluated in the same way. In some sense, it amounts to consider that the evaluation produced by the last event before observing quiescence "terminates" the execution sequence. Thus, if the *r-property* is not satisfied by the last observed sequence, then the *r-property* is negatively determined by it.

Revisiting previous results. With quiescence, the purpose of the tester is now to "drive" the IUT in a state in which the underlying *r-property* is not satisfied,

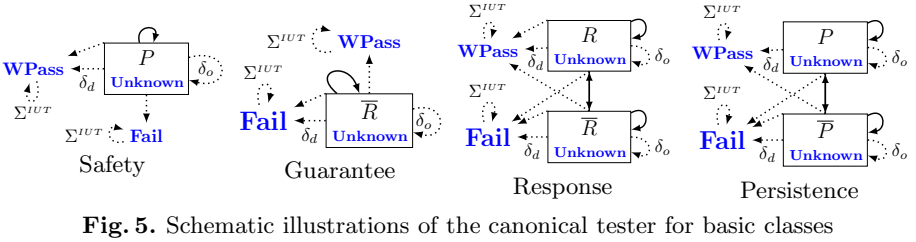


Fig. 5. Schematic illustrations of the canonical tester for basic classes

Table 2. Testability wrt. $Inter(\mathcal{P}_{\Sigma^{IUT}}) \subseteq \Pi$ with quiescence

$Exec(\mathcal{P}_\Sigma) \subseteq \Pi$	Possible Verdicts	Testability Condition
Safety $(A_f(\psi), A(\psi))$	*fail*, *unknown*	$\overline{\psi} \neq \emptyset$
Guarantee $(E_f(\psi), E(\psi))$	*fail*, *unknown*	$\{\sigma \in \overline{\psi} \mid pref(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$
Obligation $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi_i'))$ $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi_i'))$	*fail*, *unknown*	$\bigcup_{i=1}^{k}(\overline{\psi_i} \cap \{\sigma \in \overline{\psi_i'} \mid pref(\sigma) \subseteq \overline{\psi_i'}\} \neq \emptyset$ $\bigcap_{i=1}^{k}(\overline{\psi_i} \cup \{\sigma \in \overline{\psi_i'} \mid pref(\sigma) \subseteq \overline{\psi_i'}\} \neq \emptyset$
Response $(R_f(\psi), R(\psi))$	*fail*, *unknown*	$\overline{\psi} \neq \emptyset$
Persistence $(P_f(\psi), P(\psi))$	*fail*, *unknown*	$\overline{\psi} \neq \emptyset$

and then observe quiescence. Informally, the testability condition relies now on the existence of a sequence s.t. the *r-property* is not satisfied. Testability results, upon the observation of quiescence and in order to produce *fail* verdicts when the tested *r-property* is not satisfied, are updated using the notion of negative determinacy with quiescence as shown in Table 2.

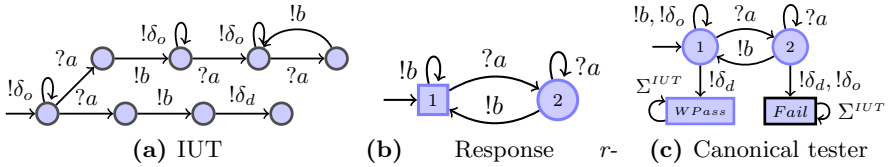
The canonical tester construction is also updated by adding the following rules for $\longrightarrow_T$: $\forall q \in B_c^{\mathcal{A}_\Pi} : q \xrightarrow{\delta_o,\delta_d}_T Fail$, $\forall q \in G_c^{\mathcal{A}_\Pi} : q \xrightarrow{\delta_o}_T q \wedge q \xrightarrow{\delta_d}_T WeakPass$. Illustrations of the construction of the canonical tester for basic classes with quiescence is given in Fig. 5, where the original (resp. modified) transitions from the Streett automaton are in plain (resp. dotted) lines.

Example 3 (Testability with quiescence). We illustrate the usefulness of quiescence. Consider the IUT depicted in Fig. 6a with observable actions $\Sigma_?^{IUT} = \{?a\}$ *and* $\Sigma_!^{IUT} = \{!b\}$. *This IUT waits for an* $?a$, *produces a* $!b$, *and then non deterministically terminates or waits for an* $?a$, *and repeats the behavior consisting in receiving an* $?a$ *and producing a* $!b$. *The executions and possible interactions with the tester are ("?" and "!" are not represented and* $x^{\&}$ *stands for* $x + \epsilon$*):*

$$Exec(\mathcal{P}_{\Sigma^{IUT}}) = {\delta_o}^{\&} \cdot \left(a^{\&} + a \cdot b \cdot (\delta_d + {\delta_o}^{\&}) \cdot (a \cdot \left[{\delta_o}^{\&} \cdot ((a \cdot b)^{\&})^*\right]^* \cdot a^{\&})^{\&}\right)$$

$$Inter(\mathcal{P}_{\Sigma^{IUT}}) = {\delta_o}^{\&} \cdot \left((a \cdot b)^{\&} \cdot (\delta_d + {\delta_o}^{\&}) \cdot (a \cdot \left[{\delta_o}^{\&} \cdot ((a \cdot b)^{\&})^*\right]^* \cdot {\delta_o}^{\&})^{\&}\right)^{\&}$$

Now let us consider the r-property defined by the Streett automaton depicted in Fig. 6b. Its vocabulary is $\{?a, !b\}$, *and it has one recurrent state:* $R = \{1\}$. *The underlying r-property states that every input* $?a$ *should be acknowledged by an output* $!b$. *Though being not testable under the conditions expressed in Section 5, this r-property is testable with quiescence. One can observe that* $Inter(\mathcal{P}_{\Sigma^{IUT}}) \not\subseteq \Pi$



(a) IUT **(b)** Response *r-property* **(c)** Canonical tester

Fig. 6. Illustrating the usefulness of quiescence

because the existence of $?a{\cdot}!b{\cdot}?a{\cdot}!\delta_o$ *in* $Inter(\mathcal{P}_{\Sigma^{IUT}})$*. The synthesized canonical tester is depicted in Fig. 6c.*

7 Related Work and Discussion

In this section we overview related work or work that may be leveraged by the results proposed in this paper. Then, we propose a discussion on the results afforded by this paper. A deeper treatment of related work is provided in [1].

Testing oriented by properties for generating test purposes. One of the limits of conformance testing [19] lies in the size of the generated test suite which can be infinite or impracticable. Some testing approaches oriented by properties were proposed to face off this limitation by focusing on critical properties. In this case, properties are used as a complement to the specification in order to generate test purposes which will be then used to conduct and select test cases [3,20]. For a presentation of some general approaches, the reader is referred to [21].

Combining testing and formal verification. In [6], the complementarity between verification techniques and conformance testing is studied. Notably, the authors shown that it is possible to detect (using testing) violations of safety (resp. satisfaction of co-safety) properties on the implementation and the specification.

Requirement-Based testing. In requirement-based testing, the purpose is to generate a test suite from a set of informal requirements. For instance, in [22,23], test cases are generated from LTL formula using a model-checker. Those approaches were interested in defining a syntactic test coverage for the tested requirements.

Property testing without a behavioral specification. In previous approaches, we used the notion of tiles which are elementary test modules testing specific parts of an implementation and which can be combined to test more complex behaviors using a property (see [24,25]).

Using the Safety-Progress classification in validation techniques. The *Safety-Progress* classification of properties is rarely used in validation techniques. We used (*e.g.*, [10]) the *Safety-Progress* classification to characterize the sets of properties that can be verified and enforced during the runtime of systems. In some sense, this previous endeavor similarly addressed the expressiveness question for runtime verification and runtime enforcement.

Discussion. Several approaches fall in the scope of the generic one proposed in this paper. For instance, our results apply and extend the approach where verification is combined to testing as proposed in [6]. Furthermore, this approach leverages the use of test purposes [2,3] in testing to guide test selection. Indeed, the characterization of testable properties gives assets on the kind of test purposes that can be used in testing. Moreover, the properties considered in this paper are framed into the *Safety-Progress* classification of properties [12,9] which is equivalently a hierarchy of regular properties. Thus the results proposed by this paper concern previous depicted approaches in which the properties at stake can be formalized by a regular

language. Furthermore, classical conformance testing falls in the scope of the proposed framework. Indeed, suspended traces of an implementation preserving the *ioco* relation wrt. a given specification can be expressed as a safety property [6].

8 Conclusion and Perspectives

Conclusion. In this paper, we study the space of testable properties. We use a testability notion depending on a relation between the set of execution sequences that can be produced by the underlying implementation and the *r-property*. Leveraging the notions of positive and negative determinacy of properties, we have identified for each *Safety-Progress* class and according to the relation of interest, the testable fragment. Moreover we have seen that the framework of *r-properties* in the *Safety-Progress* classification provides a decidable test oracle in order to produce a verdict depending on the interaction between the tester and the IUT. Furthermore, we also propose some conditions under which it makes sense for a tester to state weak verdicts. Finally, results of this paper are implemented in an available prototype tool for which a description is given in [1].

Perspectives. A first research direction is to investigate the set of testable properties for more expressive formalisms. Indeed, the *Safety-Progress* classification is concerned with regular properties, and classifying testable properties for *e.g.*, context-free properties would be of interest. Another perspective is to combine the approach proposed with weak verdicts to a notion of *test coverage*. Indeed, in order to bring any confidence in the fact that *e.g.*, the implementation respects the property, it involves to execute the test several times to make it relevant. The various approaches [22,23] for defining test coverage for property-oriented testing could be used to reinforce a set of weak verdicts.

References

1. Falcone, Y., Fernandez, J.C., Jéron, T., Marchand, H., Mounier, L.: More Testable Properties. Technical Report 7279, INRIA (2010)
2. Koch, B., Grabowski, J., Hogrefe, D., Schmitt, M.: Autolink: A Tool for Automatic Test Generation from SDL Specifications. In: Industrial-Strength Formal Specification Techniques (1998)
3. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer (STTT), 297–315 (2005)
4. Traon, Y.L., Mouelhi, T., Baudry, B.: Testing Security Policies: Going Beyond Functional Testing. In: Int. Symp. on Software Reliability Engineering, pp. 93–102 (2007)
5. Mallouli, W., Orset, J.M., Cavalli, A., Cuppens, N., Cuppens, F.: A Formal Approach for Testing Security Rules. In: SACMAT 2007: Proceedings of the 12th ACM symposium on Access control models and technologies, pp. 127–132. ACM, New York (2007)
6. Constant, C., Jéron, T., Marchand, H., Rusu, V.: Integrating Formal Verification and Conformance Testing for Reactive Systems. IEEE Trans. Software Eng. 33, 558–574 (2007)
7. Nahm, R., Grabowski, J., Hogrefe, D.: Test Case Generation for Temporal Properties. Technical report, Bern University (1993)

8. Grabowski, J.: SDL and MSC based test case generation– an overall view of the SAMSTAG method. Technical report, University of Berne IAM-94-0005 (1994)
9. Chang, E., Manna, Z., Pnueli, A.: Characterization of Temporal Property Classes. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 474–486. Springer, Heidelberg (1992)
10. Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime Verification of Safety-Progress Properties. In: The 9th Int. Workshop on Runtime Verification, pp. 40–59 (2009)
11. Pnueli, A., Zaks, A.: PSL Model Checking and Run-Time Verification Via Testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)
12. Manna, Z., Pnueli, A.: A Hierarchy of Temporal Properties (invited paper 1989). In: PODC 1990: Proceedings of The 9th symp. on Principles Of Distributed Computing, pp. 377–410. ACM, New York (1990)
13. Lamport, L.: Proving the Correctness of Multiprocess Programs. IEEE Trans. Softw. Eng., 125–143 (1977)
14. Alpern, B., Schneider, F.B.: Defining Liveness. Technical report, Cornell University, Ithaca, NY, USA (1984)
15. Chang, E., Manna, Z., Pnueli, A.: The Safety-Progress Classification. Technical report, Stanford University, Dept. of Computer Science (1992)
16. Streett, R.S.: Propositional Dynamic Logic of looping and converse. In: STOC 1981: Proceedings of the 13th Symp. on Theory Of computing, pp. 375–383. ACM, New York (1981)
17. Falcone, Y., Fernandez, J.C., Mounier, L.: What can You Verify and Enforce at Runtime? Technical Report TR-2010-5, Verimag Research Report (2010)
18. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL Semantics for Runtime Verification. Journal of Logic and Computation (2009)
19. Tretmans, J.: Test Generation with Inputs, Outputs, and Quiescence. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. Tretmans, J, vol. 1055, pp. 127–146. Springer, Heidelberg (1996)
20. de Vries, R.G.: Towards formal test purposes. In: FATES 2001: Formal Approaches to Testing of Software, pp. 61–76 (2001)
21. Machado, P.D.L., Silva, D.A., Mota, A.C.: Towards Property Oriented Testing. Electron. Notes Theor. Comput. Sci., 3–19 (2007)
22. Rajan, A., Whalen, M., Heimdahl, M.: Model Validation using Automatically Generated Requirements-Based Tests. In: HASE 2007: 10th IEEE Symposium on High Assurance Systems Engineering, pp. 95–104 (November 2007)
23. Pecheur, C., Raimondi, F., Brat, G.: A Formal Analysis of Requirements-based Testing. In: ISSTA 2009: Proceedings of the 18th International Symposium on Software Testing and Analysis, pp. 47–56. ACM, New York (2009)
24. Darmaillacq, V., Fernandez, J.C., Groz, R., Mounier, L., Richier, J.L.: Test Generation for Network Security Rules. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 341–356. Springer, Heidelberg (2006)
25. Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A Compositional Testing Framework Driven by Partial Specifications. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 107–122. Springer, Heidelberg (2007)

Alternating Simulation and IOCO

Margus Veanes and Nikolaj Bjørner

Microsoft Research, Redmond, WA, USA
{margus,nbjorner}@microsoft.com

Abstract. We propose a symbolic framework called *guarded labeled assignment systems* or GLASs and show how GLASs can be used as a foundation for symbolic analysis of various aspects of formal specification languages. We define a notion of i/o-refinement over GLASs as an alternating simulation relation and provide formal proofs that relate i/o-refinement to ioco. We show that non-i/o-refinement reduces to a reachability problem and provide a translation from bounded non-i/o-refinement or bounded non-ioco to checking first-order assertions.

1 Introduction

The view of a system behavior as a labeled transition system (LTS) provides the semantical foundation for many behavioral aspects of systems in the context of formal verification and testing. The central problem in testing is to determine if an implementation LTS *conforms* to a given specification LTS and to find a counterexample if this is not the case. In the case of open systems, or in the presence of input (controllable) and output (observable) behavior, the conformance relation is commonly described as input-output conformance or *ioco* [18]. A closely related notion of *alternating simulation* [3] is used in the context of open system verification, in particular for interface automata refinement [9,8]. In this paper we propose a theory of *guarded labeled assignment systems* or *GLASs* that formally relates these two notions and provides a foundation for their symbolic analysis.

GLASs are a generalization of *non-deterministic model programs* [23] to a purely symbolic setting, by abstracting from the particular background universe and the particular (action) label domain. The semantics of GLASs uses classical model theory. A GLAS is a symbolic representation of behavior whose trace semantics is given by an LTS that corresponds to the least fix-point of the strongest post-condition induced by the assignment system of the GLAS. We define the notion of *i/o-refinement* over GLASs that is based on alternating simulation and show that it is a generalization of *ioco* for all GLASs, generalizing an earlier result [21] for the deterministic case. The notion of i/o-refinement is essentially a *compositional* version of *ioco*. We provide a rigorous account for formally dealing with *quiescence* in GLASs in a way that supports symbolic analysis with or without the presence of quiescence. We also define the notion of a symbolic composition of GLASs that respects the standard parallel synchronous composition of LTSs [15,16] with the interleaving semantics of unshared labels. Composition

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 47–62, 2010.

of GLASs is used to show that the i/o-refinement relation between two GLASs can be formulated as an condition of the composite GLAS. This leads to a mapping of the non-i/o-refinement checking problem into a reachability checking problem for a pair of GLASs. For a class of GLASs that we call *robust* we can furthermore use established methods developed for verifying safety properties of reactive systems. We show that the non-i/o-refinement checking problem can be reduced to first-order assertion checking by using proof-rules similar to those that have been formulated for checking invariants of reactive systems. It can also be approximated as a *bounded model program checking problem* or BMPC [23]. Detailed proofs of all statements omitted here can be found in the technical report [22].

Although the focus of the paper is theoretical, GLASs provide a foundation of applying state-of-the-art *satisfiability modulo theories* [5] (SMT) technology to a wide range of problems that are difficult to tackle using other techniques. SMT solving is a hybrid technology that has a flavor of model checking, SAT solving, and theorem proving. An advantage over model checking is avoidance of *state-space explosion*. Compared to SAT solving, *bit blasting* can be avoided by encoding operations over unbounded universes, such as integers, more succinctly. Compared to many automated theorem proving techniques, a *solution* is provided as a witness of satisfiability. The following three are sample applications: 1) symbolic model-checking of a given specification GLAS [23] with respect to a given property automaton; 2) symbolic refinement checking between two symbolic LTSs represented as GLASs; 3) incremental model-based parameter generation during on-the-fly testing for increased specification GLAS coverage. In all cases, the use of GLAS composition is central, e.g., for symbolic i/o-refinement or *ioco*, composition is used in Theorem 5. All examples used in the paper are tailored to such analyses and illustrate the use of background theories that are supported by state-of-the-art SMT solvers such as Z3 [10].

2 Preliminaries

We use classical logic and work in a fixed multi-sorted universe $\mathcal{U}$ of values. For each sort σ, $\mathcal{U}^\sigma$ is a sub-universe of $\mathcal{U}$. The basic sorts needed in this paper are the Boolean sort $\mathbb{B}$, ($\mathcal{U}^{\mathbb{B}} = \{true, false\}$), and the integer sort $\mathbb{Z}$. There is a collection of functions with a fixed meaning associated with the universe, e.g., arithmetical operations over $\mathcal{U}^{\mathbb{Z}}$. These functions (and the corresponding function symbols) are called *background* functions. For example, the background function $<: \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ denotes the standard order on integers. There is also a generic background function $Ite: \mathbb{B} \times \sigma \times \sigma \to \sigma$ where σ is a given sort.

Terms are defined by induction as usual and are assumed to be well-sorted. The sort σ of a term t is denoted by $sort(t)$ or by $t{:}\,\sigma$. We write $FV(t)$ for the set of free variables in t. Boolean terms are also called *formulas* or *predicates*. We use x' as an injective *renaming* operation on variables x, and lift the renaming to sets of variables, $\Sigma' \stackrel{\text{def}}{=} \{x' \mid x \in \Sigma\}$. A term t *over* Σ has $FV(t) \subseteq \Sigma$.

A *Σ-model* M is a mapping from Σ to $\mathcal{U}$.[1] The interpretation of a term t over Σ in a Σ-model M, is denoted by t^M and is defined by induction as usual. In particular, $Ite(\varphi, t_1, t_2)^M$ equals t_1^M, if φ^M is true; it equals t_2^M, otherwise.

M satisfies φ or *φ is true in M*, denoted by $M \models \varphi$, if φ^M is *true*. A formula φ is *satisfiable* if it has a model and *valid*, denoted by $\models \varphi$, if φ is true in all models. For two formulas φ and ψ, $\varphi \models \psi$ means that any model of φ is also a model of ψ. We use elements in $\mathcal{U}$ also as terms and define the *predicate of a Σ-model M* as the predicate $P_M \stackrel{\text{def}}{=} \bigwedge_{x \in \Sigma} x = x^M$ over Σ.

3 Guarded Labeled Assignment Systems

This section introduces Guarded Labeled Assignment Systems, GLAS for short. The definition of GLAS combines labels, guarded updates, and internal choice. They capture the semantics of model programs. We start by providing the formal definition, which is followed by examples illustrating the definition. An *assignment* is a pair $x := u$ where x is a variable, u is a term, and $sort(x) = sort(u)$.

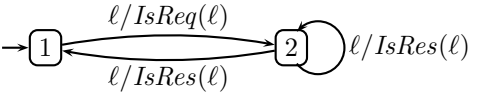
Definition 1. A *Guarded Labeled Assignment System* or *GLAS* G is a tuple $(\Sigma, X, \ell, \imath, \alpha, \gamma, \Delta)$ where

- Σ is a finite set of variables called the *model signature*;
- X is a finite set of variables disjoint from Σ called the *choice signature*;
- ℓ is a variable not in Σ or X, called the *label variable*;
- $\imath$ is a satisfiable formula over Σ called the *initial condition*;
- α is a formula over $\{\ell\}$ called the *label predicate*;
- γ is a formula over $\Sigma \cup X \cup \{\ell\}$ called the *guard*;
- Δ is a set $\{z := u_z\}_{z \in \Sigma}$ where each u_z is a term over $\Sigma \cup X \cup \{\ell\}$, called the *assignment system*.

The set $\Sigma \cup X$ is called the *internal signature* of G.

We first illustrate a simple two-state GLAS.

Example 1. Consider the FSM A:



Intuitively, A specifies a sequence of request and response labels where a single request is followed by one or more respones. Suppose that the labels have sort $\mathbb{L}$ and that $\mathbb{L}$ is associated with predicates $IsReq, IsRes\colon \mathbb{L} \to \mathbb{B}$. A can be represented by the GLAS $G_A = (\{z\colon \mathbb{Z}\}, \{x\colon \mathbb{B}\}, \ell\colon \mathbb{L}, z = 1, IsReq(\ell) \vee IsRes(\ell), Ite(IsReq(\ell), z = 1, z = 2), \{z := Ite(z = 1, 2, Ite(x, 1, 2))\})$. Note that x represents a nondeterministic choice of the target state of a response transition. ⊠

[1] More precisely, variables are viewed as fresh constants expanding the background signature. Note that the background function symbols have the same interpretation in all models (and are thus implicit).

The following example illustrates how an AsmL [4] program can be represented as a GLAS. Other encodings are possible using different techniques. The example makes use of several background sorts. Such sorts are derived from the given program. An important point regarding practical applications is that all sorts and associated axioms that are used are either directly supported, or user definable without any significant overhead, in state-of-the-art SMT solvers.

Example 2. We consider the following model program called *Credits* that describes the message-id-usage facet of a client-server sliding window protocol [14].

```
var ranges as Set of (Integer,Integer) = {(0,0)}
var used as Set of Integer = {}
var max as Integer = 0
var msgs as Map of Integer to Integer = {->}

IsValidUnusedMessageId(m as Integer) as Boolean
  return m notin used and  Exists r in ranges where First(r)<=m and m<=Second(r)

[Action] Req(m as Integer, c as Integer)
  require IsValidUnusedMessageId(m) and c > 0
  msgs(m) := c
  add m to used

[Action] Res(m as Integer, c as Integer)
   require m in msgs and 0<=c and c<=msgs(m)
   remove m from msgs
   if c>0 add (max, max+c) to ranges
   max := max+c
```

Let us assume a sort $\mathbb{L}$ derived from the method signatures of the program; $\mathcal{U}^{\mathbb{L}}$ is an *algebraic data type*. In addition to the predicates *IsReq* and *IsRes* introduced in Example 1, $\mathbb{L}$ is associated with the constructors: $Req, Res\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{L}$ and accessors: $Req\_m, Res\_m, Req\_c, Res\_c\colon \mathbb{L} \to \mathbb{Z}$. For example, $IsReq(Res(6,7))$ is false and $Req\_c(Req(3,4))$ is equal to 4.

The example uses *tuples*. There is a generic n-tuple sort $\mathbb{T}(\sigma_0, \ldots, \sigma_{n-1})$ of given element sorts σ_i for $i < n$. An n-tuple constructor is denoted by $\langle t_0, \ldots, t_{n-1}\rangle$ and the projection functions are denoted by π_i for $i < n$. For example $\pi_1(\langle t_0, t_1\rangle) = t_1$.

The example also uses *arrays*, the sort $\mathbb{A}(\sigma, \rho)$ is a generic sort for extensional arrays (mathematical maps) with domain sort σ and range sort ρ. The functions on arrays are reading and storing elements in the array:

$$Read\colon \mathbb{A}(\sigma,\rho) \times \sigma \to \rho, \quad Store\colon \mathbb{A}(\sigma,\rho) \times \sigma \times \rho \to \mathbb{A}(\sigma,\rho).$$

The *empty array* ε maps every domain element to a *default* value of the range sort. (For $\mathbb{Z}$ the default is 0 and for $\mathbb{B}$ the default is *false*. The axioms assumed for arrays are the usual ones for propagating reads over store and the extensionality axiom.)

We map *Credits* to the GLAS $G_{Credits}$: $(\Sigma, \emptyset, \ell, \imath, IsReq(\ell) \vee IsRes(\ell), \gamma, \Delta)$ where $\Sigma = \{ranges\colon \mathbb{A}(\mathbb{T}(\mathbb{Z},\mathbb{Z}),\mathbb{B}), used\colon \mathbb{A}(\mathbb{Z},\mathbb{B}), max\colon \mathbb{Z}, msgs\colon \mathbb{A}(\mathbb{Z},\mathbb{Z})\}$. The initial condition $\imath$ is

$$ranges = Store(\varepsilon, \langle 0,0\rangle, true) \wedge used = \varepsilon \wedge max = 0 \wedge msgs = \varepsilon.$$

Given by the require-statements, the guard γ is:

$$\begin{aligned}
&(IsReq(\ell) \wedge Req_m(\ell) \notin used \\
&\qquad \wedge \exists r\,(r \in ranges \wedge \pi_0(r) \leq Req_m(\ell) \wedge Req_m(\ell) \leq \pi_1(r)) \\
&\qquad \wedge Req_c(\ell) > 0) \vee \\
&(IsRes(\ell) \wedge Res_m(\ell) \in msgs \wedge 0 \leq Res_c(\ell) \\
&\qquad \wedge Res_c(\ell) \leq Read(msgs, Res_m(\ell)))
\end{aligned}$$

The assignment system Δ consists of the assignments:

$$\begin{aligned}
ranges{:=}&Ite(IsReq(\ell), ranges, Ite(Res_c(\ell) > 0, \\
&\qquad Store(ranges, \langle max, max + Res_c(\ell)\rangle, true), ranges)) \\
used{:=}&Ite(IsReq(\ell), Store(used, Req_m(\ell), Req_c(\ell)), used) \\
max{:=}&Ite(IsReq(\ell), max, max + Res_c(\ell)) \\
msgs{:=}&Ite(IsReq(\ell), Store(msgs, Req_m(\ell), Req_c(\ell)), \\
&\qquad Store(msgs, Res_m(\ell), default_{\mathbb{Z}}))
\end{aligned}$$

The right-hand-sides of the assignments are easy to automatically generate from the program, but much harder to comprehend than the original assignments in the program, since they combine all the assignments from the separate actions by doing a case split based on the action label. They also add trivial assignments that take care of the implicit *frame condition* in AsmL that states that all variables not updated retain their previous values. ⊠

A GLAS is a symbolic representation of a labeled transition system (LTS). In order to keep the paper self-contained and to fix the notations we include the standard definitions of LTSs and traces.

Definition 2. An *LTS* is a tuple $\mathcal{L} = (\mathbf{S}, \mathbf{S}^0, L, T)$, where $\mathbf{S}$ is a set of *states*; $\mathbf{S}^0 \subseteq \mathbf{S}$ is a nonempty set of *initial states*; L is a set of *labels*; $T \subseteq \mathbf{S} \times L \times \mathbf{S}$ is a *transition relation.* A label $a \in L$ is *enabled in a state* S if $(S, a, S') \in T$ for some $S' \in \mathbf{S}$. $\mathcal{L}$ is *deterministic* if $\mathcal{L}$ has a single initial state and for all $a \in L$ and $S \in \mathbf{S}$ there is at most one $S' \in \mathbf{S}$ such that $(S, a, S') \in T$.

We use $\mathcal{L}$ as a subscript to identify its components. If $(S, a, S') \in T_{\mathcal{L}}$ we write $S \xrightarrow{a}_{\mathcal{L}} S'$ or $S \xrightarrow{a} S'$ if $\mathcal{L}$ is clear from the context. If $a \in L_{\mathcal{L}}$ is enabled in $S \in \mathbf{S}_{\mathcal{L}}$ write $S \xrightarrow{a}_{\mathcal{L}}$. If $a \in L_{\mathcal{L}}$ is not enabled in $S \in \mathbf{S}_{\mathcal{L}}$, we write $S \not\xrightarrow{a}_{\mathcal{L}}$. In this paper we are only concerned with *finite* traces.

Definition 3. A label sequence $\mathbf{a} = (a_i)_{i<k}$ such that $S_i \xrightarrow{a_i}_{\mathcal{L}} S_{i+1}$, $i < k$, is a *trace of* $\mathcal{L}$ *from* S_0 or a *trace of* $\mathcal{L}$ if $S_0 \in \mathbf{S}^0_{\mathcal{L}}$; we write $S_0 \xrightarrow{\mathbf{a}} S_k$ and $S \xrightarrow{\epsilon} S$ where ϵ is the empty sequence. The set of all traces of $\mathcal{L}$ is denoted by $Tr(\mathcal{L})$.

When $\mathcal{L}$ is deterministic, we view $\mathcal{L}$ as a function from all label sequences $\mathbf{a}$ to states or the value $\perp_{\mathcal{L}}$ when $\mathbf{a}$ is not a trace of $\mathcal{L}$. Thus,

$$\mathcal{L}(\mathbf{a}) \stackrel{\text{def}}{=} \begin{cases} \perp_{\mathcal{L}}, \text{ if } \mathbf{a} \notin Tr(\mathcal{L}); \\ S, \quad \text{otherwise, where } \mathbf{S}^0_{\mathcal{L}} = \{S^0\} \text{ and } S^0 \xrightarrow{\mathbf{a}}_{\mathcal{L}} S. \end{cases} \tag{1}$$

Note that $\mathcal{L}(\epsilon)$ is the unique initial state of a deterministic LTS $\mathcal{L}$.

A GLAS is associated with a transition relation formula that describes a single application of its assignments and a predicate transformer that maps a given predicate to a new predicate. The predicate transformer is used below to define semantics of GLASs in terms of LTSs.

Definition 4. Let $G = (\Sigma, X, \ell, \imath(\Sigma), \alpha, \gamma(\Sigma), \{z := u_z(\Sigma)\}_{z\in\Sigma})$ be a GLAS. We define the *transition relation* TR_G, and the *strongest post-condition predicate transformer* SP_G, for G, where $P(\Sigma)$ is a predicate over Σ:

$$TR_G(\Sigma', \ell, \Sigma) \stackrel{\text{def}}{=} \alpha \wedge \exists X\, (\gamma(\Sigma') \wedge \bigwedge_{z\in\Sigma} z = u_z(\Sigma'))$$

$$SP_G(P, \ell) \stackrel{\text{def}}{=} \exists \Sigma'\, (P(\Sigma') \wedge TR_G(\Sigma', \ell, \Sigma))$$

Note that, for $a \in \mathcal{U}^{sort(\ell)}$, $SP_G(P, a)$ is a predicate over Σ. Next, we define two related semantics of a GLAS G in terms of LTSs. One is the *concrete semantics* $\lfloor G \rfloor$ and the other one is the *symbolic semantics* $\lceil G \rceil$. In the concrete semantics, states are Σ_G-models. In the symbolic semantics, states are predicates over Σ_G in the SP_G-closure of $\{\imath_G\}$. We define the set of *labels of* G as

$$L_G \stackrel{\text{def}}{=} \{\ell_G^M \mid M \models \alpha_G\}.$$

Definition 5. $\lfloor G \rfloor = (\mathbf{S}, \{M \mid M \models \imath_G\}, L_G, T)$ where $\mathbf{S}$, T are the least sets such that $\mathbf{S}^0_{\lfloor G \rfloor} \subseteq \mathbf{S}$ and $(M, a, N) \in T$ for $a \in L_G$, $M \in \mathbf{S}$, and $N \models SP_G(P_M, a)$, then $N \in \mathbf{S}$.

Definition 6. $\lceil G \rceil = (\mathbf{S}, \{\imath_G\}, L_G, T)$ where $\mathbf{S}$, T are the least sets such that $\imath_G \in \mathbf{S}$, $(P, a, SP_G(P, a)) \in T$ for $a \in L_G$, $P \in \mathbf{S}$ where $SP_G(P, a)$ is satisfiable.

The notion of traces of G is based on the symbolic semantics of G.

Definition 7. $Tr(G) \stackrel{\text{def}}{=} Tr(\lceil G \rceil)$.

We show that both semantics yield the same traces, i.e., $\lceil G \rceil$ does not introduce new traces, although several models of $\lfloor G \rfloor$ may collapse into a single state in $\lceil G \rceil$. We use the following technical lemma. Note that $\lceil G \rceil$ is deterministic and recall (1); let $\bot_{\lceil G \rceil} \stackrel{\text{def}}{=} \mathit{false}$. Given a sequence $\mathbf{a}$ and an element a, we write $\mathbf{a} \cdot a$ for the extended sequence. The empty sequence is denoted by ϵ.

Lemma 1. *For all* $\mathbf{a}$, $\{M \mid M \models \lceil G \rceil(\mathbf{a})\} = \{M \mid \exists M_0 \in \mathbf{S}^0_{\lfloor G \rfloor}\, (M_0 \xrightarrow{\mathbf{a}}_{\lfloor G \rfloor} M)\}$.

Proof. By induction over the length of $\mathbf{a}$. The base case, $\mathbf{a} = \epsilon$, holds trivially by $\{M \mid M \models \lceil G \rceil(\epsilon)\} = \mathbf{S}^0_{\lfloor G \rfloor} = \{M \mid \exists M_0 \in \mathbf{S}^0_{\lfloor G \rfloor}\, (M_0 \xrightarrow{\epsilon}_{\lfloor G \rfloor} M)\}$. Assume by IH that the statement holds for $\mathbf{a}$, we prove it for $\mathbf{a} \cdot a$.

$$\{M \mid M \models \lceil G \rceil(\mathbf{a} \cdot a)\} \stackrel{(\text{def } 6)}{=} \{M \mid M \models SP_G(\lceil G \rceil(\mathbf{a}), a)\}$$
$$\stackrel{(\text{def } 4)}{=} \{M \mid M \models \exists \Sigma'(\lceil G \rceil(\mathbf{a})(\Sigma') \wedge TR_G(\Sigma', a, \Sigma))\}$$

$$
\begin{aligned}
&= && \{M \mid \exists N (N \models \lceil G \rceil(\mathbf{a}), \\
&&& \quad M \models \exists \Sigma' (P_N(\Sigma') \wedge TR_G(\Sigma', a, \Sigma)))\} \\
&\overset{\text{(IH)}}{=} && \{M \mid \exists N \, \exists M_0 \in \mathbf{S}^0_{\lfloor G \rfloor} \, (M_0 \xrightarrow{\mathbf{a}}_{\lfloor G \rfloor} N, \\
&&& \quad M \models \exists \Sigma' (P_N(\Sigma') \wedge TR_G(\Sigma', a, \Sigma)))\} \\
&\overset{(\text{def } 4)}{=} && \{M \mid \exists N \, \exists M_0 \in \mathbf{S}^0_{\lfloor G \rfloor} \, (M_0 \xrightarrow{\mathbf{a}}_{\lfloor G \rfloor} N, \\
&&& \quad M \models SP_G(P_N, a))\} \\
&\overset{(\text{def } 5)}{=} && \{M \mid \exists N \, \exists M_0 \in \mathbf{S}^0_{\lfloor G \rfloor} \, (M_0 \xrightarrow{\mathbf{a}}_{\lfloor G \rfloor} N, \\
&&& \quad N \xrightarrow{a}_{\lfloor G \rfloor} M)\} \\
&= && \{M \mid \exists M_0 \in \mathbf{S}^0_{\lfloor G \rfloor} \, (M_0 \xrightarrow{\mathbf{a} \cdot a}_{\lfloor G \rfloor} M)\}
\end{aligned}
$$

The statement follows by the induction principle. ⊠

The lemma implies the following theorem that is a fundamental property of the symbolic semantics. It justifies the whole approach presented in the paper and provides a symbolic generalization of the classical LTS determinization.

Theorem 1. $Tr(\lceil G \rceil) = Tr(\lfloor G \rfloor)$.

Proof. $Tr(\lceil G \rceil)$ equals $\{\mathbf{a} \mid \{M \mid M \models \lceil G \rceil(\mathbf{a})\} \neq \emptyset\}$ that, by Lemma 1, equals $\{\mathbf{a} \mid \{M \mid \exists M_0 \in \mathbf{S}^0_{\lfloor G \rfloor}(M_0 \xrightarrow{\mathbf{a}}_{\lfloor G \rfloor} M)\} \neq \emptyset\}$ that equals $\{\mathbf{a} \mid \exists M \, \exists M_0 \in \mathbf{S}^0_{\lfloor G \rfloor}(M_0 \xrightarrow{\mathbf{a}}_{\lfloor G \rfloor} M)\}$ that is the definition of $Tr(\lfloor G \rfloor)$. ⊠

There is an important point about this choice of trace-style semantics. It is tailored for the case where internal choices of GLASs are *opaque*. Symbolic semantics plays an important role when we later define alternating simulation and conformance, where G may be nondeterministic, i.e., $\lfloor G \rfloor$ is nondeterministic, but where $\lceil G \rceil$ is used, which, by Theorem 1, does not change the intended trace semantics of G. Moreover, $\lceil G \rceil$ directly reflects the symbolic unfolding of the transition relation of a GLAS, that is fundamental in the construction of first-order assertions for reduction to symbolic analysis.

Example 3. The Credits program in Example 2 is deterministic. The following is a trace of $G_{Credits}$: $(Req(0,3), Res(0,2), Req(2,1), Req(1,1), Res(2,0), Res(1,0))$. Intuitively, the trace describes a valid communication scenario between the client and the server (based on a sliding window protocol), where the client is able to use message ids based on credits granted earlier by the server. ⊠

3.1 GLAS Composition

Composition of GLASs is a purely symbolic construction.

Definition 8. Let $G_i = (\Sigma_i, X_i, \ell, \imath_i, \alpha_i, \gamma_i, \{z := u_z\}_{z \in \Sigma_i})$, for $i \in I$, be GLASs with disjoint internal signatures. The *composition of* G_i *for* $i \in I$, is the GLAS

$$\bigotimes_{i \in I} G_i \overset{\text{def}}{=} (\bigcup_{i \in I} \Sigma_i, \bigcup_{i \in I} X_i, \ell, \bigwedge_{i \in I} \imath_i, \bigvee_{i \in I} \alpha_i, \bigwedge_{i \in I} (\alpha_i \Rightarrow \gamma_i), \bigcup_{i \in I} \{z := Ite(\alpha_i, u_z, z)\}_{z \in \Sigma_i})$$

We abbreviate $\bigotimes_{i\in I} G_i$ by $\bigotimes_I G_i$ and for $\bigotimes_{\{1,2\}} G_i$ we write $G_1 \otimes G_2$. Note that $\bigotimes_I G_i$ is indeed well-defined as a GLAS. In particular, $\imath_{\bigotimes_I G_i}$ is satisfiable because all the individual initial conditions are satisfiable and do not share free variables. The other side conditions in Definition 1 hold similarly. The following technical lemma is used below. Let G_i, for $i \in I$, be as above.

Lemma 2. *Let $G = \bigotimes_I G_i$. Assume $L_{G_i} = L_{G_j}$ for $i, j \in I$. Let P_i be a predicate over Σ_i for $i \in I$. Then $\models SP_G(\bigwedge_{i\in I} P_i, a) \Leftrightarrow \bigwedge_{i\in I} SP_{G_i}(P_i, a)$.*

Proof. We first show (*): $\models TR_G(\Sigma'_G, \ell, \Sigma_G) \Leftrightarrow \bigwedge_{i\in I} TR_{G_i}(\Sigma'_{G_i}, \ell, \Sigma_{G_i})$ by using the assumption, definition of TR_G, Definition 8, and standard logical transformations (that use disjointness of the internal signatures of G_i for $i \in I$). The lemma follows by using (*), Definition 4, and further logical transformations. ⊠

One can show that composition of GLASs respects the standard parallel synchronous composition of LTSs with the interleaving semantics of unshared labels. Here we assume the special case of all labels being shared, i.e. $L_{G_i} = L_{G_j}$ for $i, j \in I$. A general statement can be formulated that describes the interleaving of unshared labels, but the special case is sufficient for this paper.

Theorem 2. *Let $G = \bigotimes_I G_i$. Assume $L_{G_i} = L_{G_j}$ for $i, j \in I$.*
(i) *For all* $\mathbf{a}$, $\models \lceil G \rceil(\mathbf{a}) \Leftrightarrow \bigwedge_{i\in I} \lceil G_i \rceil(\mathbf{a})$.
(ii) $Tr(G) = \bigcap_{i\in I} Tr(G_i)$.

Proof. We prove (i) by induction over $\mathbf{a}$. The base case holds trivially since $\lceil G \rceil(\epsilon) = \bigwedge_{i\in I} \imath_i = \bigwedge_{i\in I} \lceil G_i \rceil(\epsilon)$. Assume (i) holds for $\mathbf{a}$; we prove (i) for $\mathbf{a} \cdot a$:

$$\begin{aligned} \lceil G \rceil(\mathbf{a}\cdot a) &\overset{(\text{def } 6)}{\Leftrightarrow} SP_G(\lceil G \rceil(\mathbf{a}), a) \overset{(\text{IH})}{\Leftrightarrow} SP_G(\textstyle\bigwedge_I \lceil G_i \rceil(\mathbf{a}), a) \\ &\overset{(\text{lemma } 2)}{\Leftrightarrow} \textstyle\bigwedge_I SP_{G_i}(\lceil G_i \rceil(\mathbf{a}), a) \\ &\overset{(\text{def } 6)}{\Leftrightarrow} \textstyle\bigwedge_I \lceil G_i \rceil(\mathbf{a}\cdot a) \end{aligned}$$

Statement (i) follows by the induction principle. We now prove (ii):

$$\begin{aligned} Tr(G) = \{\mathbf{a} \mid \lceil G \rceil(\mathbf{a}) \neq \mathit{false}\} &\overset{\text{by (i)}}{=} \{\mathbf{a} \mid \textstyle\bigwedge_I \lceil G_i \rceil(\mathbf{a}) \text{ is satisfiable}\} \\ &= \{\mathbf{a} \mid \forall i \in I(\lceil G_i \rceil(\mathbf{a}) \neq \mathit{false})\} \\ &= \{\mathbf{a} \mid \forall i \in I(\mathbf{a} \in Tr(G_i))\} \end{aligned}$$

The third equality assumes disjointness of Σ_{G_i} for $i \in I$. ⊠

Example 4. Consider the composition $G = G_{Credits} \otimes G_A$ with $G_{Credits}$ and G_A from examples 2 and 1, respectively. The traces of G are the traces of both $G_{Credits}$ and G_A, i.e., the traces that conform to the *Credits* specification while restricted to the scenarios described by A. For example, the trace illustrated in Example 3 is therefore *not* a trace of G. ⊠

4 I/O GLAS

Here we consider GLASs where the labels are divided into input and output labels that describe reactive or open system behavior.

Definition 9. An *i/o-GLAS* G is an extension $(G', \alpha^{\text{out}})$ of a GLAS G' where α^{out} is a formula such that $\alpha^{\text{out}} \models \alpha_G$ called the *output label predicate*.

In the corresponding i/o LTS the labels are separated so that L_G^{out} is the set of all labels that satisfy α_G^{out} and L_G^{in} is the set of all labels that satisfy $\alpha_G \wedge \neg\alpha_G^{\text{out}}$. We say GLAS (LTS) to also mean i/o-GLAS (i/o LTS) and let the context determine whether the labels are separated into input and output labels.

Example 5. Consider the *Credits* program and assume that *Req* is marked as an input-action and *Res* is marked as an output-action. The output label predicate α^{out} is a disjunction over all cases of action labels in the AsmL program that are marked as output-actions, i.e, in this case α^{out} is $IsRes(\ell)$. ⊠

When dealing with formal notions of conformance, in particular *ioco* [18], an important aspect is how to deal with *quiescence*, that is a special output label, usually denoted by δ, indicating absence of other enabled output labels in a given state. An LTS can be extended to include δ as a new output label [18]:

Definition 10. Let $\mathcal{L}$ be an LTS and $\delta \notin L_{\mathcal{L}}$. Then $\mathcal{L}^\delta$ is the extension of $\mathcal{L}$ where $L_{\mathcal{L}^\delta}^{\text{out}} = L_{\mathcal{L}}^{\text{out}} \cup \{\delta\}$ and $S \xrightarrow{\delta}_{\mathcal{L}^\delta} S$ iff for all $a \in L_{\mathcal{L}}^{\text{out}}$, $S \overset{a}{\not\rightarrow}_{\mathcal{L}}$.

We define a corresponding symbolic extension for GLASs.

Definition 11. For $G = (\Sigma, X, \ell, \imath, \alpha, \gamma, \{z := u_z\}_{z\in\Sigma}, \alpha^{\text{out}})$, $\delta \in \mathcal{U}^{sort(\ell)} \setminus L_G$:

$$G^\delta \overset{\text{def}}{=} (\ \Sigma,\ X,\ \ell,\ \imath,\ \alpha \vee \ell = \delta,\ Ite(\ell = \delta, \neg\exists \ell\, X\, (\alpha^{\text{out}} \wedge \gamma), \gamma),\\ \{z := Ite(\ell = \delta, z, u_z)\}_{z\in\Sigma},\ \alpha^{\text{out}} \vee \ell = \delta)$$

Thus, in G^δ there is a new output label δ and $M \xrightarrow{\delta}_{\lfloor G^\delta \rfloor}$ if and only if for all $a \in L_G^{\text{out}}$, $M \overset{a}{\not\rightarrow}_{\lfloor G \rfloor}$. The intended meaning of G^δ is made precise by the following theorem that says that the symbolic extension precisely captures the intended suspension trace semantics [18] of $\lfloor G \rfloor$.

Theorem 3. (i) $\lfloor G^\delta \rfloor = \lfloor G \rfloor^\delta$. (ii) $Tr(G^\delta) = Tr(\lfloor G \rfloor^\delta)$.

Proof. (i) follows from definitions. (ii) uses (i), Definition 7 and Theorem 1. ⊠

Note however that $Tr(G^\delta) \neq Tr(\lceil G \rceil^\delta)$ as illustrated by the following example which also illustrates the use of choice variables in a GLAS.

Example 6. The example is derived from a standard example that is used to illustrate properties of quiescence during determinization of non-deterministic LTSs [18, Figure 6]. The GLAS G is represented below by an FSM where there

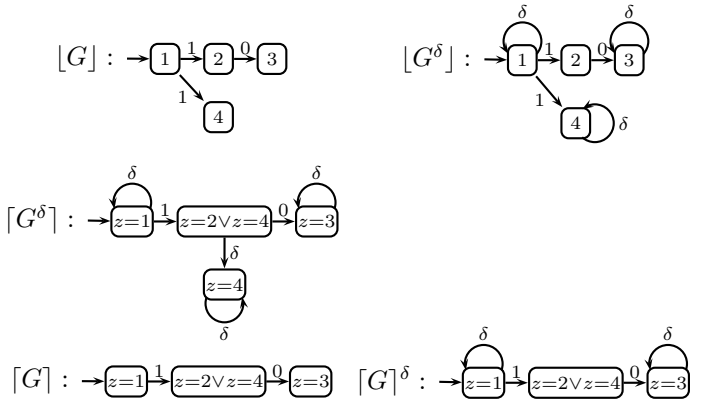
is a single input label 1 and a single output label 0. We assume the following representation for G:

$$G = (\ \{z{:}\mathbb{Z}\},\ \{x{:}\mathbb{B}\},\ \ell{:}\mathbb{Z},\ z = 1,\ 0 \le \ell \le 1,\ \ell = 0 \Leftrightarrow z = 2,\\ \{z := \mathit{Ite}(z = 1, \mathit{Ite}(x, 2, 4), 3)\},\ \ell = 0\ )$$

G^δ is the following GLAS where we have simplified γ_{G^δ} by using that the formula $\neg\exists\ell\, x\,(\ell = 0 \wedge (\ell = 0 \Leftrightarrow z = 2))$ is equivalent to $z \neq 2$. (Let, e.g. $\delta = 2$),

$$G^\delta = (\ \{z{:}\mathbb{Z}\},\ \{x{:}\mathbb{B}\},\ \ell{:}\mathbb{Z},\ z = 1,\ 0 \le \ell \le 1 \vee \ell = \delta,\\ \mathit{Ite}(\ell = \delta, z \neq 2, \ell = 0 \Leftrightarrow z = 2),\\ \{z := \mathit{Ite}(\ell = \delta, z, \mathit{Ite}(z = 1, \mathit{Ite}(x, 2, 4), 3))\},\ \ell = 0 \vee \ell = \delta\ )$$

We can illustrate the GLASs as follows:



Thus $\lceil G^\delta \rceil \neq \lceil G \rceil^\delta$. Moreover, $\mathit{Tr}(\lceil G^\delta \rceil) \neq \mathit{Tr}(\lceil G \rceil^\delta)$. ⊠

Example 7. Consider $G = G_{Credits}$ from Example 2. The formula that defines absence of outputs in G, $\neg\exists\ell\, X_G(\alpha_G^{\text{out}} \wedge \gamma_G)$, is, after simplifications, equivalent to the formula $\mathit{msgs} = \varepsilon$. Intuitively, there should not be a response from the server, i.e. the server must be quiescent, if there is no pending request from the client, i.e., δ is enabled in any model of $\lfloor G^\delta \rfloor$ where msgs is empty. ⊠

We define a notion of conformance between two GLASs that is based on *alternating simulation* [3] between two LTSs and show below that this notion of conformance coincides with *ioco* for GLASs.

Let $\mathcal{M}_i = (\mathbf{S}_i, \{S_i^0\}, L_i, L_i^{\text{in}}, L_i^{\text{out}}, T_i)$, for $i = 1, 2$, be deterministic LTSs.[2] The intuition behind the following definition is that $\mathcal{M}_1$ can only make outputs that $\mathcal{M}_2$ can make, and $\mathcal{M}_2$ can only make inputs that $\mathcal{M}_1$ can make.

Definition 12. $\mathcal{M}_1$ *i/o-refines* $\mathcal{M}_2$, $\mathcal{M}_1 \preceq \mathcal{M}_2$, iff there exists an alternating simulation ρ from $\mathcal{M}_1$ to $\mathcal{M}_2$ such that $(S_1^0, S_2^0) \in \rho$, where an *alternating simulation from* $\mathcal{M}_1$ *to* $\mathcal{M}_2$ is a relation $\rho \subseteq \mathbf{S}_1 \times \mathbf{S}_2$ such that, for all $(S_1, S_2) \in \rho$

[2] Deterministic LTSs are called *interface automata* in [9].

$$\forall o \in L_1^{\text{out}}(S_1 \xrightarrow{o}_{\mathcal{M}_1} S_1' \Rightarrow \exists S_2'(S_2 \xrightarrow{o}_{\mathcal{M}_2} S_2' \wedge (S_1', S_2') \in \rho))$$
$$\forall i \in L_2^{\text{in}}(S_2 \xrightarrow{i}_{\mathcal{M}_2} S_2' \Rightarrow \exists S_1'(S_1 \xrightarrow{i}_{\mathcal{M}_1} S_1' \wedge (S_1', S_2') \in \rho))$$

Given GLASs G and H then $G \preceq H \stackrel{\text{def}}{=} \lceil G \rceil \preceq \lceil H \rceil$.

Definition 12 is consistent with [8]. In particular, several foundational properties of $\preceq$ (like reflexivity and transitivity) are established in [8] that show that $\preceq$ is a suitable refinement relation.

Example 8. Consider two GLASs *Spec* and *Impl* where $\ell : \mathbb{B}$ and α^{out} is $\neg\ell$.

$$\lceil Spec \rceil = (\{S_1\}, \{S_1\}, \mathcal{U}^{\mathbb{B}}, \{true\}, \{false\}, \{(S_1, false, S_1)\})$$
$$\lceil Impl \rceil = (\{S_2\}, \{S_2\}, \mathcal{U}^{\mathbb{B}}, \{true\}, \{false\}, \{(S_2, true, S_2)\})$$

It is easy to see that $Impl \preceq Spec$ and $Spec \not\preceq Impl$. ⊠

A useful characterization of i/o-refinement uses counter-examples.

Definition 13. A sequence $\mathbf{a} \cdot a$ is a *witness of* $\mathcal{M}_1 \not\preceq \mathcal{M}_2$ if $\mathbf{a} \in Tr(\mathcal{M}_1) \cap Tr(\mathcal{M}_2)$ and either $a \in L_1^{\text{in}}$ and $\mathbf{a} \cdot a \in Tr(\mathcal{M}_1) \setminus Tr(\mathcal{M}_2)$, or $a \in L_1^{\text{out}}$ and $\mathbf{a} \cdot a \in Tr(\mathcal{M}_2) \setminus Tr(\mathcal{M}_1)$.

For example, the (singleton) sequence *true* is a witness of $\lceil Spec \rceil \not\preceq \lceil Impl \rceil$ in Example 8. The following lemma justifies Definition 13.

Lemma 3. $\mathcal{M}_1 \preceq \mathcal{M}_2 \iff \mathcal{M}_1 \not\preceq \mathcal{M}_2$ *has no witnesses.*

For symbolic analysis, we are interested in the approximations of i/o-refinement that hold for a given upper length bound on traces.

Definition 14. $\mathcal{M}_1 \preceq_n \mathcal{M}_2 \stackrel{\text{def}}{=} \mathcal{M}_1 \not\preceq \mathcal{M}_2$ has no witness of length $\leq n$.

It follows directly from Lemma 3 that $\mathcal{M}_1 \preceq \mathcal{M}_2$ iff $\mathcal{M}_1 \preceq_n \mathcal{M}_2$ for all $n > 0$. For example, $Spec \not\preceq_1 Impl$ in Example 8. We are interested in the following decision problem. For GLASs G and H, a *witness of* $G \not\preceq H$ is a witness of $\lceil G \rceil \not\preceq \lceil H \rceil$ and we let $G \preceq_n H \stackrel{\text{def}}{=} \lceil G \rceil \preceq_n \lceil H \rceil$.

Definition 15. *Bounded Non-Conformance* or *BNC* is the problem of deciding if $G \not\preceq_n H$, for given G, H and $n > 0$, and finding a witness of $G \not\preceq_n H$.

We show how to reduce BNC to the BMPC problem [23] for a class of GLASs. There is a mapping of the *BMPC* problem over AsmL model programs and the encoding described in [23] to GLASs: given G, n, and a *reachability condition* φ that is a formula such that $FV(\varphi) \subseteq \Sigma_G$, decide if there exists a trace $\mathbf{a}$ of G of length $\leq n$ such that $M \models \varphi$ for *some* $M \models \lceil G \rceil(\mathbf{a})$. For this reduction we need to consider GLASs that are *robust* in the following sense.

Definition 16. For $a \in L_G$ and $P \in \mathbf{S}_{\lceil G \rceil}$, a is *robust in* P if $P \xrightarrow{a}_{\lceil G \rceil}$ implies $M \xrightarrow{a}_{\lfloor G \rfloor}$ for all $M \models P$.

Intuitively, if a is robust and enabled in a symbolic state, then a is enabled in *all* of the corresponding concrete states.

Definition 17. G is *output-robust* (*input-robust*) if for all $P \in \mathbf{S}_{\lceil G \rceil}$ and all $a \in L_G^{\text{out}}$ ($a \in L_G^{\text{in}}$), a is robust in P. G is *robust* if it is both input-robust and output-robust.

The intuition behind robustness is that internal choices should behave *uniformly* in terms of external behavior. For example, deterministic GLASs (such as $G_{Credits}$) are trivially robust, since there are no internal choices. The following example illustrates a nontrivial example of a robust GLAS that is nondeterministic and where internal choices arise naturally as a way of abstracting externally visible behavior.

Example 9. We consider the *Credits* program and modify it by abstracting the message ids from the labels. The constructors of the $\mathbb{L}$ sort are also modified so that $Req, Res : \mathbb{Z} \to \mathbb{L}$ and the accessors *Req_m* and *Res_m* are removed. We call the resulting program *Credits2*:

```
[Action] Req(c as Integer)
  require exists m where IsValidUnusedMessageId(m) and c > 0
  choose m where IsValidUnusedMessageId(m)
    msgs(m) := c
    add m to used

[Action] Res(c as Integer)
   require  exists m where m in msgs and 0<=c and c<=msgs(m)
   choose m where m in msgs and 0<=c and c<=msgs(m)
     remove m from msgs
     if c>0 add (max, max+c) to ranges
     max := max+c
```

We write *Credits* and *Credits2* also for the corresponding GLASs. *Credits2* has two choice variables, say m_{Req} and m_{Res}, the guard and the assignment system of *Credits2* is obtained from the guard and the assignment system of *Credits* by replacing each occurrence of $Req\_m(\ell)$ ($Res\_m(\ell)$) with m_{Req} (m_{Res}). It is easy to see that *Credits2* is non-deterministic. For example, $\mathbf{a} = (Req(3), Res(3), Req(1))$ is a trace of *Credits2*. After $Req(3)$ there is a pending request with id 0. After $Res(3)$ the range of possible message ids contains the pair $\langle 1, 3\rangle$ and the used set of messages in $\{0\}$. After $Req(1)$, i.e., in the state $S = \lceil Credits2 \rceil(\mathbf{a})$, there are 3 possible models. One can show that *Credits2* is both input-robust and output-robust, the key property that determines enabledness of a request is the *number* of available message ids, similarly for responses. ⊠

The following example illustrates a GLAS that is not output-robust.

Example 10. We consider the *Credits* program again and this time we modify only the *Req* action as in Example 9. We call it *Credits3*. For example, consider the trace $\mathbf{a} = (Req(3), Res(0, 3), Req(1))$ of *Credits3*. The state $S = \lceil Credits2 \rceil(\mathbf{a})$ contains 3 models, where, for example, the output label $Res(1, 1)$

is only enabled in the model in S where request 1 is pending but not in the model where request 2 is pending. So *Credits3* is not output-robust. ⊠

The key insight of reducing BNC to BMPC comes from Lemma 3 and the use of composition. Let α^{in} stand for the formula $\alpha \wedge \neg\alpha^{\text{out}}$. We assume that G and H below have disjoint internal signatures.

$$P_{\preceq}(G,H) \stackrel{\text{def}}{=} \forall \ell\,(((\alpha_G^{\text{out}} \wedge \exists X_G\,\gamma_G) \Rightarrow (\alpha_H^{\text{out}} \wedge \exists X_H\,\gamma_H)) \wedge ((\alpha_H^{\text{in}} \wedge \exists X_H\,\gamma_H) \Rightarrow (\alpha_G^{\text{in}} \wedge \exists X_G\,\gamma_G)))$$

The intuition behind $P_{\preceq}$ is that if an output label ℓ is possible in G then ℓ must be possible in H, and vice versa for input labels. We get the following corollary by using the definition $P_{\preceq}$, Lemma 3, and Theorem 2.

Corollary 1. $G \not\preceq H \iff \exists \mathbf{a} \in Tr(G \otimes H)(\lceil G \otimes H \rceil(\mathbf{a}) \models \neg P_{\preceq}(G,H))$.

In the following theorem we assume, without loss of generality, that $L_{\lceil G \rceil} = L_{\lceil H \rceil}$. The proof uses Lemma 3.

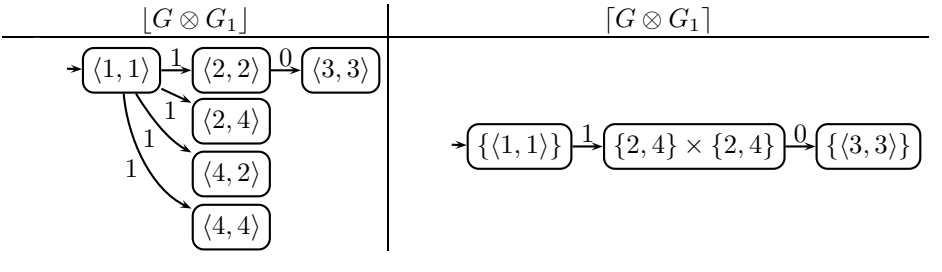
Theorem 4. *Assume G is input-robust and H is output-robust. $G \preceq_n H \iff$ forall $\mathbf{a} \in Tr(G \otimes H)$, if length$(\mathbf{a}) < n$ then $\lceil G \otimes H \rceil(\mathbf{a}) \models P_{\preceq}(G,H)$.*

The robustness assumptions are not needed for the direction $\Longleftarrow$ of the theorem. It is easy to show that the direction $\Longrightarrow$ does not hold without the assumption.

Example 11. Consider the GLAS G illustrated by the FSM in Example 6. Note that G is not output-robust. Let G_1 be a copy of G where z is replaced by z_1 and x is replaced by x_1. Clearly $\lceil G \rceil \preceq \lceil G_1 \rceil$ (since $\lceil G \rceil \preceq \lceil G \rceil$ by reflexivity of $\preceq$). Now consider $G \otimes G_1$, where

$$\begin{aligned} \imath_{G\otimes G_1} &= (z = 1) \wedge (z_1 = 1);\\ \gamma_{G\otimes G_1} &= (\ell = 1 \wedge z = 1 \wedge z_1 = 1) \vee (\ell = 0 \wedge z = 2 \wedge z_1 = 2);\\ \Delta_{G\otimes G_1} &= \{z := \mathit{Ite}(z = 1, \mathit{Ite}(x, 2, 4), 3),\\ &\qquad z_1 := \mathit{Ite}(z_1 = 1, \mathit{Ite}(x_1, 2, 4), 3)\}. \end{aligned}$$

The LTSs $\lfloor G \otimes G_1 \rfloor$ and $\lceil G \otimes G_1 \rceil$ can be illustrated as follows where a pair $\langle z, z_1 \rangle$ shows the values of the respective model variables:



Consider the singleton trace 1. Fix $M = \{z \mapsto 2, z_1 \mapsto 4\} \models \lceil G \otimes G_1 \rceil(1)$. It is easy to see that $M \not\models P_{\preceq}(G, G_1)$ because $M \cup \{\ell \mapsto 0\} \not\models z_1 = 2$. ⊠

The following example illustrates a case when H in Theorem 4 is nondeterministic but robust.

Example 12. We consider a model program *CreditsImpl* that describes the abstracted behavior of a protocol implementation.

```
var cs as Seq of Integer = []
[Action] Req(c as Integer)
  require true
  cs := cs + [c]
[Action] Res(c as Integer)
   require c <> [] and c <= Head(cs) and c >= 0
   cs := Tail(cs)
```

The GLASs *Credits2* (from Example 9) and *CreditsImpl* are robust. On can show that *CreditsImpl* $\preceq_n$ *Credits2* for any n by using the product encoding and Theorem 4. ⊠

Theorem 4 identifies conditions where we can use standard techniques for verification of safety formulas. We use this in Theorem 5 to formulate checking for $P_{\preceq}(G,H)$ as a symbolic bounded model checking problem.

Theorem 5. *Assume that G is input-robust and H is output-robust. There is an effective procedure that given G, H and a bound $n > 0$, creates a formula $BNC(G,H,n)$ of size $O(n(|G| + |H|))$ with free variables ℓ_i for $i < n$, such that $BNC(G,H,n)$ is satisfiable iff $G \not\preceq_n H$, and if $M \models BNC(G,H,n)$ then for some a, and $m < n$, $(\ell_0^M, \ldots, \ell_m^M, a)$ is a witness of $G \not\preceq_n H$.*

Proof. Given a GLAS G, we can characterize the set of states reachable after n steps by unfolding of the transition relation of G n times. The corresponding formula is $Reach(G,n) \stackrel{\text{def}}{=} \imath_G \wedge \bigwedge_{i=0}^{n-1} TR_G(\Sigma_G^i, \ell_i, \Sigma_G^{i+1})$ where $\Sigma_G^0 = \Sigma_G$ and $\Sigma_G^{i+1} = (\Sigma_G^i)'$. The bounded non-conformance checking formula $BNC(G,H,n)$ is now $Reach(G \otimes H, n) \wedge \bigvee_{i=0}^{n} \neg P_{\preceq}(G,H)(\Sigma_{G\otimes H}^i)$. The formula is satisfiable if and only if $P_{\preceq}(G,H)$ is violated within n steps. The size of the formula is $O(n(|G \otimes H| + |\neg P_{\preceq}(G,H)|))$. Theorem 4 ensures that it suffices to check $P_{\preceq}$ as a state invariant. ⊠

An LTS $\mathcal{L}$ is *input-enabled* if in all states in $\mathcal{L}$ that are reachable from the initial state, all input-labels are enabled.[3] The following definition of *ioco* is consistent with the definition in [18] provided that δ is part of the output labels.

Definition 18. Let $\mathcal{L}$ be an LTS and $\mathcal{M}$ an input-enabled LTS. $\mathcal{M}$ *ioco* $\mathcal{L}$ iff, for all $\mathbf{a} \in Tr(\mathcal{L})$ and output-labels a, if $\mathbf{a} \cdot a \in Tr(\mathcal{M})$ then $\mathbf{a} \cdot a \in Tr(\mathcal{L})$.

Theorem 6. *If $\lceil G \rceil$ is input-enabled then $\lceil G \rceil$ ioco $\lceil H \rceil \iff G \preceq H$.*

Proof. Assume $\lceil G \rceil$ is input-enabled. ($\Longrightarrow$): Assume $G \not\preceq H$. We show that $\lceil G \rceil$ *ioco* $\lceil H \rceil$ does not hold. From Definition 12 follows that there exists a trace $\mathbf{a}$ such that $S_G^0 \stackrel{\mathbf{a}}{\rightarrow} S'_G$ and $S_H^0 \stackrel{\mathbf{a}}{\rightarrow} S'_H$, and there is a label a such that either

[3] Such LTSs are called *input-output transition systems* in [18].

1) a is a output-label that is enabled in S'_G but not enabled in S'_H, or 2) a is a input-label that is enabled in S'_H but not enabled in S'_G. The second case cannot be true since $\lceil G \rceil$ is input-enabled. Thus, there is a trace $\mathbf{a} \in Tr(\lceil H \rceil)$ and an output-label a such that $\mathbf{a} \cdot a \in Tr(\lceil G \rceil)$ but $\mathbf{a} \cdot a \notin Tr(\lceil H \rceil)$. ($\Longleftarrow$): Assume $\lceil G \rceil$ *ioco* $\lceil H \rceil$ does not hold. We show that $G \not\preceq H$. From Definition 18 follows that there exists a trace $\mathbf{a} \in Tr(\lceil H \rceil)$ and an output-label a such that $\mathbf{a} \cdot a \in Tr(\lceil G \rceil)$ but $\mathbf{a} \cdot a \notin Tr(\lceil H \rceil)$. Now use Lemma 3. ⊠

5 Related work

The current paper generalizes the notion of model programs to GLASs and generalizes the results in [21] related to deterministic input-output model programs to GLASs. We introduced the notion of robustness as a nontrivial extension of deterministic GLASs by supporting "safe" internal nondeterminism, while retaining the property that non i/o-refinement checking reduces to safety analysis.

The literature on ioco [6,19,20] and various extension of ioco is extensive. A recent overview and the formal foundations are described in [18]. An extension of ioco theory to symbolic transition systems is proposed in [13]. Composition of GLASs is related to composition of symbolic transition systems [12]. The application of composition for symbolic analysis and formal relation to open system verification has not been studied in those contexts as far as we know. We believe that the results presented here can be used and complement the work on symbolic transition systems in [13,12].

We believe that GLASs can be used as a foundation for symbolic analysis of Event-B models [2] that is an extension of the B-method [1] with events (corresponding to labels of a GLAS) that describe atomic behaviors, where each event is associated with a guard and an assignment, that causes a state transition when the guard is true is a given state. Composition of Event-B models is discussed in [17,7].

BMPC [23], that is used in Section 4, is a generalization of SMT based bounded model checking [11] to GLASs. The notion of *i/o-refinement* of GLASs builds on the game view of systems [8], that can also be used to formulate other problems related to input-output GLASs, such as finding winning strategies to reach certain goal states.

References

1. Abrial, J.-R.: The B-Book: Assigning programs to meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae 77(1-2), 1–28 (2007)
3. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)
4. AsmL, http://research.microsoft.com/fse/AsmL/

5. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories, ch. 26. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
6. Brinksma, E., Tretmans, J.: Testing Transition Systems: An Annotated Bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–193. Springer, Heidelberg (2001)
7. Butler, M.: Decomposition structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
8. de Alfaro, L.: Game models for open systems. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 269–289. Springer, Heidelberg (2004)
9. Alfaro, L.d., Henzinger, T.A.: Interface automata. In: ESEC/FSE, pp. 109–120. ACM, New York (2001)
10. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. de Moura, L., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 438–455. Springer, Heidelberg (2002)
12. Frantzen, L., Tretmans, J., Willemse, T.: 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)
13. Franzen, L., Tretmans, J., Willemse, T.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005) (to appear)
14. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-based Software Testing and Analysis with C#. Cambridge University Press, Cambridge (2008)
15. Keller, R.: Formal verification of parallel programs. Communications of the ACM, 371–384 (July 1976)
16. Lynch, N., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, pp. 137–151. ACM Press, New York (1987)
17. Poppleton, M.: The composition of Event-B models. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 209–222. Springer, Heidelberg (2008)
18. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
19. Tretmans, J., Belinfante, A.: Automatic testing with formal methods. In: EuroSTAR 1999: 7th European Int. Conference on Software Testing, Analysis & Review, Barcelona, Spain, November 8–12, EuroStar Conferences, Galway, Ireland (1999)
20. van der Bij, M., Rensink, A., Tretmans, J.: Compositional testing with ioco. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)
21. Veanes, M., Bjørner, N.: Input-Output Model Programs. In: Leucker, M., Morgan, C. (eds.) Theoretical Aspects of Computing - ICTAC 2009. LNCS, vol. 5684, pp. 322–335. Springer, Heidelberg (2009)
22. Veanes, M., Bjørner, N.: Alternating Simulation and IOCO. Technical Report MSR-TR-2010-38, Microsoft Research (April 2010)
23. Veanes, M., Bjørner, N., Gurevich, Y., Schulte, W.: Symbolic bounded model checking of abstract state machines. Int. J. Software Informatics 33(2-3), 1–22 (2009)

Reducing the Cost of Model-Based Testing through Test Case Diversity

Hadi Hemmati[1,2], Andrea Arcuri[1], and Lionel Briand[1,2]

[1] Simula Research Laboratory
[2] Department of Informatics, University of Oslo
{hemmati,arcuri,briand}@simula.no

Abstract. Model-based testing (MBT) suffers from two main problems which in many real world systems make MBT impractical: scalability and automatic oracle generation. When no automated oracle is available, or when testing must be performed on actual hardware or a restricted-access network, for example, only a small set of test cases can be executed and evaluated. However, MBT techniques usually generate large sets of test cases when applied to real systems, regardless of the coverage criteria. Therefore, one needs to select a small enough subset of these test cases that have the highest possible fault revealing power. In this paper, we investigate and compare various techniques for rewarding diversity in the selected test cases as a way to increase the likelihood of fault detection. We use a similarity measure defined on the representation of the test cases and use it in several algorithms that aim at maximizing the diversity of test cases. Using an industrial system with actual faults, we found that rewarding diversity leads to higher fault detection compared to the techniques commonly reported in the literature: coverage-based and random selection. Among the investigated algorithms, diversification using Genetic Algorithms is the most cost-effective technique.

Keywords: Test case selection; Model-based testing; Search-based testing; Clustering algorithms; Similarity measure; Genetic Algorithms; Adaptive Random Testing; Jaccard Index; UML state machines.

1 Introduction

The idea of model-based testing (MBT) [1] is to generate executable test cases by systematically analyzing specification models (e.g. represented as UML state machines) following a test strategy such as a coverage criterion, that aims to cover certain features of the model (e.g., all transitions). MBT brings many advantages but also entails the additional cost of modeling the software under test (SUT). In addition, there are two factors that significantly increase the cost of MBT: (1) the lack of automated oracle (e.g., when assessing the subjective perception of a media quality in a videoconference system), and (2) the high cost of test case execution (e.g., when testing must be performed on actual hardware or a restricted-access network). In both situations, the test suite must be as small as possible while, to the extent possible, preserving its fault revealing power. However, for real world size models, MBT

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 63–78, 2010.

techniques usually generate large sets of test cases regardless of the applied coverage criteria. Therefore, a model-based technique is required to select an optimal subset of test cases to be executed, which is, in general, a NP-hard problem.

In similarity-based test case selection, the idea is to diversify the selected test cases with respect to a similarity measure. In [2, 3], we proposed a similarity-based selection technique for testing based on UML state machines (SMBT). We compared different similarity measures in terms of what information from the test cases they have to evaluate (test case encodings) and how this evaluation should be done (similarity functions). The results showed that, in the context of SMBT, the similarity measure that represents a test case as a set of trigger-guards [2] and uses Jaccard Index [4] as the similarity function [3] is the most effective measure in terms of fault detection rate (FDR).

In this paper, we take a deeper look into the idea of diversifying test cases and investigate why similarity-based selected test cases are effective in finding faults. We also study different strategies that, given a similarity measure and a test suite, we can use to select a subset of the test suite. We applied our experiments on an industrial software system and a set of actual faults, and the results clearly showed that rewarding diversity is effective. The main explanation is that the test cases that find different faults belong to distinct clusters based on the similarity measure. In addition we found that, among different selection strategies, Genetic Algorithms (GAs) [5] are the most cost-effective technique for similarity-based test case selection. We also have shown that, in our case study, we could save up to 80% of test case executions, and get more than 99% FDR, by using a GA compared to a coverage-based selection technique.

The rest of the paper is organized as follows. Section 2 introduces the similarity-based test case selection technique. Section 3 discusses the different strategies which are used in this paper to diversify test cases. Section 4 provides a brief overview of related works covering similarity-based selection techniques. Section 5 reports the experimentation results of applying the selection techniques on an industrial case study. Section 6 concludes the paper and outlines our future work plan.

2 Similarity-Based Test Case Selection

Unlike coverage-based selection, where the goal is maximizing the coverage of the test model by the selected test cases (e.g. transition coverage in SMBT) to maximize chances of fault detection, similarity-based selection techniques maximize diversity among the selected subset. Diversity is calculated using a (dis)similarity measure between pairs of test cases. A similarity measure is the value that a similarity function assigns to two inputs which are being compared. In a testing context, inputs are usually encoded as a set or sequence of elements. In the context of MBT, the inputs are abstract test cases defined on the test model rather than concrete test cases. We do not use the execution information of the test case as, in our context, the goal is to select them before execution. Abstract test cases are naturally generated as a first step by MBT and can hide the unnecessary information for similarity comparisons. For example, in SMBT an abstract test case representation can be a path in the state machine specifying the SUT. In general, different faults can be detected by the same test path instantiated with different test data (e.g., event parameter values). Therefore, to

compare different techniques, it is necessary to run the selected test paths with different input data and analyze their FDR distribution.

Representation (encoding) of the test cases has an important effect on the similarity measure. Though in SMBT a test path represents an encoded abstract test case, the test path can be described at different levels of details. In [2], we studied three encodings for a test path in UML state machine: state-based, transition-based, and trigger-guard-based, and reported that trigger-guard-based encoding is the most effective one in terms of fault detection, where a test path(*tp*) is represented as:

$$<\text{tp}> \quad ::= <\text{TrGu}> \mid <\text{TrGu}> \text{","} <\text{tp}>$$
$$<\text{TrGu}> ::= \textit{trig} \mid \textit{guard} \mid \textit{id} \mid \textit{guard}\text{"+"}\ \textit{trig}$$

where *trig* is the identification of a trigger, and *guard* is the identification of a guard in the state machine. In this representation, a transition is identified by its trigger and guard. It can be only a trigger, or a guard or both together. If there is a transition with no guard and trigger, we use the transition id (*id*) as its identifier.

Given an encoding, one may use different similarity functions to calculate the similarity value. In [3] we studied different set-based and sequence-based similarity functions and proposed Jaccard Index as the most cost-effective. Given a set of *n* encoded test cases (s_n) and a similarity function (*SimFunc*), the test case selection problem is reformulated as minimizing $SimMsr(s_n)$:

$$SimMsr(s_n) = \sum_{tp_i, tp_j \in s_n \wedge i>j} SimFunc(tp_i, tp_j)$$

where $SimFunc(tp_i, tp_j)$ returns the similarity of two test paths (or other encoded abstract test cases in MBT) in s_n represented by tp_i and tp_j. The last step in similarity-based selection is using a strategy to select a subset of test cases with minimum average pair-wise similarity (*SimMsr*). In the rest of this paper, we focus on finding the best strategy for this selection.

3 Strategies for Maximizing Diversity

Given a similarity measure we have two strategies to select the most diverse test cases. One is based on clustering test cases and taking samples from each cluster and the second is searching for the most diverse subsets. In this section, we introduce one clustering and two search techniques that will be investigated.

3.1 Clustering-Based Techniques

Clustering algorithms divide data instances into natural groups by maximizing their internal homogeneity and external separation [6]. Regardless of the specific algorithm which is used for clustering, most clustering techniques use a proximity measure as a mean to determine the closeness (similarity), or dissimilarity (distance) between pairs of instances and pairs of clusters.

In this study, we are using one of the simplest clustering algorithms, which has been frequently used in software engineering, including software testing [7]: Agglomerative Hierarchical Clustering (AHC) [6]. AHC starts with forming clusters containing

each exactly one object (a test case in this study). A sequence of merge operations is then performed until the desired number of clusters is achieved. At each step, the two most similar clusters will be joined together. The measure that we used for assessing similarity between two clusters, inter-cluster similarity, is Average Linkage and it is defined as the average of all pair-wise similarities between all instances of those two clusters [6]. After applying clustering, we need a sampling technique for selecting one or more test case per cluster. We use one-per-cluster sampling where the number of clusters is the same as the selected sample size and then randomly select one member from each cluster. The pseudo-code of the employed AHC follows:

(1) Make one cluster (C_k) per test path (tp_i).
(2) While the number of clusters is more than *sampleSize*
(3) Find the two most similar clusters C_x and C_y (with the maximum $InterClusterSim(C_x, C_y)$). Where:

$$InterClusterSim(C_x, C_y) = \frac{\sum_{tp_i \in C_x \wedge tp_j \in C_y} SimFunc(tp_i, tp_j)}{|C_x| * |C_y|}$$

(4) Merge the two clusters.

3.2 Test Case Selection Using Adaptive Random Testing

Another technique that we investigate is Adaptive Random Testing (ART), which has been proposed as an extension to Random Testing [8]. Its main idea is that diversity among test cases should be rewarded, because failing test cases tend to be clustered in contiguous regions of the input domain. This has been shown to be true in empirical analyses regarding applications whose input data are of numerical type [8]. Recently, Object-Oriented software has been also shown to manifest such a property [9]. Therefore, ART is a candidate selection strategy in our context as well. In this paper, we use the basic ART algorithm described in [8], but we ensure that no replicated test case is given in output. The pseudo-code for ART is:

(1) Z={}
(2) Add a random test case to Z
(3) Repeat until |Z|= *sampleSize*
(4) Sample K random test cases that are different from Z
(5) For each of these test cases k
(6) k.maxSim = max(SimFunc(k , z ∈ Z))
(7) Add the k with minimum maxSim to Z

3.3 GA-Based Test Case Selection

A GA [5] is used in this paper since the nature of our problem, which is a form of optimization, resembles typical problems addressed in search-based software engineering [10]. GAs are the most used and successful reported technique in this domain [10] and rely on four basic features: population, selection, crossover and mutation. More than one solution is considered at the same time (population). At each

generation (i.e., at each step of the algorithm), some good solutions in the current population, selected by the selection mechanism, generate offspring using the crossover operator. This operator combines parts of the chromosomes (i.e., the solution representation) of the offspring with a certain probability; otherwise it just produces copies of the parents. These new offspring solutions will fill the population of the next generation. The mutation operator is applied to make small changes in the chromosomes of the offspring. Eventually, after a number of generations, an individual that solves the addressed problem will be evolved.

In this paper, we use a steady state GA as a selection technique, in which only the offspring that are not worse than their parents are added to the next generations. An individual in our context is a subset of size n from the original test suite (denoted s_n). Given a similarity function $SimFunc(tp_i , tp_j)$, the fitness function f to minimize is the sum, for all pairs (tp_i , tp_j) in s_n, of $SimFunc(tp_i , tp_j)$, denoted $SimMsr$. We use a single point crossover with probability of P_{xover} to combine two different parents s_n^x and s_n^y. A mutated test path is replaced by a test path that is selected at random from the set of all possible test paths. A valid solution is a set of test cases in which there is no duplicate. We have applied two types of stopping criteria for the GA in this study: (1) stopping after specific number of fitness evaluations, and (2) stopping after a fixed period of time (e.g., 350ms). The pseudo-code of the employed GA follows:

(1) Sample a population G of m sets of test cases uniformly from the search space (i.e., the set of all possible valid sets with a given size n)
(2) Repeat until the stopping criterion is met
(3) Choose s_n^x and s_n^y from G
(4) $(\acute{s}_n^x , \acute{s}_n^y) := \text{crossover } (s_n^x , s_n^y, P_{xover})$
(5) $\text{Mutate}(\acute{s}_n^x, \acute{s}_n^y)$
(6) If valid $(\acute{s}_n^x , \acute{s}_n^y) \wedge \min (f(\acute{s}_n^x), f(\acute{s}_n^y)) \leq \min (f(s_n^x), f(s_n^y))$
(7) Then $s_n^x := \acute{s}_n^x$ and $s_n^y := \acute{s}_n^y$

4 Related Work

There are three approaches reported in the literature to select a subset of test cases from a test suite that can be applied in our context: (1) Random or semi-random selection [11], where there is no guidance to select test cases; (2) Coverage-based selections, where we hypothesize that "the test cases which have more coverage are more likely to detect faults" (e.g., in [12] a Greedy search selects, at every step, the test case that covers the most uncovered statements whereas in [13, 14] a GA is used to find the maximum coverage.); (3) Similarity-based selections try to diversify test cases, given a similarity measure, assuming that maximizing diversity among the selected test cases maximizes the number of detected faults.

Diversifying test cases has been studied on code-based test case selection and mostly in the context of regression testing. The similarity measure in such cases is usually based on code coverage [7, 15-18]. In [19] a sequence of memory operations is used to calculate the similarity and in [20] the authors use the whole test script in string format as the input for similarity function. The work in [21] is the only one where the similarity function is based on model-level information. Test cases are

represented as sequence of transitions in a LTS model of the system and the number of identical transitions in the sequence is the similarity function. Our similarity measure is different from theirs, both in terms of encoding and similarity function—we use trigger-guard sets on UML state machines and apply the Jaccard Index. In [2, 3] we have compared the effectiveness of our similarity measure with the measure in [21] and the results showed a great improvement using our technique, which therefore is applied in this study as well. Given a similarity measure, different strategies have been used to diversify the selected subsets: Greedy search in [17, 19-21], Neural network based classification in [18], ART in [17], AHC in [7, 15, 16]. In this paper, using our similarity measure, we compare ART, AHC, and a GA.

5 Empirical Evaluation

5.1 Case Study Description

The SUT under study is a typical safety monitoring component in a safety-critical control system implemented in C++ and modeled as UML state machines complemented by constraints specifying state invariants and guards. This SUT is typical of a broad category of reactive systems interacting with sensors and actuators. The first and the subsequent maintained versions of the system (including models and code) were developed and verified by company experts and our research team. The correct and the most up-to-date UML state machine, representing the latest version of the SUT's behavior, consists of one orthogonal state, 17 simple states, six simple-composite states, and a maximum hierarchy level of two. The unflattened state machine contains 61 transitions and the flattened state machine consists of 70 simple states and 349 transitions.

The correct latest UML state machine was given to our test case generation tool (TRUST) [22] as an input model. Using All-Transitions coverage, 281 test paths and corresponding executable test cases were automatically generated. In our case study, if a test path has the ability to detect a fault, it can be detected by any valid test data for that test path. Therefore, in our experiment, we have one test case per test path and the FDR of a test path is equal to the FDR of the corresponding test case. As it is typical in many embedded systems, the average execution time for these test cases is in the order of minutes, which makes running all the 281 test cases very time consuming.

We use 15 faulty versions of the code that are made by introducing one real fault per program. The 15 faults used in the study were introduced during maintenance activities by developers and re-introduced for the purpose of the experiment in the latest version of the SUT. Each of these faults belongs to one of the following categories: wrong guards on transitions, wrong state invariant, missing transition, and wrong OnEntry action in states. Among 281 test cases, 207 cannot detect any faults and 74 catch at least one fault. The average number of detected faults per test case for the 15 faulty versions is 0.72 and the maximum is five. Each fault is also detected on average by 13 test cases. There are nine faults which are only detected by three test cases and two faults are detectable by 65 test cases.

5.2 Experiment Design

In our industrial case study, we investigate the following research questions:

- **RQ1.** Why does diversifying test cases improve fault detection?
  - **RQ1.1.** Do test cases that find the same faults tend to be more similar to each other than with other test cases?
  - **RQ1.2**. Do test cases that find different faults tend to be more different from each other than test cases that find the same faults?
- **RQ2.** What is the most cost-effective way to diversify (given our similarity measure) a set of test cases?
  - **RQ2.1.** Does clustering-based test case selection improve the average FDR compared to coverage-based and random selection?
  - **RQ2.2.** Are search-based techniques more cost-effective than clustering-based selection in terms of fault detection?
- **RQ3.** How cost-effective is diversifying test cases compared to state of practice techniques for test case selection?

In RQ1 we are analyzing why diversifying test cases improves FDR. In other words, are test cases distinctly clustered with respect to different faults? We have carried out an exhaustive analysis based on our industrial case study. Given N=281 test cases, we ran all of them on the actual SUT and all its faulty version to check which of the M faults they are able to detect (in our case study M=15). We then calculated the similarity of each pair of test cases, for a total of N*(N-1)/2 pairs. Note that the exhaustive analysis of the search space landscape is based on the similarity values of all test case pairs. However, test case selection is performed for any arbitrary *sampleSize* where using an exhaustive search is not an option, since the search space size for selecting a subset of size *sampleSize* is equal to the number of possible *sampleSize* combinations within a test suite of a given size. In our case, as an example, the search space size for *sampleSize* =28 (~10% of the test suite) is 2.9*10^38.

To address RQ1, we investigate two hypotheses: (1) For each fault cluster, the similarity between pairs of test cases that find the same faults is, on average, significantly higher than the similarity of other test case pairs in the test suite, and (2) For each pair of fault clusters, the similarity between test cases that find different faults is significantly lower than the similarity of test case pairs that find the same fault in the test suite. If hypothesis (1) holds, then test cases finding the same faults will cluster in close areas of the test case space. As a result, rewarding diversity in test case selection would be beneficial. But hypothesis (2) should also hold, otherwise diversity might be harmful since we would need more than one test case from the same area to detect all faults.

In RQ2, we are interested in how to diversify the test cases, given the similarity measure used in RQ1. Our baselines of comparison are random selection (Rnd) and a coverage-based selection technique (CovGr) which is based on one of the most used selection techniques in the literature: it applies a Greedy search to maximize the coverage of the selected test cases [12]. In this paper, in each step of the Greedy search in CovGr, we look for the test cases which cover the most yet uncovered transitions on the UML state machine representing the SUT. Finally, in RQ3 we look at the practical benefits of our proposed approach based on our industrial SUT. In this study, as

mentioned in Section 3, AHC is used as our clustering algorithm and a GA and ART as search-based techniques. Our measure of effectiveness is the FDR of the selected subset from the original test suite. Ideally, given the same amount of computational cost, we would say that a technique is better than the other if it obtains higher average FDR. For practitioners, such cost would typically be measured as the time that an algorithm takes before completing its task. Comparing algorithms using time is not a robust option from a practical standpoint though. Low-level implementation details may have a strong effect on computational time. If we use time as stopping criterion, then we may not truly compare algorithms but instead their implementations [23]. To cope with this problem, a measure that is independent from implementation details would be useful. For example, when comparing search algorithms, it is a common practice to allow each algorithm to run until a maximum number of fitness evaluations is executed (e.g., 100,000 [24]). However, the assumption here is that the total search cost is proportional to the number of fitness evaluations and the cost of other operations than fitness evaluation is either equal or negligible in both algorithms.

To compare GAs with ART, following the same general reasoning, we use the number of similarity comparisons (C) as stopping criterion, where *n* is the size of the output test case set. We hence can run both the GA and ART with the same preset number of similarity comparisons. For a GA that runs for W fitness evaluations (each consisting of Q similarity comparisons), we have that C(GA) = W * Q = W * n * (n-1)/2 whereas for ART we have [8]: C(ART) = K * n * (n-1)/2.

We would like to run both ART and the GA such that C(ART)=C(GA), but that might not be possible because K (the size of the candidate set in ART) is a constant that is upper bounded by N (281 in our case). In other words, the basic ART cannot be run for an arbitrary amount of computational resources as it is the case for GAs (for which we can choose arbitrarily high values for W). To cope with this problem, we can just run ART several independent times (e.g., J times), and then take the best result out of these J runs. Therefore, to obtain fair comparisons using similarity measures, we can simply enforce W=J*K.

Whenever we could not use a fair metric (as the number of fitness evaluations) for comparing different algorithms for test selection, we used the time expressed in milliseconds as stopping criterion, which is the time spent by our implementation of the algorithms on a PC with Intel Core(TM)2 Duo CPU 2.40 Hz and 4 GB memory running Windows 7. As we previously discussed, though this is not particularly robust in general, it is a reasonable option in our context as a significant effort was made to optimize implementations and the execution environment was stable.

To account for the randomness of the results, which exists for all selection algorithms, we ran each experiment 100 times and analyzed distributions. We report the results for different techniques for sample sizes less than 140 (~50% of the test suite) with intervals of 10, since our focus is, for practical reasons, on smaller size subsets. (In practice, test case selection is mostly used for selecting a relatively small sample of large test suites.) Furthermore, for large sample sizes, all selection techniques will usually be as good as random selection and typically detect most faults. We have performed non-parametric (Mann-Whitney U-test) statistical tests, using a significance level of 0.05, to compare the FDR distribution of the proposed and alternative selection techniques. Non-parametric tests are more robust than a parametric test (e.g.,

the *t*-test) when there are strong departures from normality and for large enough samples, as this is the case in this study (100 observations).

5.3 Experiment Results

5.3.1 Why Does Diversifying Test Cases Improve Fault Detection?

For each of the M=15 faults, we calculated the similarity of the test case pairs that both found each of these faults (groups of test case pairs, from F1 to F15). Mann-Whitney U-tests were performed (α =0.05) to see whether there was a difference in similarity value between the pairs in F1 to F15 and the set of all remaining pairs of test cases (T - Fi). Table 1 summarizes the results where bold median values represent statistically significant differences between the distributions of these Fi with T - Fi. Note that F1 and F2, F3 and F4, and F7 to F15 are on the same table row, as they have the same descriptive statistics. This is due to the fact that most test case pairs are the same and those that are not the same have high similarity values (according to our similarity measure).

The results show that the difference is significant for the first six groups. The other groups also show a high difference in terms of mean and median but, since there are only three observations for each of those groups, we cannot get statistically significant differences. Therefore, the first hypothesis of RQ1.1: "Test cases that find the same faults tend to be more similar to each other than with other test cases" is confirmed.

To investigate RQ1.2, for each pair of fault clusters Fi and Fj, let us consider the similarity distribution (Dd) of test case pairs which belong to two different clusters, i.e., test cases that find different faults. We compare Dd with the similarity distribution (Ds) of test case pairs which both are in one of those two clusters, i.e., test cases that find the same fault. The median of Dd and Ds per cluster pair is reported above the diagonal in Table 2.

There are cases where fault clusters Fi are exactly the same, i.e., their respective faults are found by exactly the same set of test cases. Distinguishing them does not have any effect on the FDR results (either all or none of the faults will be revealed by a selected set of test cases) and therefore such clusters are not distinguished. As a result, there are seven distinct fault clusters (labeled as A to G) matching the columns and rows of Table 2. Their mapping to the 15 fault clusters is as follows: A(F1 and F2), B(F3 and F4), C(F5), D(F6), E(F7 to F9), F(F10 to F12), G(F13 to F15).

The bold values show the cases where there is a statistically significant difference between Dd and Ds, based on a Mann-Whitney U-test. The presence of significant differences support the claim that fault clusters are far away from each other and therefore that rewarding diversity is useful. In cases where two clusters are overlapping, the size of the overlap compared to the size of their union will determine whether rewarding diversity is harmful. If the ratio of the overlapping part (intersection) over the union is high, a test case that finds one of the two faults would have a high probability of finding the other. In this case, rewarding diversity is still a reasonable option. We measure this ratio by dividing the size of two clusters' intersection $|I|$ by the size of their union $|U|$: $I_U=|I|/|U|$. The cells below the diagonal of Table 2 report this measure per cluster pair.

Among 21 cluster pairs, 15 contain distinct clusters with significant differences between Dd and Ds. There are three clusters (E, F, and G) that only contain a few test

cases (three per cluster), which are not amenable to statistical analysis and show no statistically significant differences. Clusters B and D which are not significantly different from each other show a high overlapping value (0.57), implying that although these clusters are not distinct, there is a 57% probability that a test case that is selected from their union can find both faults. Two cluster pairs, <A,D> and <C,D>, show unexpected results—Dd median lower than the Ds median—and they are not highly overlapping. Therefore, since among 21 pairs, 15 pairs fit the situation where similarity-based selection is effective, two do not, and four are neutral, we can conclude that, overall, in most cases "test cases that find different faults tend to be more different from each other than test cases that find the same faults".

Table 1. Min, max, median, mean, and standard deviation of similarity values of the test cases that find the same faults

Groups	Pairs	Min	Median	Mean	Max	SD
T	39340	0.076	0.250	0.291	1.000	0.166
F1,F2	2080	0.181	**0.4**	0.432	1.000	0.173
F3,F4	91	0.375	**0.571**	0.561	0.833	0.143
F5	28	0.200	**0.464**	0.475	0.800	0.168
F6	28	0.714	**0.714**	0.714	0.714	0.000
F7 to 15	3	0.375	0.428	0.434	0.500	0.062

Table 2. Each cell above the diagonal shows the median of Dd and Ds (Dd/Ds) and each cell below the diagonal shows the overlapping measure (I_U), per cluster pairs. Bold median values highlight significant differences (Mann-Whitney U-test) between the Dd and Ds.

	A	B	C	D	E	F	G
A	-	**0.33/0.42**	**0.33/0.40**	**0.71/0.40**	**0.18/0.40**	**0.18/0.40**	**0.18/0.40**
B	0.21	-	**0.37/0.57**	**0.71/0.66**	**0.37/0.57**	**0.37/0.57**	**0.37/0.57**
C	0.12	0	-	0.71/0.71	**0.37/0.42**	**0.37/0.42**	**0.37/0.42**
D	0.12	0.57	0	-	**0.11/0.71**	**0.11/0.71**	**0.11/0.71**
E	0	0	0	0	-	0.37/0.42	0.37/0.42
F	0	0	0	0	0	-	0.37/0.42
G	0	0	0	0	0	0	-

Overall, the results of our analysis confirm that diversity in test case selection should be encouraged and that our similarity measure is adequate. It also seems that since test cases finding the same faults are clustered together and these clusters are mostly distinct, clustering algorithms are a reasonable candidate approach to achieve diversity, though we will investigate what is the best strategy in the next research question.

5.3.2 What Is the Most Cost-Effective Way to Diversify a Set of Test Cases?

To answer RQ2 we first compare the AHC clustering algorithm with CovGr and Rnd introduced in Section 5.2. Fig. 1 shows the FDR results of the algorithms.

Overall, the results show that for all sample sizes AHC is more effective than its two alternatives except that for sample sizes less than 30 (~10% of the test suite) the difference between the average FDRs of CovGr and AHC is not statistically significant (based on Mann-Whitney tests). Considering the fact that in practice the results

for smaller sample sizes are more important, AHC may not be preferred to CovGr given the high cost of a clustering technique compared to simple Greedy search. On average (for all sample sizes over 100 runs) each selection requires 350ms, 10ms, and less than 1ms when using AHC, CovGr, and Rnd, respectively. Though those time differences may not seem relevant, they may become so on much larger test suites of thousands of test cases. However, for sample sizes higher than 40, there is a huge (up to 40%) improvement using AHC compared to CovGr. In addition, AHC ensures 100% FDR with 80 test cases whereas CovGr and Rnd find less than 95% of the faults even with 140 test cases.

Note that, in theory, since Rnd does not use any heuristic to increase FDR, we cannot improve it. However, we can improve CovGr by running it several time with different random selections, wherever the coverage among alternative test cases is equal, and reporting the best result out of all runs. To compare the FDR results of CovGr when it costs exactly the same as AHC, we let CovGr improve its results by random reselection and stopped the algorithm after 350ms. The results showed that in our case, there is no practically significant difference in CovGr FDR for 10 and 350 ms of running time.

Addressing RQ2.1, given that the FDR of AHC is always equal or superior to that of CovGr or Rnd, and the fact that we cannot predict for a given test suite the sample size threshold above which AHC will be certain to fare significantly better, we favor the systematic use of AHC over CovGr and Rnd. Moreover, in practice, this strategy makes even more sense when considering that test case execution time (which in our case is in the range of minutes) is usually much higher than selection time for any of the techniques (which in our case is in the range of milliseconds).

Comparing search-based techniques with AHC, first we need to find out which search technique is more cost-effective. In this study, we compare the FDR of ART and a GA. The GA is stopped after 10,000 fitness evaluations, and ART is run 1000 times with K=10 (so both algorithms use the same number of similarity comparisons). Fig. 2 shows the average FDR of the techniques for each sample size over 100 runs. In general, the GA fares better and more particularly so from sample size 20 (~7% of the test suite) to 70 (~25% of the test suite). For sample sizes larger than 70, the FDR of both techniques converges to 1.0. The differences for smaller sample sizes are statistically significant but, because these differences may not practically significant (at most 10% improvement for the GA), we need to look closely at the relative cost of ART and the GA.

As we mentioned earlier, the number of fitness evaluations is usually a good platform-independent measure for the cost of search techniques. However, in our implementation, a matrix made of all pair-wise similarities is created before any search. Therefore, this overhead is the same for all search algorithms and the fitness evaluation is not an expensive part of the search. Therefore, we cannot be sure that total cost is proportional to the number of fitness evaluation. In Fig. 3, we have plotted the actual time spent by the two algorithms (ART and the GA with 10,000 fitness evaluations). The required time for 10,000 fitness evaluations using both techniques is exponentially increasing and they both spend almost the same time for very small sample sizes (less than 20). For sample sizes higher than 20 (~7% of the test suite), ART quickly gets more expensive than the GA. Given that it always has equal or worse FDR results, there is no reason for choosing ART over the GA.

In the next step, we compare AHC with the GA but using the same execution time that AHC requires for its selection (350ms). Fig. 4 shows that the GA is clearly preferable over AHC considering that spending the same time as AHC, the GA fares in general much better and can almost double the average FDR results of AHC for small sample sizes. It is also more effective in finding all faults: AHC requires 80 test cases whereas the GA only needs 40 test cases to achieve 100% FDR. To draw more conservative conclusions regarding the superiority of the GA, we even conducted another experiment and ran AHC a relatively long time (20,000ms) to compare its FDR result with the GA using 350ms (Fig. 4). However, even when letting AHC work for almost 60 times longer than the GA it still yields much lower FDR. Therefore, our suggested answer to RQ2 is using the GA over the other alternatives.

5.3.3 How Cost-Effective Is Diversifying Test Cases Compared to State of Practice Techniques for Test Case Selection?

To answer RQ3, we compare our best candidate based on RQ2, which is similarity-based selection using a GA, with a coverage-based Greedy search (CovGr). Looking at Fig. 1, the first observation is that the GA can save more than 80% of the test case execution cost, given the fact that the GA, on average, finds more than 99% of the faults by 40 test cases whereas CovGr requires more than 220 test cases to achieve the same (not plotted in the figure). To have a more detailed cost-effectiveness assessment, we look at the improvement that the GA may provide over time. Since this improvement varies over sample sizes as well, we plotted in Fig. 5 the percentage of FDR improvement provided by the GA over CovGr for five sample sizes: 10, 15, 20, 25, and 50 (ranging from 4 to 18 % of the test suite), over a time period of 10ms (the average time required by CovGr to select test cases) to 350ms (the average time required by AHC to select test cases). Note that as we mentioned earlier, as opposed to the GA, CovGr does not improve over time.

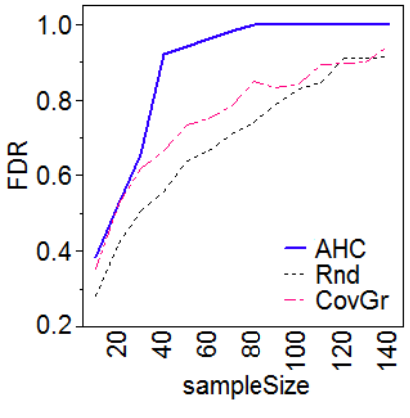


Fig. 1. The average FDR of AHC, Rnd, and CovGr for different sample sizes

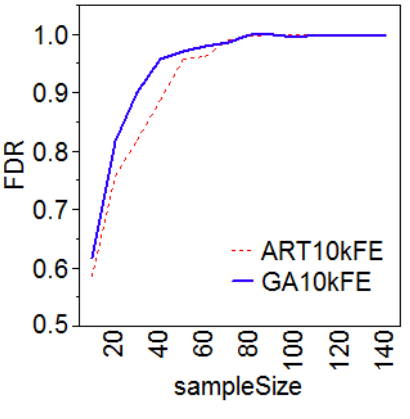


Fig. 2. The average FDR of ART and the GA with 10,000 fitness evaluations for different sample sizes

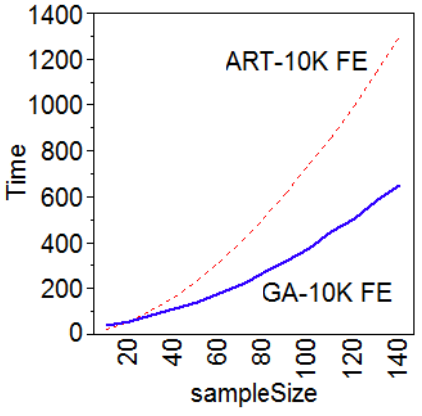


Fig. 3. The time in milliseconds required by the GA and ART to run 10,000 fitness evaluations for different sample size

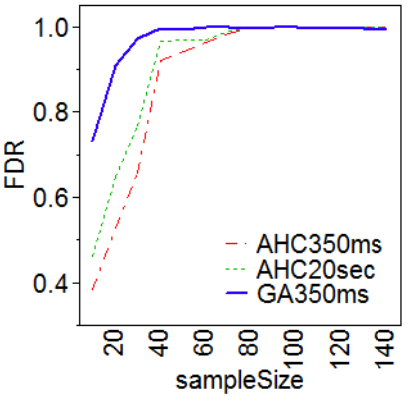


Fig. 4. The average FDR of the GA with 350ms and AHC with 350 and 20,000ms for different sample sizes

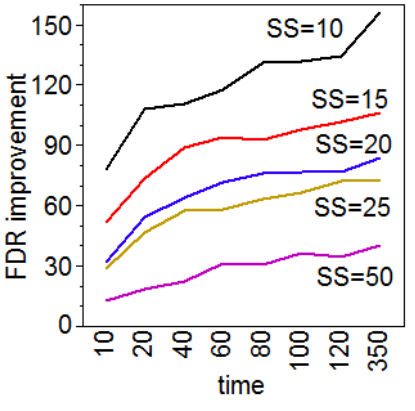


Fig. 5. The percentage of improvement of similarity-based selection using GA over CovGr for different sample sizes (SS) in time range of 10 to 350ms

A first observation from Fig. 5 is that the smaller the sample size, the larger the improvement provided by the GA. Also, it is interesting to see that the GA, even with 10ms execution time, always detects more faults than CovGr. For example, the average FDR of the GA is 80% larger than the CovGr FDR for 10ms. Finally, a cost analysis shows that in cases where we can afford spending more time for selection, the GA can be greatly improved. For all five sample sizes shown in Fig. 5, the GA's improvement over CovGr almost doubles if we give it 350ms instead of 10ms. This improvement over time gets very large when the sample size gets smaller. For example, for sample size 10 the GA can yield a 160% higher FDR than CovGr, which in practice is a great benefit given that the cost for this improvement is only 350ms for a test suite of 281 test cases where the cost of running one extra test case is in the order of minutes.

6 Discussion on Validity Threats

The main threats to the validity of this study are firstly the fairness of comparisons in terms of cost and secondly the generalizability of the results.

Similarity comparisons of test cases and clusters are the most influential part of selection techniques. In our implementations of the algorithms, all pair-wise similarities are pre-calculated in a similarity matrix which is given to the selection algorithm as an input parameter. Obviously, this implementation is not scalable and the similarity matrix will face memory limitations for large test suites. However, if we can afford pre-calculation, then the most expensive part of the search algorithms may not be the fitness evaluation anymore. We can see its effect on comparing ART and the GA where having the same number of similarity evaluations ART requires much more time. We have not studied on-demand similarity calculations, which might give different FDR results using the same stopping time. In addition, inter-cluster similarity calculation in AHC is very expensive and in our implementation it is repeated for each iteration of the algorithm. The code can be optimized by caching the similarities between clusters in each iteration and in the next iteration only calculate the similarities if it is not already available. However, implementing this improvement is not trivial since saving similarities of all combinations of clusters in all iterations may be not possible due to memory limitations. There is a tradeoff to be made between memory cost and execution speed.

The second issue is due to the fact that all our results and conclusions rely on a single industrial case study using a given set of real faults. Though running such studies is time consuming, it must obviously be replicated. However, as discussed earlier, the system used here is typical of a broad category of industrial systems: control systems with state-dependent behavior, controlling sensors and actuators.

7 Conclusion and Future Work

In practice, executing test cases generated by model-based testing (MBT) techniques is costly. This cost is due to the large test suites which are typically generated by MBT tools on industrial-scale systems to systematically achieve a coverage/adequacy criterion. However, for system level testing, in many situations testing should take place on the deployment platform where the cost (time and resource) of each test execution may be high. This may be due to the cost of using actual hardware, potential damages in case of failure, or access to restricted infrastructure (e.g., test network). In addition, for many systems, automatically generating oracles from models is very difficult or impossible. In such cases, test cases should be evaluated manually, greatly increasing the cost of test execution and analysis. In cases such as the ones mentioned above, one must execute a subset of the generated test suite whose size is dependent on context. In this paper, we propose a new approach for test case selection from UML state machines, by maximizing the diversity of the selected test cases. To measure diversity we used a specific test case representation for UML state machines (triggers-guards sets), which should be adapted in case of using other models, and a model-independent similarity function (Jaccard Index). We investigated why diversifying test cases with respect to our similarity measure increases fault detection rates and compared different strategies to diversify the test cases: Clustering, Adaptive

Random Testing, and Genetic Algorithms (GAs). The results of our study on an industrial software system and actual faults showed that: (1) rewarding diversity leads to finding more faults, (2) our proposed similarity-based selection (using Jaccard Index on the set of trigger-guards with a GA selection) is the most cost-effective approach compared to the other alternatives. In addition, we showed that in practice this approach can reduce the cost of test case execution in MBT by selecting a small set of test cases which can find all (or most) faults in short amount of time. In the future, we plan to replicate the study on another industrial system. In addition, we will evaluate alternative optimization and search techniques.

References

1. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, San Francisco (2006)
2. Hemmati, H., Briand, L., Arcuri, A., Ali, S.: An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study. In: 18th ACM International Symposium on Foundations of Software Engineering, FSE (2010)
3. Hemmati, H., Briand, L., Arcuri, A.: Investigation of Similarity Measures for Model-Based Test Case Selection. Simula Research Laboratory, Technical Report (2010-05) (2010)
4. Teknomo, K.: Similarity Measurement, `http://people.revoledu.com/kardi/tutorial/Similarity`
5. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional, Reading (2001)
6. Xu, R., Wunsch II, D.C.: Survey of Clustering Algorithms. IEEE Transactions on Neural Netwoks 16, 645–678 (2005)
7. Yoo, S., Harman, M., Tonella, P., Susi, A.: Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: 18th ACM International Symposium on Software Testing and Analysis, ISSTA (2009)
8. Chen, T.Y., Kuoa, F.-C., Merkela, R.G., Tseb, T.H.: Adaptive Random Testing: The ART of test case diversity. Journal of Systems and Software 83, 60–66 (2010)
9. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: ARTOO: Adaptive Random Testing for Object-Oriented Software. In: 30th IEEE International Conference on Software Engineering (ICSE) (2008)
10. Harman, M.: The Current State and Future of Search Based Software Engineering. In: Future of Software Engineering, pp. 342–357. IEEE Computer Society, Los Alamitos (2007)
11. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional, Reading (1999)
12. Elbaum, S.G., Malishevsky, A.G., Rothermel, G.: Test Case Prioritization: A Family of Empirical Studies. IEEE Transactions on Software Engineering 28, 159–182 (2002)
13. Li, Z., Harman, M., Hierons, R.M.: Search Algorithms for Regression Test Case Prioritization. IEEE Transactions on Software Engineering 33, 225–237 (2007)
14. Ma, X.Y., Sheng, B.K., Ye, C.Q.: Test-Suite Reduction Using Genetic Algorithm. In: Cao, J., Nejdl, W., Xu, M. (eds.) APPT 2005. LNCS, vol. 3756, pp. 253–262. Springer, Heidelberg (2005)
15. Leon, D., Podgurski, A.: A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases. In: 14th IEEE International Symposium on Software Reliability Engineering, ISSRE (2003)

16. Masri, W., Podgurski, A., Leon, D.: An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows. IEEE Transactions on Software Engineering 33 (2007)
17. Jiang, B., Zhang, Z., Chan, W.K., Tse, T.H.: Adaptive random test case prioritization. In: 25th IEEE/ACM International Conference on Automated Software Engineering, ASE (2009)
18. Simão, A.d.S., Mello, R.F.d., Senger, L.J.: A Technique to Reduce the Test Case Suites for Regression Testing Based on a Self-Organizing Neural Network Architecture. In: 30th Annual International Computer Software and Applications Conference, COMPSAC (2006)
19. Ramanathan, M.K., Koyutürk, M., Grama, A., Jagannathan, S.: PHALANX: a graph-theoretic framework for test case prioritization. In: 23rd Annual ACM Symposium on Applied Computing (2008)
20. Ledru, Y., Petrenko, A., Boroday, S.: Using String Distances for Test Case Prioritisation. In: 24th IEEE/ACM International Conference on Automated Software Engineering, ASE (2009)
21. Cartaxo, E.G., Machado, P.D.L., Neto, F.G.O.: On the use of a similarity function for test case selection in the context of model-based testing. In: Software Testing, Verification and Reliability (2009)
22. Ali, S., Hemmati, H., Holt, N.E., Arisholm, E., Briand, L.: Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies. Simula Research Laboratory, Technical Report (2010-01) (2010)
23. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation. IEEE Transactions on Software Engineering, Special issue on Search-Based Software Engineering, SBSE (in press, 2010)
24. Harman, M., McMinn, P.: A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search. IEEE Transactions on Software Engineering 36, 226–247 (2010)

Built-In Data-Flow Integration Testing in Large-Scale Component-Based Systems*

Éric Piel, Alberto Gonzalez-Sanchez, and Hans-Gerhard Gross

Software Technology Department, Delft University of Technology, The Netherlands
{e.a.b.piel,a.gonzalezsanchez,h.g.gross}@tudelft.nl

Abstract. Modern large-scale component-based applications and service ecosystems are built following a number of different component models and architectural styles, such as the data-flow architectural style. In this style, each building block receives data from a previous one in the flow and sends output data to other components. This organisation expresses information flows adequately, and also favours decoupling between the components, leading to easier maintenance and quicker evolution of the system. Integration testing is a major means to ensure the quality of large systems. Their size and complexity, together with the fact that they are developed and maintained by several stake holders, make Built-In Testing (BIT) an attractive approach to manage their integration testing. However, so far no technique has been proposed that combines BIT and data-flow integration testing. We have introduced the notion of a virtual component in order to realize such a combination. It permits to define the behaviour of several components assembled to process a flow of data, using BIT. Test-cases are defined in a way that they are simple to write and flexible to adapt. We present two implementations of our proposed virtual component integration testing technique, and we extend our previous proposal to detect and handle errors in the definition by the user. The evaluation of the virtual component testing approach suggests that more issues can be detected in systems with data-flows than through other integration testing approaches.

1 Introduction

The component paradigm and the service paradigm advocate to the rapid construction of large-scale systems-of-systems. Both help facilitate the integration of third-party building blocks through fostering loose coupling, and ameliorating system maintenance to the extent that it can be carried out online. Many large-scale systems-of-systems, such as situational awareness systems, support-systems of all kinds, swarm robotics, and distributed sensor and actuator networks, employ powerful data sharing and event processing techniques and middleware platforms. Such event- and data-driven systems or parts thereof are more naturally

* This work has been carried out as part of the Poseidon project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 79–94, 2010.

expressed through event processing, data-flow processing, or message-driven architectural styles, implemented on top of, or being part of the component and service platforms. The size and complexity of these large systems, together with the fact they are developed and maintained by multiple parties, make the quality assurance activities focus not only on unit testing but also on the validation of the adaptation and integration process [6,8].

Built-In Testing (BIT) [9,20,18] is a powerful method for validating the adaptation and integration of systems-of-systems of such dynamic and hybrid nature. BIT prescribes components to be equipped with the ability to check their execution environment, and the ability of being checked by their execution environment [10], before or during runtime. BIT also aims at a better maintainability of testing aspects surrounding each component. The responsibility in validating the components' environment is distributed and assigned to the components themselves which makes this method viable to assessing the integration and also the evolution of dynamic systems-of-systems.

The objective of integration testing is to uncover errors in the interaction between components or services, and their execution environment, i.e., other components and services, or the underlying middleware platform. The integration of a system must be assessed in its final context, because typically, such systems are extremely difficult to duplicate for testing. The integration must also be re-validated along with every reconfiguration, when services are replaced and reconfigured, or the system's topology is changed in any kind, in order to address evolving requirements.

Various techniques to support integration testing of systems following the event- or data-processing architectural styles exist, but in this paper we concentrate on testing data-flows as units. In earlier work, we have introduced the concept of a *virtual component* [16] that combines integration testing of data-flow-type systems with the advantages that built-in testing offers. In this article, we extend this work with the following contributions:

- We extend the component enumeration algorithm with additional functions to detect ill-formed flow definitions, allowing efficient development and maintenance of the tests.
- We present approaches for realizing the virtual component testing technique, and demonstrate how this should be done in two different platform styles, namely, client-server and publish-subscribe styles.
- We evaluate our proposed method using part of a concrete industry-scale surveillance system-of-systems [5,19].

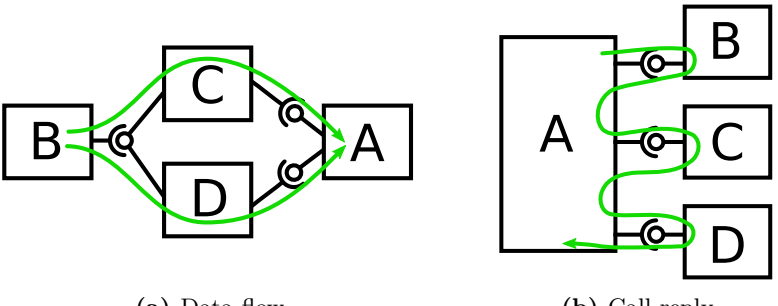
The paper is structured as follows. In Section 2, some background and related work is presented, including the concept of a virtual component. Section 3 introduces the new algorithms and methods to extend the concept of virtual component. A description of the implementation of the concept for two different component frameworks is outlined in Section 4, and the assessment of the effectiveness of virtual components for integration testing of typical data-flow-based systems is presented in Section 5. Finally, Section 6 concludes this article and presents future work.

2 Background and Related Work

Integration testing. In order to validate complex systems, one primordial step consists in performing unit testing on different levels of granularity of the system, such as *module*-level, *class*-level, *component*-level, etc. Even if every unit respects its own specification and has been successfully unit tested, there is the chance of residual system defects through component coordination issues and adaptation [1]. A common approach to ensure component integration is *integration testing* [6,8], that validates the interactions between sets of black-box components [22]. In contrast to full system testing, it concentrates on subsets of the system to be assessed in combination, allowing early checking, e.g., before all components are available.

Data-flow and call-reply architectural styles. Architectural styles [17] determine the assembly and "wiring" of components in a system, and have consequences for integration testing. Our systems of interest are aimed at high-volume data processing, and organised following the *data-flow* paradigm, also termed as "message-driven architectures", "data push technology", or "publish-subscribe architectures". They all have in common that components receive input data, process it, and generate output data for other, subsequent components in a so-called "flow". Typically processing is performed asynchronously. Another architectural style is the "call-reply" style typically found in service-oriented architectures. Here, client components are "aware" of their servers. Data is passed to the server, processed, and passed back to the client, mostly in a synchronous fashion. In data-flow styles, components do not have such "mutual awareness," which is significant for testing. Figure 1 illustrates these two styles. Both are suitable for the same application, although the implementation of their components would be different. For instance component A in the call-reply organisation must be "aware" of the component B, of the data it receives, the returned information, and even what to do next with this result.

The main advantage of the data-flow organisation is that the system architecture follows closely the data processing organisation of a physical system. This helps the designer to translate a data-flow into an implementation. It also



(a) Data-flow **(b)** Call-reply

Fig. 1. Examples of two typical architectural styles

facilitates concurrent execution of components without blocking and waiting, and components are loosely coupled, since they have no mutual behavioural expectation (contract), but only a defined data type.

Built-In Testing. Built-In Testing (BIT) is a useful paradigm in order to simplify the testing of dynamic component-based systems [18,20] and to improve the maintainability of the tests. BIT has two facets. First, components can be equipped with special ports supporting their testability, e.g., in order to control or observe internal state, or for separating testing and nominal operations [18] (test awareness). Second, the direct association of test-cases with the component [7] facilitates maintainability and traceability of test operations by keeping the test-cases and the test material closely linked together with the component [2]. This associates tests with the component throughout its life-cycle and supports re-assessment in various operation contexts. It also permits distribution of the test responsibility to the components themselves, maintaining independence of components and fostering their loose coupling, by decentralising both the definition of the tests and the testing.

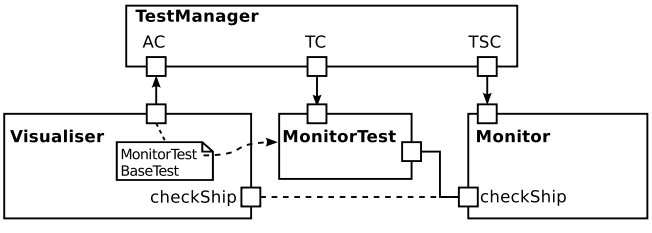


Fig. 2. Built-in Integration Testing – Example

Several approaches have been proposed to use BIT also for integration testing. In most approaches, each component carries out all or part of its integration testing itself [4,7]. The component's requirements on its execution environment (implicit execution context), and on other associated components (explicit execution context) can be validated using test-cases contained in the component, or delivered as test component in its own right [9]. We call this pattern *provider integration testing*. A typical example of such integration testing is depicted schematically in Figure 2. The Visualiser component tests its own integration with the Monitor component on which it depends. The Visualiser contacts the TestManager (one of the BIT facilities) through the Acceptance port (AC). Test Manager "knows", through the TC port, where to find the MonitorTest (a test service assessing the Monitor), and it notifies the Monitor that it will be tested, through TSC. Test results are reported back to the Visualiser. Because testing is performed from a component's perspective, it is only possible to validate the underlying platform and the components on which it depends directly. For the call-reply architectural style this kind of testing is sufficient. However, in a data-flow organisation, integration testing would be very limited.

Data-flow integration testing using virtual components. Earlier approaches were proposed for integration testing in data-flow architectural styles. Bertolino *et al.* have presented a method [3] to determine in a data-flow-based system which flows are the most relevant for assessing component integration. Their study deals with the combinations of concurrent executions of components, which is useful to determine a test goal. Unfortunately, they do not elaborate upon how this could be done.

Paul [15] describes "end-to-end" testing, a technique for assessment of system behaviour with respect to inputs and corresponding expected outputs. This is not an integration testing technique, but a system-level testing technique focusing on system use cases which does not address distribution of tests in large systems. Similarly, Jorgensen [13] describes "thread testing" as a series of inputs and expected outputs to validate the functional behaviour at the system level. It only focuses on the theoretical design of the test cases, and does not treat their implementation on any specific platform.

Another technique for integration testing relies on the generation of a large number of random input-data sequences to probe the various possible combinations of executions [21]. Each sequence is considered one test-case. Drawbacks are the sheer number of test-cases necessary to execute this technique, and that the oracle must fit any randomly generated sequence. Therefore the oracle only detects generic fault behaviour such as non-handled Java exceptions. Alternatively, the oracle can be based on a "golden" implementation, but this exists only in mature projects. There is no possibility to explicitly test typical component protocol interactions.

In our earlier work [16] to test data-flow-based component systems and services, we introduced the notion of *virtual component*, which permits the determination of one data-flow (or a part of a flow) through a system, and its representation in terms of a component in its own right, i.e. the virtual component. The data-flow may involve several physical components and represents them and their data exchange as one single "unit of high cohesion and low coupling with contractually specified ports for external communication". Testing this representation is then equivalent to integration testing the data-flow and its components associated. Here, BIT can be used to maintain tests for each flow, by associating them with the corresponding virtual component, in the same way as outlined above, in Figure 2. Consequently, tests can be managed independently in separate parts of the system. The component framework can set up the tests associated with the virtual components, execute the tester components, process and report any issues found, and maintain a history of testing along the evolution of the system.

Compared to a composite component, which is made of sub-components and is present in hierarchical component models [14], a virtual component does not influence the topology of the system. It is only a logical entity used for testing. Each virtual component is independent from any other one, so they can overlap, corresponding to the presence of several data-flows involving the same physical components. For instance, in Figure 3, two virtual components $VC1$ and $VC2$

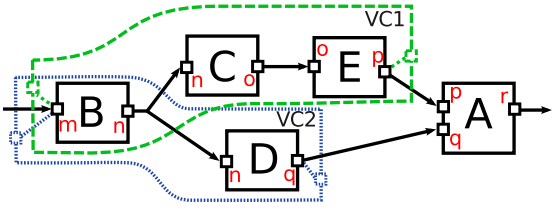


Fig. 3. Example of two virtual components

are defined to represent the two data-flows B, C, E and B, D, whose integration can be tested independently.

Another difference between a virtual component and a composite component is the way it is defined. The latter is defined by the set of enclosed components, and the connections between its interfaces and the interfaces of the sub-components. A virtual component is only defined by the connections between its interfaces and the interfaces of the sub-components (situated on the edges of the flow). The set of components it encloses is computed dynamically according to a specific algorithm. This permits to adapt easily to the evolution of the system: when a component is added or removed, or when a connection is modified, if the modifications are not on the edges of the flow, the virtual component adapts to the new data-flow automatically. This is a strong advantage in the context of large systems where the virtual components are created by the testers while the architecture is modified by the developers. For example in Figure 3, the virtual component $VC1$ is defined to represent the data-flow going from component B to E. It is defined only by the input port m of B, and the output port p of E.

The algorithm to compute the set of components enclosed in a virtual component has been defined [16] so that in the typical cases its result is "intuitive". The set C of components contained in a virtual component specified by its sets of inputs P_i and outputs P_o is computed as follow:

1. The set C_p is computed by iteratively adding all the components predecessor to P_o. For each output port, the component owner of this port is added to the set. For each newly added component, the input ports which are not in P_i are followed, and the component generating input for this port is again added to the set C_p. This is repeated until the set has not been extended.
2. The set C_s is computed similarly by iteratively adding all the components successor to P_i.
3. C is defined as $C_p \cap C_s$.

For example, the set of components in the virtual component $VC1$ of Figure 3, defined with $P_i = \{m_B\}$ (the port m of B) and $P_o = \{p_E\}$ (the port p of E), is computed by finding the sets C_s, which is $\{B, C, E, D, A\}$, and C_p, which is $\{E, C, B\}$. The intersection of these two sets is $\{B, C, E\}$, the components in the data-flow.

As we will see in the next section, in practice, this algorithm is not sufficient to detect and correctly report errors in the definition of a virtual component, which

can be caused either by a mistake during the definition or by a modification of the system architecture.

3 Realizing Virtual Components in Component Models

Additional concepts and techniques are required in order to practically use virtual components in real component models. This section gives an overview of the properties of virtual components, and outlines additional requirements in order to realize the concepts of virtual components in two concrete component execution frameworks.

3.1 Detecting Ill-Formed Virtual Components

Virtual components are defined solely via their inputs and outputs. The algorithm to determine which components are part of a flow, and hence a virtual component, was presented in [16]. In practice, only a limited combination of inputs and outputs will lead to a meaningful data-flow. Incorrect combinations can be due to user errors in flow definition, or due to changes in the system architecture. Such combination can lead to tests validating component interactions not representative of the interactions in the complete system, prevent the test component to connect to the virtual component, or reveal directly that the implementation does not conform to the specification. In order to provide a user-friendly integration testing environment, ill-formed virtual components must be detected and reported with enough information, so that it is easy to correct or accept them. In the following, we discuss algorithms to handle most of the issues in flow definition.

Weak flows. A virtual component should always correspond to a complete set of component interactions, i.e. incorporate all components that contribute to the considered flow. However, some components in a flow might receive inputs from components which have not been defined as being part of that particular flow. We refer to these flows as *weak flows*. For example, the flow in Fig. 4, does not incorporate the input to s_C as part of the virtual component. In real systems, the combined behaviour of the components in the flow will also likely depend on these inputs unidentified in the virtual component. This might depend on the particular context of the running system and cannot be determined from the topology of the system alone. An integration test may fail because of a

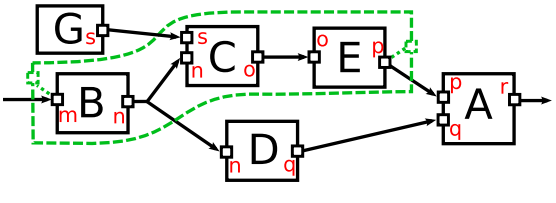


Fig. 4. Example of a *weak flow*

poorly defined virtual component, and not because of a fault. Inversely, the test could pass while in the actual system, with all the inputs, the implementation behaves wrongly. Generally, weak flows are signs of an oversight from the tester. Weak flows must be detected and indicated, so that the integration tester may determine the full test flow. Let us note that this is different for the symmetrical case with outputs, because not taking into account an output in the test cannot change the behaviour of the components.

The following algorithm can be used to verify the completeness of virtual components. P_i is the set of input ports, P_o is the set of output ports, P_w is empty initially, and C is the set of components in the virtual component:

1. For each input of each component in C, add it to the set P_w.
2. For each output of each component, for each of the input ports to which they are connected, remove the input port from P_w.
3. Remove all inputs of P_i from P_w.

If the P_w is not empty, the flow is weak, and the inputs contained in this set are the ones causing the weak flow.

Empty flows. Another problem of a virtual component is that of an *empty flow*, as illustrated in Fig. 5: there is no flow from input n_D to output p_E. Such ill-formedness appears if an input or an output explicitly part of the virtual component is not used in component interaction. In Fig. 5, the error is either the absence of port t_F in the virtual component (an error in the test definition), or the need for component D to also transmit its output to E (an error in the implementation). Such topologies should not be accepted as virtual component, and should be reported to the user as an error. The following algorithm permits to verify such condition:

1. For each input in P_i, add the component owning it to C_m.
2. For each output in P_o, add the component owning it to C_m.
3. Remove from C_m all the components in C.

In case the set C_m is not empty, there is one or more empty flow. Each empty flow starts or ends with one of the component in C_m.

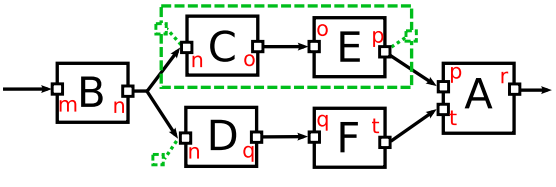


Fig. 5. Example of an *empty flow*

Parallel flows. The third peculiar topology of a virtual component is *parallel flows*, illustrated in Fig. 6, in which the virtual component has been defined to correspond to two independent flows C, E and D, F. This is likely a sign that

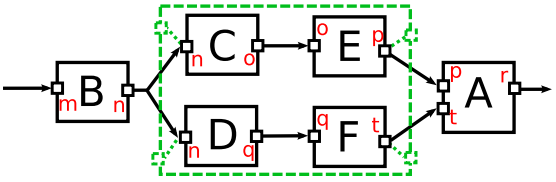


Fig. 6. Example of a *parallel flow*

inputs and outputs which were related in the specification are not related to each other in the implementation. In such a case, the implementation is probably incorrect. At least, this ill-formedness is an indicator that one large-scale virtual component could be redefined as several smaller-scale virtual components, which would be easier to test and to maintain. However, this is not necessarily an issue caused by the tester, it could be intentional, for example, to validate the timing between the two flows. This is why only a warning should be displayed in such a case. Detecting parallel flows is equivalent to the connected component problem in graph theory, which may be addressed through techniques described in [11]:

1. Select i, one of the inputs in P_i, and remove from P_i.
2. Initialize C_c as an empty set.
3. Starting from the component owning i, recursively add all the successors and predecessors to C_c. For each of these components remove all their input ports from P_i.
4. Add C_c to the set of sets SC_c.
5. Repeat until P_i is empty.

If SC_c contains more than one set, then the virtual component contains parallel flows. Each of the flows corresponds to one of the sets in SC_c.

3.2 Extending the Component Model

For a component framework to support virtual components, its API and implementation must be extended. Here, we describe the amendments in general terms, as most component models provide comparable concepts supporting these modifications. Two concrete implementations are described in Section 4.

Typically, an API provides functions to start and stop components, to bind and unbind their interfaces – unless connections are implicit and components are automatically linked when using the same data type – and to add components to and remove components from the framework. The most essential change to be introduced in the component model is the concept of a "virtual" component. This adds to the concepts of composite and primitive types of most component models. In hierarchical frameworks, we propose to associate each virtual component with the composite component that is parent of all components comprising a data-flow. In other words, each composite component has a set of virtual components to test several interactions between its sub-components. This organisation

permits to follow the component and BIT paradigms naturally. That is, components are black boxes, and their development and deployment are separate. This allows for a scalable virtual component approach. Every component will have its own integration testing facility, which is also managed independently. Later in the development and testing process, additional virtual components can be associated with larger composite components to validate the global composites of these building blocks.

A new interface is added to the composite components for adding/removing virtual components. Note that composite components have already an interface to add/remove components but it is better not to use it, in order to avoid mixing testing functionality with nominal functionality. Using a separate interface to manage the virtual components means that only the parts of the framework involved with testing must be updated, and it ensures that they are completely transparent for the normal existing components.

The new component type "virtual" shares many of the typical component interfaces, but also has its own characteristics. The BIT interface used to associate and run test-cases can be realized exactly like in the other component types. The BIT interface used to notify a component of the fact that it is tested is also identical to the normal interface. It notifies all components contained in the flow. Similarly, the interface to request the start and stop of a component passes the requests to the components contained in the flow. This is used to initialise and end the components during a test. The interface found in composite components used to add and remove sub-components is not necessary for the virtual components, as components are automatically enclosed. Nevertheless, the functionality to list the sub-components inside a component is replicated in order to retrieve the information about which components are contained in a flow.

In composite components, the bind and unbind interface allow to associate the external interfaces with the sub-component interfaces. This API can be exploited to specify the inputs and outputs of the virtual component, with the idiosyncrasy that this specifies the actual shape of the virtual component. It should be noted that if multiple modifications to the connections are required and successively applied, the topology of the virtual component might be temporary incorrect (as specified in Subsection 3.1). Therefore, unless the framework supports to group modifications in an atomic way, the construct verification cannot be done directly after a change. The verification should be done either whenever the set of contained components is queried, or just before the test-cases associated with the flow are executed.

4 Implementation

In the context of this research, the concepts of virtual components have been implemented in two different component models. One adaptation was performed for the OpenSplice[1] framework, which is a non-hierarchical publish-subscribe platform in which components are not explicitly connected, but are automatically assembled when they share common "topics". This adaptation is not freely

[1] http://www.opensplice.org

```
<virtual-composite name="flowRawData2FilteredData">
  <interface name="in" role="server" signature="AISin"/>
  <interface name="out" role="client" signature="AISin"/>
  <binding client="this.in" server="ais-listener.ais-in"/>
  <binding client="filter.ais-out0" server="this.out"/>
  <test provider="JUnitProviderFlow" name="RawProcess" definition="RawProcess"/>
</virtual-composite>
```

Listing 1.1. Definition of a virtual component with a test-case in an ADL file

available due to confidentiality restrictions. The other adaptation was performed for our Atlas component framework, which is based on the Fractal component model[2]. This framework supports hierarchical structure, has explicit connections, and permits reflective view on the components. It is freely available from our website[3], including the virtual component extension.

The Atlas extension introduces a new component type to represent virtual components, as well as the interfaces discussed above. Because the framework is fully aware of the virtual components, tests can be executed automatically during initialisation of system. Moreover, if the system is modified, the notification of changes are passed to the virtual component infrastructure, which will automatically reset the test status for the data-flows affected. The instantiation of the test component and its binding and unbinding are also handled automatically by the framework. In case a test-case fails, the failure is displayed on the framework console, and the system cannot be started until this is fixed.

The Architecture Description Language (ADL) used by Atlas for describing the system has also been extended to support these new concepts. Listing 1.1 presents an example of a virtual component expressed in this ADL. This flow is taken from the example system described in Section 5. The `interface` tags define the interface of the component. The `binding` tags define the beginnings and ends of the flow, by identifying the specific components at its edges. Finally, the `test` tag denotes the test component containing the test-cases for this flow. The `JUnitProviderFlow` is a component belonging to the BIT infrastructure of Atlas, which is in charge of handling the execution of test components.

Test-cases for data-flow integration testing are defined in terms of a JUnit class, provided that this class also implements the complementary interfaces of the flow. An excerpt of such a class is displayed in Listing 1.2. This corresponds to the flow mentioned in the previous listing, with one test-case `oneMessage` which validates the correct transmission of a message through the flow. The method `init()` is executed just before the test-cases are executed (and after the component has been created and bound to the flow). The method `AISin()` corresponds to the input interface of the testing component.

The OpenSplice implementation follows a different approach, mainly due to the lack of component hierarchy, and of centralised management functions (components can start and stop completely independently from the rest of the system). A special program was created to handle all the virtual components of a

[2] `http://fractal.ow2.org`
[3] `http://swerl.tudelft.nl/bin/view/Main/Atlas`

```
public class RawProcess  implements AISin, BindingController {
  private AISin myServer;
  private List<AISMessage> AISrcv;

  @Before
  public void init() {
    AISrcv = new ArrayList<AISMessage>();
  }

  @Test
  public void oneMessage() throws NoSuchInterfaceException {
    AISMessage mes = new DynamicAISMessage("32w@HUP0380@O`s@1T1P06", 5925L);
    myServer.AISin(mes);
    assertEquals("Message not received.", AISrcv.get(0), mes);
  }

  public void AISin(AISMessage m) {
    AISrcv.add(m);
  }
  ...
}
```

Listing 1.2. Definition of a test-case for validating the flow

given system. Following the definition of the flows given in separate files, and the information about which components have been modified (to be provided by the user, because automatic notification of changes is not available), the program establishes the flows to be tested. Each flow is associated with an OpenSplice component written as a JUnit class. This contains all the test-cases for testing the integration of all components associated with the flow. The framework automatically connects components with compatible topics, so the test-cases, integrated as yet another component, are directly attached with the right flow. It is important to note that we use existing operations of the component model in order to implement these mechanisms.

In the next section, we use the Atlas implementation to evaluate with a real system to which extent the advantages of the virtual component approach pay off during integration testing.

5 Evaluation

In addition to performing a feasibility study, the goal of this evaluation is the assessment of whether the virtual component approach improves the failure detection rate during integration testing compared to the provider testing approach presented in Section 2. In order to being able to compare the performance of the two approaches in terms of failure detection, mutations are introduced in the system and for each of them, integration tests are executed, according to the principles of the two approaches. The evaluation is performed on a part of a maritime surveillance system used as case study.

Study Subject. Before going into more details on the evaluation, we briefly present the case study system. The surveillance system receives information broadcasts from ships, called *AIS messages* [12], and it processes them in order

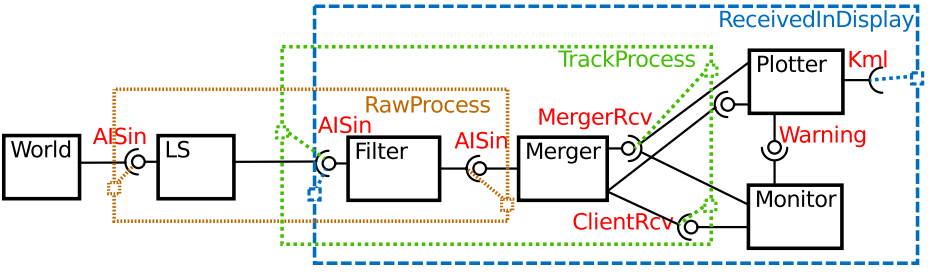


Fig. 7. Architecture of the surveillance system used, with 3 virtual components

to form a situational picture of the coastal waters. The (simplified) architecture of this system is displayed in Fig. 7. The `World` component simulates the ships transmitting data, by replaying AIS messages recorded from reality. The `LS` component receives all AIS data from the antennas physically spread along the coast. The `Filter` component suppresses duplicate messages, because some receivers cover overlapping areas. The `Merger` acts as a temporary database of AIS messages, and client components can consult it to receive tracking information of a ship. The clients must send typical database query requests for retrieving the ship tracks. They are connected following a call-reply architectural style. However, at a high level of abstraction, they are organised according to a data-flow architectural style. The `Monitor` and the `Plotter` are both clients of the `Merger`. The former detects discrepancies in the data, while the latter displays the ship tracks on the screen of the command and control centre (by sending vector drawings to the actual display system). The components are implemented as Atlas components in Java.

Test-cases. Three virtual components were defined, i.e., `RawProcess`, `TrackProcess`, and `ReceivedInDisplay` (Fig. 7). These correspond to three data-flows each of which has a defined expected behaviour. Every flow (i.e., each virtual component) is associated with several test-cases used to validate the defined behaviour. For example, one of the test-cases of `ReceivedInDisplay` sends AIS messages from two ships and verifies that instructions to display both ships are sent. For the provider integration testing, the components are also equipped with test-cases for assessing the correct responses of the components on which they depend. As an example, the LS component has a test-case which transmits some AIS messages and validates that no exceptions happened. The Plotter component comes with test-cases validating the interpretation of the database protocol by the Merger component.

Component Mutation Testing. Mutation testing is a technique in which faulty programs, i.e., the mutants, are generated in order to check the efficiency of a test method to uncover failures. A mutant is a semantic modification in the implementation of a component introducing a fault.

Table 1. Mutation test results

	Source	True positive	False positive	True negative	False negative	Total
Provider testing	Filter	36	26	2	0	64
	Merger	51	62	4	0	117
Virtual component	Filter	36	7	21	0	64
	Merger	51	41	25	0	117

We had these mutants generated through the μJava[4] tool for the Merger and Filter component. Each of the mutations was applied separately, providing a different version of the system, for which both integration testing methods were executed. "Equivalent" mutants, i.e., modifications that cannot lead to a fault because the system performs nominally as if it was the original version, were sorted out manually. Out of the 181 generated mutants, 94 were deemed as non-equivalent and included in the study. When a test does not find any errors, i.e., the mutated system is considered to operate fine, the result is termed "positive". When an error is reported, the result is termed "negative". "False positives" are the mutants which are said to be working fine, although it was manually verified that they behave outside of the specification. "False negative" represent cases for which a correct system is classified as having an error.

All tests pass when applied to the original (non-mutated) system. Table 1 summarizes the integration testing results obtained when using the provider and virtual component testing approaches. None of the tests applied has produced false negatives. This had been expected because all the tests passed on the original system. The provider integration testing approach is only able to trigger a few failures, i.e., 6% of the faulty mutants detected. In contrast, the virtual component integration testing approach is able to detect a much larger population of the faulty mutants, i.e., 49%. All the failures triggered by the provider testing approach are also identified by the virtual component testing approach.

Architecture mutation testing. To evaluate the detection of faults in the architecture, we seeded faults in the case study system by changing or removing connections between components. All the 5 mutated configurations had significant incorrect behaviour. The provider testing method detected 2 of the 5 faults, while the virtual component method detected all the 5 faults. More precisely, 3 of the faults were directly detected by the well-formedness checks, therefore not even requiring the execution of the test-cases.

Discussion. First, these results confirm our initial supposition that the virtual component integration testing approach is successful in detecting failures in the specific context of a data-flow architectural style. Second, the results suggests a much better capacity of detecting problems compared with a more "traditional" or "typical" integration testing approach. It should be noted that the

[4] http://cs.gmu.edu/~offutt/mujava/

tests were written without knowledge of the mutants. The creation of additional tests specifically crafted to detect the mutants, would have probably increased the true negatives.

Unit tests for the two mutated components were able to detect 75% of the bugs introduced, and 100% would probably be achievable with more elaborate test suites. However this would not be a fair comparison. Integration testing can only detect bugs in program sections which are executed, and therefore cannot detect all the mutations. Moreover, the integration testing also targets faults that are due to different interpretations of a same specification, or due to mistakes in the architecture of the system. This cannot be simulated by mutation testing alone, and unit testing cannot detect such issues, as highlighted by the architecture mutations, which none of the unit tests would have revealed.

6 Conclusions and Future Work

We have presented the implementation and usage of *virtual components* to facilitate the integration testing of component systems organised following a data-flow architectural style. First, three algorithms have been introduced to enforce well-formedness of the virtual components. They are key to a user-friendly realization of this new concept in a component middleware platform. A guideline to extend the typical component interfaces for the manipulation of virtual components was presented. Second, we introduced two implementations of virtual component testing for two types of component middleware platforms, demonstrating the applicability of the approach in practice. Finally, the evaluation of this integration testing approach using mutation testing on a system from our industrial partner showed the effectiveness in detecting errors in systems organised following a data-flow schema. We could show that on this system half of the component mutants were detected by this approach, in contrast to 6% detected by the traditional provider integration testing approach. All 5 architecture mutants were also detected instead of the 2 detected using provider testing.

In future work, we will study ways to minimize the number of test-cases executed during regression integration testing. For example, a test might be repeated only if it assesses non-functional properties, or it is repeated depending on a modification performed.

References

1. Abdullah, K., Kimble, J., White, L.: Correcting for unreliable regression integration testing. In: ICSM 1995: Proceedings of the International Conference on Software Maintenance, p. 232. IEEE Computer Society, Washington (1995)
2. Beer, A., Heindl, M.: Issues in testing dependable event-based systems at a systems integration company. In: ARES 2007: Proceedings of the the Second International Conference on Availability, Reliability and Security, pp. 1093–1100. IEEE Computer Society, Washington (2007)

3. Bertolino, A., Inverardi, P., Muccini, H., Rosetti, A.: An approach to integration testing based on architectural descriptions. In: ICECCS 1997: Proceedings of the Third IEEE International Conference on Engineering of Complex Computer Systems, p. 77. IEEE Computer Society, Washington (1997)
4. Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., Suliman, D.: Reducing verification effort in component-based software engineering through built-in testing. Information Systems Frontiers 9(2-3), 151–162 (2007)
5. E. U. Commission, Maritime Affairs: An integrated maritime policy for the european union (October 2007)
6. Gao, J.Z., Tsao, H.S.J., Wu, Y.: Testing and Quality Assurance for Component-Based Software. Artech House, Norwood (2003)
7. González, A., Piel, É., Gross, H.G.: Architecture support for runtime integration and verification of component-based systems of systems. In: 1st International Workshop on Automated Engineering of Autonomous and run-time evolving Systems (ARAMIS 2008), pp. 41–48. IEEE Computer Society, L'Aquila (Septmeber 2008)
8. Green Hat Software: Lessons from testing service oriented architectures white paper (2008)
9. Gross, H.G.: Component-Based Software Testing with UML. Springer, Heidelberg (2005)
10. Gross, H.G., Mayer, N.: Built-in contract testing in component integration testing. Electronic Notes in Theoretical Computer Science 82(6), 22–32 (2004)
11. Hopcroft, J., Tarjan, R.: Algorithm 447: efficient algorithms for graph manipulation. Commun. ACM 16(6), 372–378 (1973)
12. International Telecommunication Union: Recommendation ITU-R M.1371-1 (2001)
13. Jorgensen, P.: Software Testing: A Craftman's Approach. CRC Press, Inc., Boca Raton (2001)
14. Object Management Group: UML 2 Infrastructure (Final Adopted Specification) (Septmeber 2003), http://www.omg.org/cgi-bin/doc?ptc/2003-09-15
15. Paul, R.: End-to-end integration testing. In: APAQS 2001: Proceedings of the Second Asia-Pacific Conference on Quality Software, p. 211. IEEE Computer Society, Washington (2001)
16. Piel, É., Gonzalez-Sanchez, A.: Data-flow integration testing adapted to runtime evolution in component-based systems. In: Workshop Software Integration and Evolution @ Runtime. ACM, Amsterdam (August 2009)
17. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an emerging discipline. Prentice-Hall, Englewood Cliffs (1996)
18. Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.: The MORABIT approach to runtime component testing. In: 30th Annual International Computer Software and Applications Conference. pp. 171–176 (Septmeber 2006)
19. Thales Group: Maritime safety and security (2007), http://www.thalesgroup.com/Portfolio/Security/D3S_Maritime_Safety_and_security/
20. Vincent, J., King, G., Lay, P., Kinghorn, J.: Principles of built-in-test for run-time-testability in component-based software systems. Software Quality Journal 10(2), 115–133 (2002)
21. Yuan, X., Memon, A.M.: Generating event sequence-based test cases using GUI run-time state feedback. IEEE Transactions on Software Engineering 36(1) (2010)
22. Zhu, H., He, X.: A Methodology of Component Integration Testing. In: Testing Commercial-off-the-Shelf Components and Systems, pp. 239–269. IEEE Computer Society, Los Alamitos (2005)

Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing

Andrea Arcuri[1], Muhammad Zohaib Iqbal[1,2], and Lionel Briand[1,2]

[1] Simula Research Laboratory, P.O. Box 134, Lysaker, Norway
[2] Department of Informatics, University of Oslo
{arcuri,zohaib,briand}@simula.no

Abstract. Testing real-time embedded systems (RTES) is in many ways challenging. Thousands of test cases can be potentially executed on an industrial RTES. Given the magnitude of testing at the system level, only a fully automated approach can really scale up to test industrial RTES. In this paper we take a black-box approach and model the RTES environment using the UML/-MARTE international standard. Our main motivation is to provide a more practical approach to the model-based testing of RTES by allowing system testers, who are often not familiar with the system design but know the application domain well-enough, to model the environment to enable test automation. Environment models can support the automation of three tasks: the code generation of an environment simulator, the selection of test cases, and the evaluation of their expected results (oracles). In this paper, we focus on the second task (test case selection) and investigate three test automation strategies using inputs from UML/MARTE environment models: Random Testing (baseline), Adaptive Random Testing, and Search-Based Testing (using Genetic Algorithms). Based on one industrial case study and three artificial systems, we show how, in general, no technique is better than the others. Which test selection technique to use is determined by the failure rate (testing stage) and the execution time of test cases. Finally, we propose a practical process to combine the use of all three test strategies.

Keywords: Search based software engineering, branch distance, model based testing, environment, context, UML, MARTE, OCL.

1 Introduction

Real-time embedded systems (RTES) represent a major proportion of the software being developed [1]. The verification of their correctness is of paramount importance, particularly when these RTES are used for business or safety critical applications (e.g., controllers of nuclear reactors and flying systems). Testing RTES is particularly challenging since they operate in a physical environment composed of possibly large numbers of sensors and actuators. The interactions with the environment can be bound by time constraints. For example, if the RTES of a gate is informed by a sensor that a train is approaching, then the RTES should command the gate to close down before the train reaches the gate. Missing such time deadlines can have disastrous consequences in the environment in which the RTES works. In general, the timing of interactions with the

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 95–110, 2010.

real-world environment in which the RTES operates can have a significant effect on the resulting behavior of test cases.

In this paper our objective is to enable the black-box, automated testing of RTES based on environment models. More precisely, our goal is to make such environment modeling as easy as possible, and allow the testers to automate testing without any knowledge about the design of the RTES. This is typically a practical requirement for independent system test teams in industrial settings. In addition, the test must be automated in such a way to be adaptable and scalable to the specific complexity of a RTES and available testing resources. By adaptable, we mean that a test strategy should enable the test manager to adjust the amount of testing to available resources, while retaining as high a fault revealing power as possible.

The system testing of a RTES requires interactions with the actual environment or, when necessary and possible, a simulator. Unfortunately, testing the RTES in the real environment usually entails a very high cost and in some cases the consequences of failures would not be acceptable, for example when leading to serious equipment damage or safety concerns. In our context, a test case is a sequence of stimuli, generated by the environment or its simulator, that is sent to the RTES. If a user interacts with the RTES, then the user would be considered as part of the environment as well. There is usually a great number and variety of stimuli with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. A test case can also contain changes of state in the environment that can affect the RTES behavior. For example, with a certain probability, some hardware components might break, and that has effect on the expected and actual behavior of the RTES. A test case can contain information regarding when and in which order to trigger such changes.

Testing all possible sequences of environment stimuli/state changes is not feasible. In practice, a single test case of an industrial RTES could last several seconds/minutes, executing thousands of lines of code, generating hundreds of threads/processes running concurrently, communicating through TCP sockets and/or OS signals, and accessing the file system for I/O operations. Hence, systematic testing strategies that have high fault revealing power must be devised.

The complexity of modern RTES makes the use of systematic testing techniques, whether based on the coverage of code or models, difficult to apply without generating far too many test cases. Alternatively, manually selecting and writing appropriate test cases based on human expertise for such complex systems would be far too challenging and time consuming. If any part of the specification of the RTES changes during its development, a very common occurrence in practice, then many test cases might become obsolete and their expected output would potentially need to be recalculated manually. The use of an automated oracle is hence another essential requirement when dealing with complex industrial RTES.

In this paper we present a Model-Based Testing (MBT) [2] methodology to carry out system testing of RTES in a fully automated, adaptable, and scalable way. We tailor the principles of Adaptive Random Testing (ART) [3] and Search-Based Testing (SBT) [4] to our specific problem and context. For our empirical evaluation, we use Random Testing (RT) [5] as baseline. One main advantage of ART and SBT is that it can be tailored to whatever time and resources are available for testing: when resources are

expended and time is up, we can simply stop their application without any side effect. A coverage-based strategy could not be, for example, interrupted at any time. Furthermore, ART and SBT attempt, through different heuristics, to maximize the chances to trigger a failure within time constraints. We will also see how their combined use can be helpful to gain the most out of testing resources in practice. The RTES under test (SUT) is treated as a black box: no internal detail or model of its behavior is required, as per our objectives. The first step is to model the environment using the UML standard and its MARTE profile, the latter being necessary to capture real-time properties. The use of international standards rather than academic notations is dictated by the fact that our solutions are meant to be applied by our industry partners. Environment models support test automation in three different ways:

- The environment models describe some of the structural and behavioral properties of the environment. Given an appropriate level of detail, they enable the automatic generation of an environment simulator to satisfy the specific needs of software testing.
- The models can be used to generate automated oracles. These could for example be invariants and error states that should never be reached by the environment during the execution of a test case (e.g., an open gate while a train is passing). In general, error states can model unsafe, undesirable, or illegal states in the environment. We used error states as oracles in our case studies.
- Test cases can be automatically selected based on the models, using various heuristics to maximize chances of fault detection. In our case studies we use ART and SBT.

In this paper we focus on the third item above and assess RT, ART, and SBT on the production code of a real industrial RTES. Due to space constraints, and because our focus in this paper is test automation, we do not explain in detail how to use UML/MARTE to model the environment of a RTES and how simulator code can be automatically generated (which we investigated in [6]). To the best of our knowledge, no MBT automation results for ART and SBT on an actual RTES have ever been reported in the research literature. Since no freely available RTES was available, we also constructed three different artificial RTES in order to extend our investigation and better understand the influence of various factors on test cost-effectiveness such as the failure detection rate. The use of publicly available artificial RTES will also facilitate future empirical comparisons with our work since, due to confidentiality constraints, our industrial case study cannot be made public.

The paper is organized as follows. Section 2 provides an overview of related work. How the context is modeled and simulated is shortly discussed in Section 3. Section 4 describes the different strategies we used to generate test cases. Their empirical validation is described in Section 5 and threats to validity are discussed in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

A large body of literature has been dedicated to test RTES. For reason of space, here we can only give a very brief and incomplete overview.

Most of the approaches to test RTES are based on violating their timing constraints [7] or checking their conformance to a specification [8]. The specification is generally a formal model of the system and this model is then used to generate test cases. A number of approaches have been proposed over the years to address the above problem. The most widely discussed approaches model the system using Timed Automata [9]. A number of Timed Automata extensions, such as Timed I/O Automata [10], have also been used for conformance testing. For the same purpose, UML statechart [11], Extended Finite State Machines [12] and Attributed Event Grammar [13] have also been used.

There are several works using SBT techniques for testing different aspects of RTES [14], as for example identify deadline misses [15] and testing functional properties [16].

The work presented here is significantly different from most the above approaches as we adopt, for practical reasons presented above, a black-box approach to system testing that relies on modeling the RTES environment rather than its internal design properties. As noted above, this is of practical importance as independent system test teams usually do not have easy access to precise design information. Most existing work does not focus on system testing, hence their emphasis on modeling the RTES internal behavior and structure. Another difference of practical importance, though this is not detailed in this paper, is that we use UML and its standard extensions for modeling the environment. Last but not least, as opposed to published case studies (e.g., [13,12]), we assess our test strategies on the actual production code of an industrial RTES.

3 Environment Modeling and Simulation

For RTES system testing, software engineers would typically be responsible for developing the environment models. Therefore, the modeling language should be familiar to them and therefore based on software engineering standards. In other words, it is important to use a modeling language for environment modeling that is widely accepted and used by software engineers. Furthermore, standard modeling languages are widely supported in terms of tools and training. The Unified Modeling Language (UML) and its extensions are therefore a natural choice to consider in our context.

Several modeling and simulation languages are available and can be used for modeling and simulating the context (e.g., DEVS [17]). But in our case using these simulation languages raises a number of issues, including the fact that software engineers in the development team are usually not familiar with the notations and concepts of such languages.

Higher level programming languages (such as Java or C) can also be used as simulation languages. The major problem with the use of such languages is the low level of abstraction at which they "model" the environment. The software engineers will have to deal with all the programming language constructs (such as threads) while at the same time trying to focus on the details of the environment itself.

RTES testing through an environment simulator faces the question of how time is handled. Indeed, many properties of the RTES depend on whether some time constraints are fulfilled or not. Ideally, we would like to be able to simulate the passing of time in a deterministic way, but it is not always possible for large and complex RTES.

The opposite approach to time simulation would be to run the RTES with its simulated environment using the real clock of the CPU used to run the empirical analysis. On one hand, it has the benefit that we do not have any particular constraint on the type of RTES that can be analyzed. On the other hand, it adds noise and variance in the scheduled time events. If time constraints of the RTES are very tight (e.g., in the order of few milliseconds), then this approach is not a viable option.

In our work, we have used UML/MARTE as a simulation language. Models are developed in UML as classes and their state-machines. These models are then transformed into Java using model to text transformations. The activities and actions are written in Java and are converted into Java method calls. This was appropriate for the RTES considered in this paper. For other types of RTES, different programming languages could be necessary. Notice that our methodology is general. We chose Java only for practical reasons. In particular, in our empirical analyses we did not face the problem of the garbage collector interfering with time properties. The garbage collector was never called during the execution of a test case.

4 Automated Testing

4.1 Test Case Representation

In our context, a test case execution is akin to executing the environment simulator. Each state machine represents a component of the environment. There can be more instances of a state machine with different settings to represent different sensors/actuators of the same type. For example, in a gate controller RTES, we can have a state machine representing the trains. For each simulated train we will have an independent running instance of that state machine. The domain model is used to identify how many instances can or should run in parallel for each state machine. Based on the domain model, there could be different possible configurations of the environment, but in this paper we focus only on one fixed configuration.

In the behavioral models of the environment (i.e., the state machines) there can be non-deterministic parts. For example, a timeout transition could be triggered within a minimum and a maximum time value but the exact value cannot be determined. This is very typical when real-world components are modeled, in which for example there is always a natural variance when time-related properties are represented. Another example is when we assign probabilities p in the models to represent failure scenarios, as for example the breakdown of sensors/actuators. In our context, input data of a test case are the choice of the actual values to use in these non-deterministic events.

In our modeling methodology, we have non-deterministic choices only in the transitions between states. They can be in the trigger, the guard and the action of the transition. A transition might be taken several times, and this number might be unknown before executing the test case. Therefore, for each instance of the environment state machines, for each non-deterministic choice, we allocate in the test case a vector of possible values. The length of this vector is l. Each time such non-deterministic choice needs to be made, a value from the corresponding vector is selected. Because the vector has finite length l, it is used as a ring: The values are taken in order, and after l request for values, it starts again from the beginning of the vector. Figure 1 shows an example.

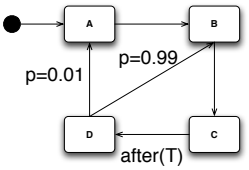


Fig. 1. Example of a reduced UML/MARTE state machine

Let the transition $C \rightarrow D$ have a non-deterministic choice in $[0,1]$, for example the timeout $T \in [0,1]$. Given for example $l = 2$, we would have a data vector containing for example $\{0.4,0.32\}$. The first time the transition $C \rightarrow D$ is taken, the value 0.4 is used fort the non-deterministic choice. The second time, the value 0.32 is used. The third time, the value 0.4 is used again, and so on.

Given n state machine instances, and m non-deterministic choices in each of them (for simplicity, because in general instances of different machines will have a different number of non-deterministic choices), we would have that each test case contains $L = n*m*l$ values, which can be represented as a vector. The choice of l is arbitrary but has significant consequences. On one hand, a small number of possible values could make it impossible to represent sequences of event patterns that lead to failures in the RTES. On the other hand, a high number of possible values will lead to long vectors and might harm the effectiveness of test selections techniques such as ART and SBT (discussed in more details in the next sections).

In our case studies, the values to include in the test case data are chosen before the execution of the test cases. This means that the domain of these values should be static and not depending on the dynamic execution of the test cases. For example, if a variable is constrained within a minimum and maximum limit, then these boundaries should be known before test execution. This is the case for the industrial RTES analyzed in this paper and for other RTES we have worked with. When this is not the case, we would need to enable the choice of non-deterministic options at runtime.

4.2 Testing Strategies

As described in the previous section, a test case can be seen as a vector V. Elements in this vector can be of different types, but their domain of valid values should be known. Given $D(i)$ the domain of the ith variable in V, we obtain that the number of possible valid test cases is $\prod |D(i)|$, which is an extremely large number. An exhaustive execution of all possible test cases is infeasible.

In this paper we consider the testing problem of sampling test cases to detect failures of the RTES with automated oracles derived from the environment models. For all test strategies, the oracle checks whether a transition to an error state specified in the model occurs during test execution. We choose and execute test cases one at a time. We stop sampling test cases as soon as a failure has been found. A test strategy that requires the sampling of fewer test cases to detect failures should obviously be preferred.

The simplest, automated technique to choose test cases is Random Testing (RT). For each variable in V, we simply sample a value from its domain with uniform probability.

Although RT can be considered to be a naive technique, it has been shown to be effective in many testing situations [18, 19].

Another technique that we investigate is Adaptive RT (ART) [3], which has been proposed as an extension of RT. The underlying idea of ART is that diversity among test cases should be rewarded, because failing test cases tend to be clustered in contiguous regions of the input domain. ART can be automated if one can define a meaningful similarity function for test cases. To the best of our knowledge, we are aware of no previous application of ART to test RTES. In this paper we use the basic ART algorithm described in [3].

Because in our case studies all the variables in V are numerical, for the distance between two test case data vectors $V1$ and $V2$ we use the following $dis(V1,V2) = \sum abs(V1[i]-V2[i])/|D(i)|$. We sum the absolute difference of each variable weighted by the cardinality of the domain of that variable. Often, these variables represent the time in timeout transitions. Therefore, ART rewards diversity in the triggering time of events.

In this paper we also investigate the use of search algorithms to tackle the testing of RTES. In particular we consider the use of Genetic Algorithms (GAs), which are the most used search algorithms in the literature on search-based testing (SBT) [14]. To use search algorithms to tackle a specific problem, a fitness function needs to be defined tailored to solve that problem. Search algorithms exploit the fitness function to guide the search toward promising areas of the search space. The fitness function is used to heuristically evaluate how "good" a test case is. In our case, the fitness function is used to estimate how close a test case is from triggering a failure in the RTES, that is when at least one component of the environment enters an error state. This is once again determined by analyzing the environment models.

To tackle the testing problem described in this paper, we developed a novel fitness function f that can be seen as an extension of the fitness functions that are commonly used for structural testing [4] and MBT [20]. In our case, the goal is to minimize the fitness function f. If at least one error state is reached when a test case with test data V is executed, then $f(V) = 0$. For each error state E in each state machine instance we employ the so called approach level A and branch distance B. The approach level calculates the minimum number of transitions in the state machine to reach an error state from the closest executed state. The branch distance is used to heuristically score the evaluation of the Object Constraint Language (OCL) constraints in the closest executed state from which the approach level is calculated. The branch distance is used to guide the search to find test data that satisfy those OCL constraints. A transition could be triggered several times but never executed because the guard fails. For the branch distance, we calculate it every time but then we only consider the minimum value it obtains. Because the branch distance is less important than the approach level, it is normalized in the range [0,1]. We use the following normalizing function $nor(x) = x/(x+1)$, which has been shown to be better than other normalizing functions used in the literature [21]. Notice that, in the case of MBT, it is not always possible to calculate the branch distance when the related transition has never been triggered. In these cases, we assign to the branch distance B its highest possible value.

The extension of the fitness function we make in this paper exploits the time properties of the RTES. Some of the transitions are triggered when a time-threshold is violated. For example, an error state could be reached if a sensor/actuator does not receive a message from RTES within a time limit. If such transitions exist on the path toward the execution of the error states, then we need a way to reward test data that get the execution closer to violate those time constraints. If a transition is taken after a threshold z, then we calculate the maximum consecutive time t the state machine stays in the state from which that transition can be triggered (this would be the same state from which the approach level is calculated from). Then, to guide the search we can use the following heuristic $T = z - t$, where $t \leq z$.

Finally, the fitness function f for a test data vector V is defined as:

$$f(V) = min_E(A_E(V) + nor(T_E(V)) + nor(B_E(V))).$$

Notice that, to collect information such as the approach level, the source code of the simulator needs to be instrumented. This is automatically done when this code is generated from the environment models.

Once the fitness function is defined, we can use it to guide the GA to select test cases. But GAs have many parameters that need to be set. In this paper we use a Steady State GA [4]. We employ rank selection with bias 1.5 to choose the parents. A single point crossover is employed with probability $P_{xover} = 0.75$. This operator chooses a random point inside the data vectors V of the parents sx and sy. The elements in the data vector after that splitting point are swapped between the two parent solutions. Each of the L elements in a data vector is mutated with probability $1/L$. A mutation consists of replacing a value with another one at random from the same domain. The population size is chosen to be 10. The optimal configuration of search algorithms is in general problem dependent [22]. Due to the large computational cost of running our empirical analysis, we have not tuned the GA. We simply use reasonable parameter values given in the literature of GAs.

5 Empirical Study

5.1 Case Study

To validate the novel approach presented in this paper, we have applied it to test an industrial RTES. The analyzed system is a very large and complex controller that interacts with several sensors/actuators. The company that provided the system is a market leader in its field. For confidentiality reasons we cannot provide full details of the system. Information of the environment models of this RTES is provided in Table 1. Notice that for this case study there are several state machines, and for each of them there can be one or more instances running in parallel at the same time. For each test case, 23 instances of state machines run in parallel, each of them can start several threads. The total number of non-deterministic choices (NDCs) is 82. The UML/MARTE context models were developed in IBM Rational Software Architect. Constraints, such as guards, were expressed in OCL.

Table 1. Summary of the state machines of the environment of the industrial RTES. NDC stands for "Non-Deterministic Choice".

State Machine	States	Transitions	Error States	Instances	NDCs for Instance
S1	19	29	1	10	6
S2	4	7	0	11	2
S3	3	8	1	1	0
S4	5	5	0	1	0

Table 2. Properties of the three artificial problems. LoC stands for "Lines of Code", whereas NDC stands for "Non-Deterministic Choice".

Artificial Problem	LoC of RTES	LoC of Environment	State Machines	States	Transitions	Instances	Total NDCs
AP1	227	259	1	5	7	10	20
AP2	409	271	1	5	7	2	4
AP3	337	318	2	9	13	5	18

To facilitate future comparisons with the techniques described in this paper, it would be necessary to also employ a set of benchmark systems that are freely available to researchers. Unfortunately, we have not found any RTES satisfying this criterion. Therefore, in addition to our industrial case study, we have designed three artificial RTES, called AP1, AP2 and AP3. Two of them are inspired by the industrial RTES used in this paper, whereas the third is inspired by the control gate system described in [12]. The RTES are written in Java to facilitate their use on different machines and operating systems. For the same reason, the communications between the RTES and their environments are carried out through TCP. The use of TCP was also essential to simplify the connection of the RTES with its environment. For example, if the simulator of the environment is generated from the models using a different target language (e.g., C/C++), then it will not be too difficult to connect to the artificial RTES written in Java. These RTES are all multithreaded. Table 2 summarizes the properties of these artificial RTES. In each of them, there is only one error state. We introduced by hand a single non-trivial fault in each of these RTES.

5.2 Experiments

We have carried out two different sets of experiments. One for the artificial problems, and one for the industrial RTES. In all these experiments, the value l for the non-deterministic choices is set to $l = 3$. This means that the number of input variables in each test case is 60 for AP1, 12 for AP2, 54 for AP3 and finally 246 for the industrial RTES.

In the first step of the experiments, we ran RT, ART and GA on each of the three artificial problems. Because the execution of a single test case takes 10 seconds, we stop each algorithm after 1000 sampled test case or as soon as one of the error state is reached. Notice that the value 10 seconds is fixed, and it does not depend on the used

Table 3. Success rate (out of 100 runs) for the three artificial problems.

Algorithm	AP1	AP2	AP3
RT	6	35	49
ART	0	40	74
GA	90	21	31

Table 4. Number of sampled test cases to detect the first failure in the considered industrial RTES. "SD" stands for Standard Deviation.

Algorithm	Min	Median	Mean	Max	SD
RT	1	73.0	131.9	912	164.9
ART	1	75.5	104.6	525	99.7
GA	1	99.0	160.0	767	155.2

execution platform. Using faster hardware would not change the amount of time required to run these experiments. The only requirement is that the hardware used for the experiments is fast enough to sustain the CPU load without introducing delays higher than a few milliseconds. Because in these simulations most of the time the CPU is in idle state, the computers used in the experiments were appropriate.

For each test strategy and each case study, we ran the algorithms 100 times with different random seeds. Because these algorithms are randomized, a large number of experiments is required to obtain statistically significant results. The total number of sampled test cases is hence at most $3 * 3 * 1000 * 100 = 900{,}000$, which can take up to 104 days on a single computer. To cope with this problem, we used a cluster to run these experiments.

Given an upper bound of 1000 test cases, it is not always the case that any of the test strategies is able to trigger a failure in the RTES. In Table 3 we report how many times each algorithm was able to do so out of the 100 experiments. Because the process of detecting failures in 100 experiments can be considered to be a binomial process with unknown probability [23], we use the Fisher Exact test to compare the success rate of RT with the ones of ART and GA. The significance level of the tests is set to 0.05. Results show that the only case in which there is no significant difference in the success rate is for problem AP2 when RT is compared to ART.

The second set of experiments has been carried out on an industrial RTES. In system testing of RTES, the simulation of the environment can in general be run for any arbitrary amount of time. But there should be enough time to render possible the execution of all the functionalities of the RTES. For example, in the RTES for a train/gate controller, we should run the simulation at least long enough to make it possible for a train to arrive and then leave the gate. Choosing for how long to run a simulation (i.e., a test case) is conceptually the same as the choice of test sequence length in unit testing [24] (i.e., many short test cases or only few ones that are long?). But in contrast to unit testing in which often the execution time of a test case is in the order of milliseconds, in the system testing of RTES we have to deal with much longer execution time. In this paper, we run each test case for 20 seconds. This choice has been made based on the properties of the RTES and discussions with its software testers.

We evaluated the use of RT, ART and GA to find failures in this RTES. We could not run this empirical analysis on a cluster due to technical reasons. We used a single dedicated computer, and it took nearly ten days to run these experiments. The failure rate of the SUT in these experiments was quite high, so we did not use any upper bound for the number of sampled test cases. The results of experiments are shown in Table 4.

To analyze the results in a sound manner we carried out a set of statistical tests on the data presented in Table 4. We used parametric *t*-tests to see whether there is

Table 5. Results of the statistical tests for the data in Table 4

Comparison	*t*-tests p-value	Cohen D	U-test p-value	Vargha-Delaney A
RT vs ART	0.1588	0.2012	0.9708	0.5015
RT vs GA	0.2150	-0.1768	0.0334	0.4129
ART vs GA	0.0030	-0.4272	0.0193	0.4042

any statistical difference between the mean values of sampled test cases among the three analyzed algorithms. The scientific or practical significance of these differences is evaluated using the Cohen D coefficient. We also carried out non-parametric Mann-Whitney U tests to see whether any of the results of these algorithms is stochastically greater than the others. The scientific significance of this test is measured with the Vargha-Delaney A statistic. For both *t*-tests and Mann-Whitney U tests the significant level is set to 0.05. For the Cohed D coefficient (value d), we classify the effect size as follows [25]: small for $abs(d) = 0.2$, medium for $abs(d) = 0.5$, and finally large for value $abs(d) = 0.8$. In the case of Vargha-Delaney A statistic (value a), we use the following classification [26]: small for $abs(a - 0.5) = .06$, medium for $abs(a - 0.5) = 0.14$ and large for $abs(a - 0.5) = 0.21$. Table 5 summarizes the results of these statistical tests.

5.3 Discussion

In the results of the experiments on the artificial problems shown in Table 3, we can see that no testing technique generally dominates the others. GA is statistically better on the first problem, but it is the worst on the other two problems. Regarding RT and ART, they are equivalent on the second problem, but RT is best on the first, whereas ART is best on the third problem.

The results in Table 3 for GA can be precisely explained. Covering all the non-error states and transitions in the environment models of these problems is very easy, practically all test strategies achieve this. The only difficult part is the transition to the error state. For the first problem AP1, that transition is a time transition with no guard. After a time threshold, that transition is triggered. The novel fitness function proposed in this paper can take advantage of this information, rewarding test cases that get closer to violate that time constraint. In fact, for each test case we can automatically calculate the time that it spends in the state that could lead to the error state. This automated fitness function produces an easy fitness landscape that can be efficiently searched by GA. This explains the fact that GA gets to the error state 90% of the time, whereas RT reaches it only in 6% of the time. However, why do we obtain so much worse results in the other two problems AP2 and AP3? The reason is that the fitness function in these cases is practically a needle-in-the-haystack function. In the transition to the error state, there is a guard that is checking whether one Boolean variable is equal to true. The value of this variable depends on the interactions with the SUT, particularly whether a specific message has been received or not. This type of guard in search-based testing is a known, very difficult problem denoted as the flag problem [27]. In this case, the fitness function provides no gradient, and this makes the search difficult. Unfortunately, testability transformations [27] cannot be used in this case, because in our context the

SUT is a black box. Even if we had access to the SUT, it would still be problematic, because we are aware of no work dealing with the flag problem for the system testing of concurrent programs. Though the above issue is a limitation, in practice, we can automatically determine before running GA whether it will work.

Though we can explain why GA does not work well on AP2 and AP3, why does it behave even worse than RT? The reason is exactly the same for which ART is better than RT: the diversity of the test cases. If there is no gradient in the fitness function, all the sampled test cases would have same fitness value (i.e., the fitness landscape would have a large plateau). So any new sampled test case would be accepted and added to the next generation in GA. The crossover operator does not produce any new value in the data vector V, it simply swaps values between two parent test cases. The mutator operator does only small changes to a data vector, because on average only one variable is mutated. During the search, the offspring have genetic material (i.e., the data vectors) that is similar to the one of the parents. Therefore, the diversity of test cases during GA evolution is much lower than the one of RT. If the hypothesis of contiguous regions of faulty test cases is true for a RTES, then, when there is no gradient in the fitness function, we would a-priori expect this following relationship regarding the performance of testing strategies: $GA \leq RT \leq ART$. For problems AP2 and AP3, this is verified in the results of Table 3.

In the experiments on the industrial RTES, we can see that GA is statistically worse than the other approaches, although the difference is only small/medium in size from a scientific point of view. The results on the industrial RTES shown in Table 4 are important to stress out that the choice of a testing strategy is also heavily dependent on when the SUT is tested. The version of the industrial RTES used in this paper was not a finished product. It was in an early phase of development. The types of failure scenarios introduced with our models were not something that was fully tested before. This explains the high failure rate shown in Table 4. Notice that the failure rate θ can be simply estimated from the mean value of RT, i.e. $\theta = 1/mean(RT)$. The reason is that RT follows a geometric distribution with parameter θ, therefore $mean(RT) = 1/\theta$. In our case, we have $\theta = 1/131.9 = 0.007$, which can be considered to be a high failure rate.

5.4 Practical Guidelines

For high failure rates, it makes sense to use a simple RT instead of more sophisticated techniques, since the expected number of sampled test cases would be low on average. In practice, we would expect high failure rates at the beginning of the testing phase. The failure rate would hence be expected to decrease throughout the development process as faults get fixed. Therefore, we would expect to get good results for RT at the beginning, but then more sophisticated techniques could be required at later stages.

Our results lead us to suggest the following heuristics to apply RT, ART, and SBT in practice: In the early stages of development and testing, when failure rates are still high, one should use RT as it will be very efficient and quick to detect the first failure, without requiring any overhead like ART or SBT. One exception to this rule is when the time of executing a test case is high (e.g., in the order of several seconds or minutes), where we then suggest to use ART as one should enforce test execution diversity to

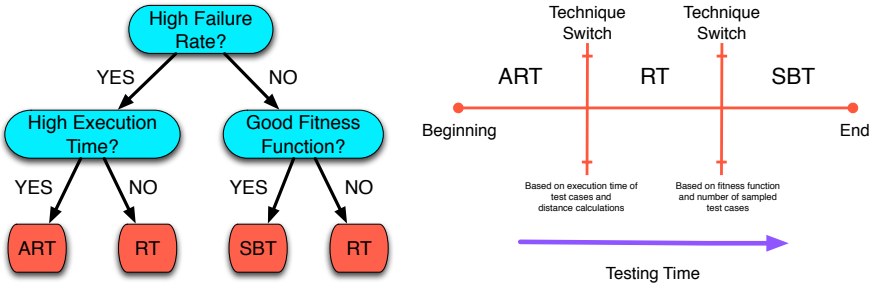


Fig. 2. Decision tree and and application timeline of the three analyzed testing strategies

prevent the execution of too many test cases. Once the failure rate decreases due to the fixing of *easy-to-detect* faults, then use SBT, but only if a proper fitness function can be derived automatically from the models, that is a fitness function that is likely to provide effective guidance for the search of failing test cases. Otherwise, use RT. ART should not be used when the failure rate is low as the overhead of distance calculations would get too high, due to the large number of test cases executed.

Figure 2 summarizes the above heuristic in a decision tree and it shows when to apply each testing technique. We provide practical advice regarding when to switch from ART to RT below. But for the switch from RT to SBT, we need more empirical/theoretical analyses to provide practical guidelines.

In the literature, it has been shown that ART can be twice as fast as RT [3]. Let us consider t_{tc} the execution time of a test case, t_{dis} the execution time of a distance calculation with d the total number of distances computed, θ the failure rate, $E[RT]$ and $E[ART]$ the expected number of test cases sampled by RT and ART. We know that $E[RT] = 1/\theta$ and that, under optimal conditions, $E[ART] = E[RT]/2$. We can develop a heuristic that is based on the following equation: $E[RT]\cdot t_{tc} = E[ART]\cdot t_{tc} + d\cdot t_{dis}$, which is a *loose approximation* to determine the failure rate θ^* above which ART is going to yield better results than RT. From that equation, it follows $\theta^* \approx \frac{t_{dis}}{4\cdot t_{tc}}$. This optimal threshold for ART for the failure rate can be estimated before test execution. Finally, we can suggest to run ART for $1/2\theta^*$ iterations, but only as long as the number of sampled test cases is not high enough to make the decision to switch to SBT. The above recommendations are heuristics and will need to be evaluated and refined as we gather more empirical data.

6 Threats to Validity

Due to the complexity of the industrial RTES used in the empirical study of this paper, we could not run the RTES and its simulated environment in such a way to obtain a precise and deterministic handling of clock time. We used the CPU clock instead. This could be unreliable if time constraints in the RTES are very tight, as for example in the order of milliseconds, because these constraints could be violated due to unpredictable changes of load balance in the CPU because of unrelated processes. Although the time constraints in this paper were in the order of seconds, the problem could still remain. To

evaluate whether our results are reliable, we hence selected a set of experiments, and we re-ran them again with exactly the same random seeds. We obtained equivalent results. For example, if RT for a particular seed obtained a failing test case after sampling 43 test cases, then, when we ran it again with the same seed, it was still requiring exactly 43 test cases. However, the experiments were not exactly the same. For example, for debugging purposes we used time stamps on log files. In these time stamps, small variances of a few milliseconds were present, but this did not have any effect on the testing results. Notice that our novel methodology can obviously be applied also when time clocks are simulated.

7 Conclusion

In this paper we proposed a black-box system testing methodology, based on environment modeling and various heuristics for test case generation. The focus on black-box testing is due to the fact that system test teams are often independent from the development team and do not have (easy) access to system design expertise. Our objective is to achieve full system test automation that scales up to large industrial RTES and can be easily adjusted to resource constraints. The environment models are used for code generation of the environment simulator, selecting test cases, and the generation of corresponding oracles. The only incurred cost by human testers is the development of the environment models. This paper, due to space constraints, has focused on the testing heuristics and an empirical study to determine the conditions under which they are effective, plus guidelines to combine them in practice.

In contrast to most of the work in the literature, the modeling and the experiments were carried out on an industrial RTES in order to achieve maximum realism in our results. However, in order to more precisely understand under which conditions each test heuristic is appropriate and how to combine them, we complemented this industrial study with artificial case studies, that will be made publicly available to foster future empirical analyses and comparisons.

We experimented with different testing heuristics, which have the common property to be easily adjustable to available time and resources: Random Testing (RT), Adaptive Random Testing (ART) and Search-Based Testing using Genetic Algorithms (GAs). All these techniques can be adjusted to project constraints as they can be run as long as time and access to CPU are available. Though RT was originally used as comparison baseline, it turned out to be the best alternative under certain conditions.

On the artificial problems, in one case GA is the best search algorithm, and the difference is very large. But on the other two cases, GA has the worst results, which are due to poor fitness functions. In one case RT and ART are equivalent, but in the other two, RT is better in one case and worse in the other.

However, on the industrial RTES, results are quite different from the artificial case studies: there is no statistical difference between RT and ART, whereas GA is slightly worse than the others (the effect size is between *small* and *medium*). After investigation, this was found to be due to the RTES high failure rate and a fitness function that offered little guidance to the search due to a Boolean guard condition. To support the claims above, we followed a rigorous experimental method based on five types of statistical analyses.

Based on our results, we have provided practical guidelines to apply the three testing techniques described in this paper, i.e. RT, ART, and GA. In fact, none of them dominates the others in all testing conditions and they must be, in practice, combined to achieve better results. However, more empirical and theoretical studies are needed to develop more precise, practical guidelines.

One current limitation of our testing approach is that the domains of valid values for the non-deterministic test inputs need to be static: they should be known before test case execution. Research will need to be carried out to design novel testing strategies for non-deterministic inputs that can only be determined at runtime.

Acknowledgements

The work described in this paper was supported by the Norwegian Research Council. This paper was produced as part of the ITEA-2 project called VERDE.

References

1. Douglass, B.P.: Real-time UML: developing efficient objects for embedded systems. Addison-Wesley Longman Publishing Co., Inc., Boston (1997)
2. Utting, M., Legeard, B.: Practical model-based testing: a tools approach. Elsevier, Amsterdam (2007)
3. Chen, T.Y., Kuoa, F., Merkela, R.G., Tseb, T.: Adaptive random testing: The art of test case diversity. Journal of Systems and Software, JSS (in press, 2010)
4. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14(2), 105–156 (2004)
5. Myers, G.: The Art of Software Testing. Wiley, New York (1979)
6. Iqbal, M.Z., Arcuri, A., Briand, L.: Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS) (2010)
7. Clarke, D., Lee, I.: Testing real-time constraints in a process algebraic setting. In: IEEE International Conference on Software Engineering (ICSE), pp. 51–60 (1995)
8. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods in System Design 34(3), 238–304 (2009)
9. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126, 183–235 (1994)
10. En-Nouaary, A.: A scalable method for testing real-time systems. Software Quality Journal 16(1), 3–22 (2008)
11. Miicke, T., Huhn, M.: Generation of optimized testsuites for UML statecharts with time. In: IFIP international conference on testing of communicating systems, pp. 128–143 (2004)
12. Zheng, M., Alagar, V., Ormandjieva, O.: Automated generation of test suites from formal specifications of real-time reactive systems. Journal of Systems and Software (JSS) 81(2), 286–304 (2008)
13. Auguston, M., Michael, J.B., Shing, M.T.: Environment behavior models for automation of testing and assessment of system safety. Information and Software Technology (IST) 48(10), 971–980 (2006)

14. Harman, M., Mansouri, S.A., Zhang, Y.: Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, King's College (2009)
15. Garousi, V., Briand, L.C., Labiche, Y.: Traffic-aware stress testing of distributed real-time systems based on uml models using genetic algorithms. Journal of Systems and Software (JSS) 81(2), 161–185 (2008)
16. Lindlar, F., Windisch, A., Wegener, J.: Integrating model-based testing with evolutionary functional testing. In: International Workshop on Search-Based Software Testing, SBST (2010)
17. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of modeling and simulation. Academic Press, New York (2000)
18. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Transactions on Software Engineering (TSE) 10(4), 438–444 (1984)
19. Arcuri, A., Iqbal, M.Z., Briand, L.: Formal analysis of the effectiveness and predictability of random testing. In: ACM International Symposium on Software Testing and Analysis, ISSTA (2010)
20. Lefticaru, R., Ipate, F.: Functional search-based testing from state machines. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 525–528 (2010)
21. Arcuri, A.: It does matter how you normalise the branch distance in search based software testing. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 205–214 (2010)
22. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1(1), 67–82 (1997)
23. Feller, W.: An Introduction to Probability Theory and Its Applications, 3rd edn. vol. 1. Wiley, Chichester (1968)
24. Arcuri, A.: Longer is better: On the role of test sequence length in software testing. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 469–478 (2010)
25. Cohen, J.: A power primer. Psychological bulletin 112(1), 155–159 (1992)
26. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics 25(2), 101–132 (2000)
27. Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Transactions on Software Engineering 30(1), 3–16 (2004)

Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage

Beatriz Pérez Lamancha[1] and Macario Polo Usaola[2]

[1] Software Testing Centre, Republic University, Montevideo, Uruguay
bperez@fing.edu.uy
[2] Alarcos Research Group, UCLM, Ciudad Real, Spain
macario.polo@uclm.es

Abstract. A Software Product Lines (SPL) is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". Variability is a central concept that permits the generation of different products of the family by reusing core assets. It is captured through features which, for a SPL, define its scope. Features are represented in a feature model, which is later used to generate the products from the line. From the testing point of view, testing all the possible combinations in feature models is not practical because: (1) the number of possible combinations (i.e., combinations of features for composing products) may be untreatable, and (2) some combinations may contain incompatible features. Thus, this paper resolves the problem by the implementation of combinatorial testing techniques adapted to the SPL context.

Keywords: testing, software product lines, combinatorial testing, feature coverage, pairwise.

1 Introduction

A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features which satisfy the specific needs of a particular market segment or mission and which are developed from a common set of core assets in a prescribed way [1]. Products in a line share a set of characteristics (commonalities) and differ in a number of variation points, which represent the variabilities of the products. Software construction in SPL contexts involves two levels: (1) Domain Engineering, which refers to the development of common features and the identification of the variation points; (2) Product Engineering, where each concrete product is built. At this second level, commonalities must be included in the products, and the corresponding variation points must be adequately managed.

Traceability and reuse are fundamental aspects in SPL development and, thus, testing is an essential task in this kind of software development paradigm. In fact, and error introduced in a common part which remains undetected may affect all the products in the line; in the same way, an error in a variation point will be propagated to all the products which include that variation.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 111–125, 2010.

In previous works [2], a framework for model-driven testing in SPL was defined. The framework includes a methodological approach to automate the generation of test models from SPL design models, and specifies a way to deal with variability: given a SPL design, the approach produces a test model which includes enough information to build specific test cases both for the common features of the line, as well as for the specific characteristics of the variation points finally implemented in each product.

However, just as the execution of integration testing is required after unit testing in a classic testing process, features of a SPL must be also tested when they are integrated into a single product; finding no faults in core assets at the Domain Engineering level does not mean that its transformation into a concrete product (generated at the Product Engineering level) does not introduce defects. In the same way, the fact of not discovering errors when an isolated feature is tested does not guarantee that a given product with that very same feature, together with others, will be free of defects, even in those features which, apparently, were previously error-free.

From a testing point of view, testing all the possible feature combinations in a SPL is unfeasible. In a SPL with just 5 features and 4 variants, the number of products that can be generated is 4^5=1024. Testing each possible product is expensive and unrealistic for software industry.

This paper defines a strategy for testing products proceeding from SPL feature models. The strategy uses pairwise as its covering criterion, in the sense that all the pairs of features must be included and tested in at least one product. The Orthogonal Variability Model (OVM, [3]) is used to represent the variation points and its variants. This does not mean any loss of generality in the proposal, since any other metamodel can be used to represent the feature model. In fact, the same rules would be applied to obtain the test suite of products to test.

One of the most widely-used strategies to obtain pairwise coverage is the AETG algorithm [4], which works in polynomial time. In the SPL context, the algorithm must be modified to deal with *requires* and *excludes* relationships between features. If a variant in a feature excludes a variant in another feature, then the pair between both variants must not be present in any product. One of the SPLs we use as a case study consists of a system to play board games over the internet. Thus, we may be dealing with four variation points (Game, Dice, Opponent and Number of Players) and several variants in each ({Ludo, Trivial, Chess, Checkers}, {Dice, No-dice} {Person, Computer}, {2, >2}). *Ludo* or *Trivial* with *No-dice* make no sense, and neither do *Chess* or *Checkers* with *Dice* or with more than two players (*>2*). Restrictions between pairs such as these are not contemplated by AETG and, therefore, the algorithm has been modified to not consider undesired pairs.

This change to the algorithm is not restricted to SPL testing, since it is common to test systems excluding invalid combinations of test values. Pairwise assumes that many errors only arise from the specific interaction of certain values of two or more parameters [5], but in the actual practice of software testing, test cases containing undesired pairs are often removed from the final test suite. For these situations, the improved version of AETG can be also used.

2 Representing Variability in SPL

Variability is a central concept in product family development. It allows for the generation of different products in the family by reusing *core assets*. Variability is

captured through *features*. A feature can be a specific requirement, a selection amongst optional or alternative requirements, or can be related to certain product (functionality, usability, performance, etc) or implementation characteristics (size, execution platform, standards compliance, etc)[6].

Domain engineering techniques are used to systematically extract features from existing or planned members of a product line. Feature trees are used to relate features to each other in various ways, showing sub-features, alternative features, optional features, dependent features or conflicting features [6]. Examples of these methods are FODA [7], FORM [8], FeatuRSEB [9], among others. Figure 1 shows a feature model example.

In this work, the proposal by Pohl et al. [3] is used to manage the variability, defined in their **Orthogonal Variability Model** (OVM). In OVM, variability information is saved in a separate model containing data about variation points and variants. A *variation point* may involve several *variants* in, for example, several products. OVM allows the representation of dependencies between variation points and variable elements, as well as associations among variation points and variants with other software development models (i.e., design artifacts, components, etc.). Variation points and variants are the core concepts of the OVM language. Each variation point offers at least one variant. Additionally, the constraints-associations between these elements describe dependencies between variable elements [3].

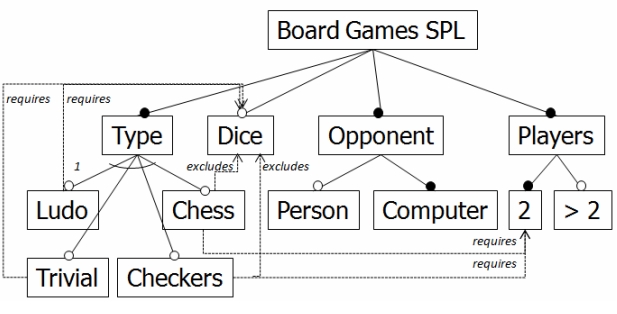


Fig. 1. Feature Model for Board Game SPL

OVM includes a graphical notation: Variation Points are represented by triangles and their variants with a rectangle. Dotted lines represent *optional* variants (i.e., they can be omitted in some products), whereas solid lines represent *mandatory* variants (they are present in all products). The associations between variants may be *requires_V_V* and *excludes_V_V*, depending on whether they denote that a variation *requires* or *excludes* another variation. In the same way, associations between a variation and a variation point may be *requires_V_VP* or *excludes_V_VP*, also to denote whether a variation requires or excludes the corresponding variation point.

Figure 2 shows the OVM model for the board game SPL. The board games share a wide set of characteristics, such as the existence of a board, one or more players, the use of dice, possibility of taking pieces, presence or absence of cards, policies related to the assignment of the turn to the next player, etc. As we showed in the previous

section, there are 4 four variation points (Game, Dice, Number of players and Opponent) and 4, 2, 2, and 2 variants respectively.

In previous works, a specific UML profile to represent OVM models was defined[10]. Figure 3 shows the same information as in Figure 2 but using the profile.

The use of one or another metamodel is independent for the process: Roos-Frantz, Benavides and Ruiz-Cortés[11] have shown that it is possible to use model-to-model transformation in order to generate a target model conforming to an OVM metamodel, preserving all the semantics in the source models.

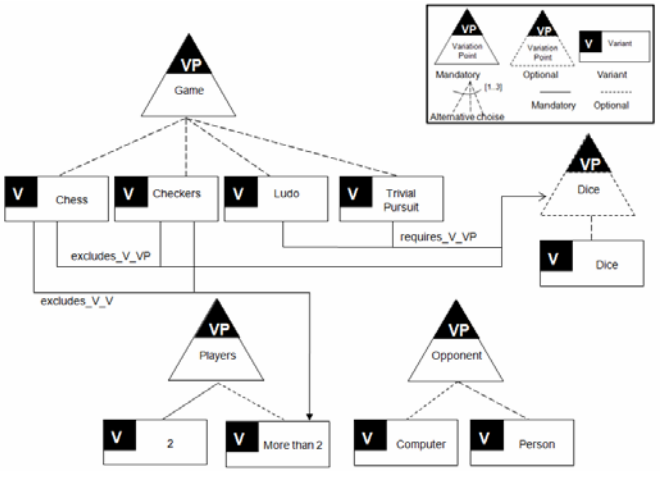


Fig. 2. OVM model for Board Game SPL

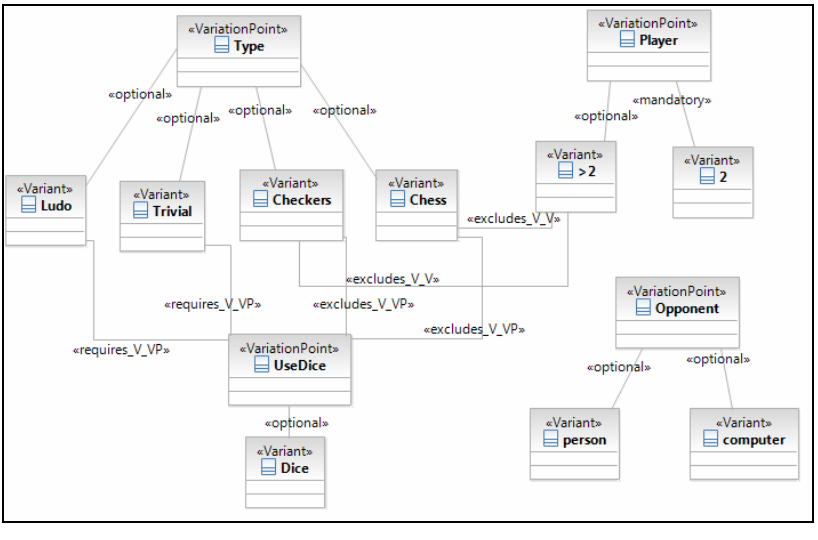


Fig. 3. Board Games SPL using UML profile for OVM

3 Combination Testing Strategies and Related Works

Combination strategies are a class of test-case selection methods where test cases are created by the combination of "interesting values", which have been previously identified by the tester. The input of all these testing strategies is a set of sets (parameters), each with some elements (values). The output is a set of combinations, all of them composed of one element from each input set.

Like many test-case selection methods, combination strategies are based on coverage. In the case of combination strategies, coverage is determined with respect to the use of the parameter values that the tester decides are interesting. Thus, for example, a test suite satisfies *Each-use* (also known as *1-wise*) coverage when each test value is included in at least one test case in the test suite. *Pairwise* (also known as *2-wise*) coverage requires that every possible pair of interesting values of any two parameters be included in some test case. Note that the same test case may cover more than one unique pair of values. A natural extension of pairwise coverage is *t-wise*, which requires that every possible combination of interesting values of *t* parameters be included in some test case in the test suite.

Different test generation strategies have been published for pairwise testing, some of them collected in a survey article by Grindal, Offut and Andler [12]. Since the problem of generating minimum pairwise test sets is NP-complete, different researchers have developed strategies to generate near-minimum pairwise test sets, such as algorithms based on orthogonal arrays [13] or covering Arrays [14]. One important drawback to these two methods is that they can only be applicable to sets (parameters) with the same number of elements (test values), which restricts the actual application of these techniques.

One very interesting approach for pairwise coverage was proposed by Cohen et al. [4], who developed the AETG algorithm for *t-wise* coverage (Figure 4).

Considering the combinatorial strategies in SPL context, Perrouin et al. [15] uses t-wise for feature coverage using SAT solvers, dividing the set of clauses (transformed from a feature diagram) into solvable subsets. They use the features as parameters; each parameter may receive two values (*true* or *false*) to represent the presence or absence of the feature: thus, the combination strategy followed up by these authors may produce much more products to be tested that those required. Actually, the feature model should have information enough to consider the relationship between a variation point and its variants. The authors take into account *mandatory* and *optional* features, the *requires* relationship, but no the *excludes* one. In our approach, variation points are considered as the parameters: instead of having two boolean values for each feature, we process the feature model to consider that each feature variant is a parameter value. Moreover, all the relationships defined in OVM are processed to include or exclude pairs. The use of combinatorial testing to cover features in SPLs has also been the focus of previous works by McGregor[16] and Cohen et al.[17], who address the issue of pairwise testing through orthogonal arrays and covering arrays respectively. However, we consider that these approaches have a key limitation in that they that require all of the features to have the very same number of variants. In our experience (also corroborated by examples found in the literature[18]), this is unrealistic, since features very rarely offer the very same number of variants. Moreover, these works neither consider the *excludes* relationship between features. Indeed, we decided to improve the AETG algorithm because the number of values in each parameter can be variable.

Assume that we have a system with k test parameters and that the *i-th* parameter has *li* different values.

Assume that we have already selected r test cases. We select the $r + 1$ by first generating M different candidate test cases and then choosing one that covers the most new pairs.

Each candidate test case is selected by the following greedy algorithm:

1. Choose a parameter f and a value l for f such that that parameter value appears in the greatest number of uncovered pairs.

2. Let $f_1 = f$. Then choose a random order for the remaining parameters. Then, we have an order for all k parameters $f_1, ..., f_k$.

3. Assume that values have been selected for parameters $f_1, ..., f_k$. For $1 \leq i \leq k$, let the selected value for f_i be called v_i. Then, choose a value v_{k+1} for f_{k+1} as follows.

   For each possible value v for f_k, find the number of new pairs in the set of pairs $\{f_{k+1} = v \text{ and } f_i = v_i \text{ for } 1 \leq i \leq k\}$. Then, let v_{k+1} be one of the values that appeared in the greatest number of new pairs.
   Note that, in this step, each parameter value is considered only once for inclusion in a candidate test case. Also, that when choosing a value for parameter f_{j+1}, the possible values are compared with only the k values already chosen for parameters $f_1,..., f_k$.

Fig. 4. Original explanation of the AETG algorithm for covering pairwise [4]

4 Selection of Products to Test in SPL

Testing all the existing combinations in a feature model is similar to exhaustive testing in traditional software development and is economically unviable. The objective, then, is to select a testing strategy to decide what products will be tested, assuming that these products are representative of the set of all the possible products in the line.

Obviously, if the core assets are tested in isolation, it is less likely to find defects when they are assembled in a product. However, it is necessary to ensure that there are no undesired results when the product is generated. Rather than exhaustive testing, a combinatorial approach can help SPL engineers to decide what combinations of features are more interesting to test, based on feature coverage and feature dependencies.

In our proposal, the variation points are the parameters for the pairwise, whereas variations are the values of the parameters. First, we define how pairs between features are generated from the OVM model and second, how the test cases are selected using the modified version of the AETG algorithm. Each test case is a combination of features, i.e., a product of the line.

4.1 Building the Pairs Set

We use the OVM model to describe the features and the relationships between features. As described in Section 2, OVM is used to exemplify the proposal, since the results of this study can be extrapolated to any other representation of feature models.

Table 1. Features and variants

	features			
variants	type	dice	opponent	players
	ludo	dice	person	2
	chess		computer	moreThan2
	trivial			
	checkers			

There are four parameters in the example of the Board Games SPL: Game, Dice, Players and Opponent. The values for the parameters are the Variations for each Variation Point. Table 1 shows the parameters and its values for the Board Games SPL.

Actually, the information in Table 1 is incomplete, as it is necessary to add the information about the relations between the parameters and their values. Table 1 is augmented with the following information:

- **Variation Point:** If the variation point is optional, then a new value is added. This value states that the entire variation point is not selected for the product. The rule is:

  > If VP is an optional Variation Point with *n* variants, then the VP parameter has *n+1* values: one for each variant and one more for the value "no".

  In the example, the variation point Dice is optional and the "no" value is added.
- **Variants:** In OVM the relationship between a Variation Point and a Variant can be optional, mandatory or alternative. For each case:
  - Optional: The optional variability dependency states that a variant can (but does not need to) be part of a product line application [3]. No values are added for this relationship.
  - Mandatory: The mandatory variability dependency states that a variant must be selected for an application if and only if the associated variation point is part of the application [3]. For example, variant 2 in Figure 2 for the Players variation point is mandatory: then, value *2* can be present in all products of the line (because the Player variation point is also mandatory) and the variant *More than 2* is optional. The rule is:

    > If VP is a variation point with *n* variants, being *k* mandatory and *n-k* optional, then the parameter VP has *(n-k)+1* values, where the first value is the selection of all the *k* mandatory values together, and the *n-k* remaining values are pairs of each optional value with the first value.

    For the example, since value 2 is mandatory, it must be added to the other values: i.e., MoreThan2 and (2,MoreThan2), which is the second value for the parameter Player.
  - Alternative: The alternative choice groups a set of variants that are related through an optional variability dependency to the same variation point and defines the range for the amount of optional variants to be selected for this group [3]. The alternative contains two attributes: *min* and *max*. The rule is:

> If VP is a variation point with *n* optional variants, where the alternative dependency is *[j, k]*, the values for the parameter VP are the result of *Comb(n,i)* where *Comb* is the combinatorial function of *i* values taken from *n* values, with *i = j..k.*

With this information, the table of parameters is built as shown in Table 2.

Table 2. Parameters for pair-wise

	features			
variants	type	dice	opponent	players
	ludo	dice	person	2
	chess	no	computer	2, moreThan2
	trivial			
	checkers			

The next step is to build the tables of pairs between the parameters shown in Table 3.

Table 3. Pairs between parameters

type-dice	type-opponent	type-players	dice-opponent	dice-players	opponent-players
ludo-dice	ludo-person	ludo-2	dice-person	dice-2	person-2
chess-dice	ludo-computer	ludo-2,more2	dice-computer	dice-2,more2	person-2,more2
trivial-dice	chess-person	chess-2	no-person	no-2	computer-2
checkers-dice	chess-computer	chess-2,more2	no-computer	no-2,more2	computer-2,more2
chess-no	trivial-person	trivial-2			
checkers-no	trivial-computer	trivial-2,more2			
ludo-no	checkers-person	checkers-2			
trivial-no	checkers-computer	checkers-2,more2			

The OVM model also states the relationship between variation points or variants belonging to different variation points. The relationship can be:

- **Variant requires variant (requires_V_V):** The selection of one variant v1 in the variation point VP1 requires the selection of another variant vk in the variation point VPk, without taking into account the variants associated. The rule is:

> For each pair *(v1, vj),* where *vj* is different from *vk*, the value *vk* is added to the pair, thus getting *(v1,vj,vk).*

- **Variant excludes variant (excludes_V_V):** The selection of one variant v1 in the variation point VP1 excludes the selection of another variant vk in the variation point VPk, without taking into account the variants associated. The rule is:

> The *(v1, vk)* pair is deleted from the corresponding pairs table.

In the example in Figure 2, the *Chess* variant excludes the *MoreThan2* variant (the same occurs with the *Checkers* variant). Thus, the pairs *(Chess-2,More2)* and *(Checkers-2,More2)* are deleted from the *(Type-Players)* pair table.

- **Variant requires Variation Point (requires_V_VP):** The selection of one variant v1 in the variation point VP1 requires the consideration of a variation point VPk. The rule is:

  If the variation point *VPk* is optional, the value "no" was added as value for the parameter VPk. The *(v1, no)* pair is deleted from the pairs between VP1 and VPk

  In the example in Figure 2, the *Ludo* variant requires *Dice* (the same occurs with the Trivial variant). The pairs *(Trivial,no)* and *(Ludo, no)* are deleted from the pairs between type and dice.
- **Variant excludes Variation Point (excludes_V_VP):** The selection of one variant v1 in the variation point VP1 excludes the consideration of variation point VPk. The rule is:

  If the variation point VPk is optional, the value "no" was added as value for the parameter VPk. All pairs between (v1, vk) are deleted from the pairs between VP1 and VPk except the pair (v1, no)

  In the example of Figure 2, the *Chess* and *Checkers* variants exclude *Dice*: thus, *(Chess, dice)* and *(Checkers, dice)* are deleted from the pairs between type and dice.
- **Variation Point requires Variation Point (requires_VP_VP):** The selection of one variation point VP1 requires the consideration of variation point VPk. The rule is:

  If the variation point VPk is optional, the value "no" was added as value for the parameter VPk. The pair (vi, no) is deleted from the pairs between VP1 and VPk where vi represents all values of VP1

- **Variation Point excludes Variation Point (excludes_VP_VP):** The selection of one variation point VP1 excludes the consideration of variation point VPk. The rule is:

  If the variation point VPk is optional, the value "no" was added as value for the parameter VPk. All pairs between (vi, vk) are deleted from the pairs between VP1 and VPk except the pair (v1, no),

Table 4 shows the resulting pairs between the parameter values, excluding the relationships between features.

Table 4. Pairs between parameters excluding relationships between features

type-dice	type-opponent	type-players	dice-opponent	dice-players	opponent-players
ludo-dice	ludo-person	ludo-2	dice-person	dice-2	person-2
trivial-dice	ludo-computer	ludo-2,more2	dice-computer	dice-2,more2	person-2,more2
chess-no	chess-person	chess-2	no-person	no-2	computer-2
checkers-no	chess-computer	trivial-2	no-computer	no-2,more2	computer-2,more2
	trivial-person	trivial-2,more2			
	trivial-computer	checkers-2			
	checkers-person				
	checkers-computer				

Once the pairs table is built, the AETG algorithm must be modified to remove the undesired pairs from the final products.

4.2 Modifications to the AETG Algorithm

The next step is to calculate the test cases using pairwise. Achieving pairwise coverage requires each pair to be covered by at least one test case. The AETG heuristic algorithm must be adapted to consider feature dependencies.

AETG selects the value for each parameter that appears in most unvisited pairs. The problem in this case is that, after removing the undesired pairs, not all pairs are present in the final set of pairs. Therefore, the algorithm must find the value in each parameter that appears in most unvisited pairs, but taking into account that the pairs between the selected values exist. Considering, for example, the pairs in Table 4, the execution of the original AETG algorithm selects *{ludo, dice, person, 2}* as first test case. The second test case selected will be *{trivial, no dice, computer, 2-moreThan2}*; however, the *(trivial, no dice)* pair is not present in the set of pairs. The original AETG algorithm (Figure 4) is improved in step 3: instead of leaving "the pair selected appears in the greatest number of new pairs", adding "and the pair exists in the pairs set" is required.

The stop condition for the algorithm also must be changed. The original AETG algorithm stops when all pairs in the pairs set have been visited. In our case, pairs may exist that are unreached. This is the case for the pair *(no dice, 2-moreThan2),* which is never visited because is not possible to find a combination of feature values where this pair is valid. Then, this pair remains unvisited at the end of the algorithm. The stop condition is changed and the algorithm stops when the test case selected does not visit any unvisited pair. We have called the AETG algorithm with these improvements *Customizable AETG*.

4.3 Implementation of a *Customizable AETG* Algorithm

Previously, a framework for combinatorial testing called Combinatorial Testing for Software Product Lines (CTSPL) was implemented as a web application[1]. Any of the testing strategies supported by the framework can be resumed as an algorithm which

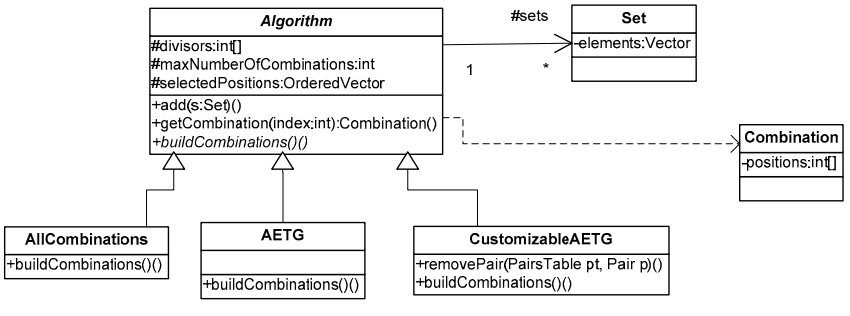


Fig. 5. Partial view of the hierarchical structure of Customizable AETG

[1] http://161.67.140.42/CombTestWeb/

takes a set of sets as input ($S=\{S_1, S_2, \ldots, S_n\}$, which correspond to the parameters or variation points) and produces a set of combinations of the elements in the sets (which correspond to the parameter values, or products in the SPL context). Thus, the algorithm implementing each strategy can be seen as a specialization of an abstract *Algorithm* (Figure 5), which builds its corresponding collection of elements by means of an abstract operation (*buildCombinations*), which is implemented in each specialization.

As seen in the figure, each algorithm holds a collection of sets, which represent the parameters. Moreover, each algorithm has a collection of integers (*selectedPositions*), which hold the positions of the selected combinations.

1. Build *pairTables* for *S*, the set of parameters (*pairTables* does not includes the unrequired pairs).
2. let *c*=combination #0
3. Add *c* to the *selected* set
4. Update *pairTables* with the pairs visited by *c*
5. while there are unvisited pairs in *pairTables and continue*
    1. initialize *c* putting the value which visits more unvisited pairs in *pairTables*
    2. complete *c* with the values of the remaining sets **in such way most pairs are jointly visited and the pairs selected exists in *pairTables***
    3. *if c covers some unvisited pair*
        3.1 Add *c* to the *selected* set
        3.2 Update *pairTables* with the pairs visited by *c*

    else continue := false

Fig. 6. Pseudocode of the Customizable AETG algorithm

Each *Combination* keeps an array of as many integers as there are sets in its *positions* field. Each integer in *positions* represents the index of the selected element from the corresponding set. Given a combination, the *algorithm* extracts the parameter values by visiting its collection of *sets*.

Figure 6 shows a pseudocode of this new version of AETG. Note the changes introduced in the stop condition (step 5) and in the selection of values (step 5.2).

4.4 Description of the Web Application

The web application accepts the description of the elements in the sets (sets are distributed in columns; their elements in rows) and allows the application of any of the implemented combination algorithms. Moreover, the application also accepts xmi files representing the feature model of the SPL. In Figure 7, the user has selected and is ready to submit the xmi file corresponding to the feature model of the Board Games SPL.

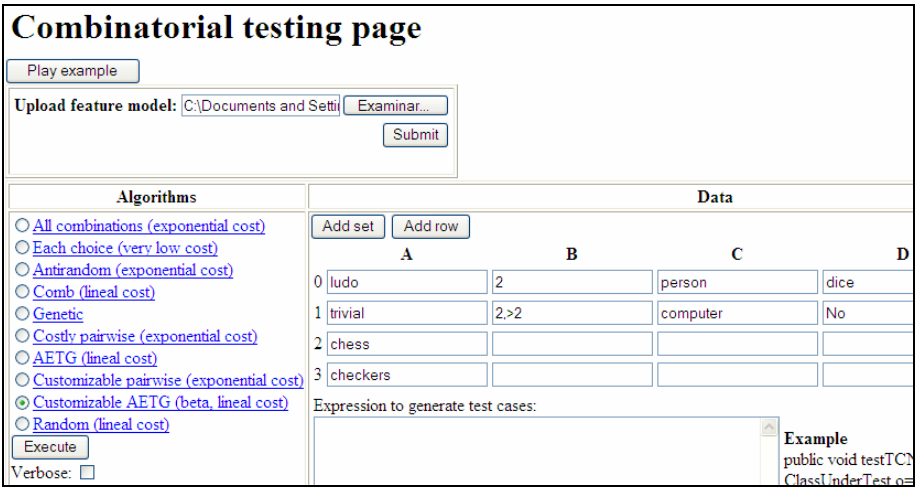


Fig. 7. Uploading the feature model shown in Figure 3

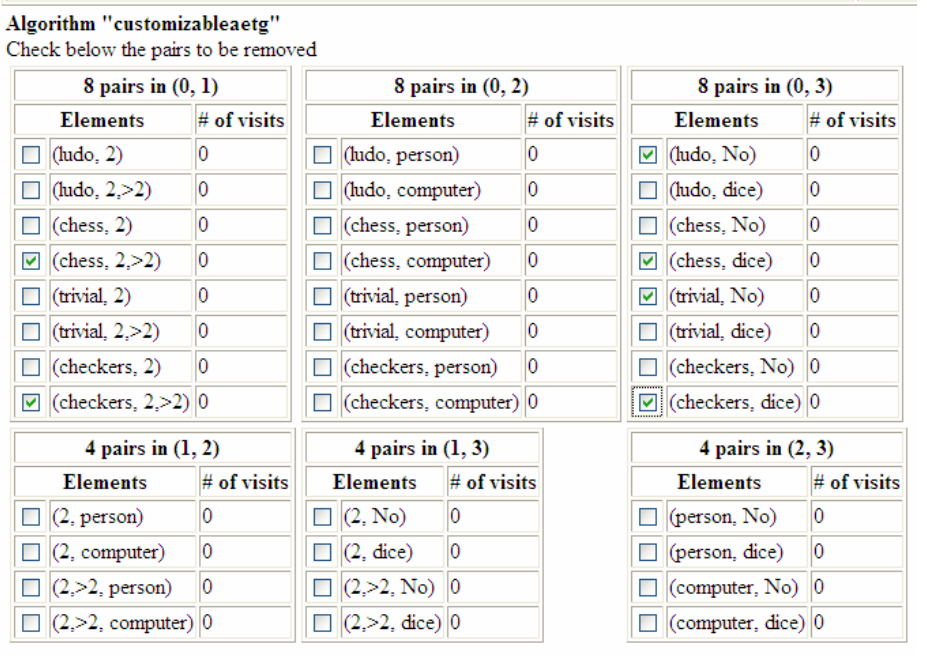


Fig. 8. The user selects the pairs to be removed

Once the application has received the feature model with the xmi file, it analyzes it and shows the pairs tables (Figure 8) leaving the user to select those that should not be included in the final suite. At this time, we are modifying the code of the subsystem in charge of processing the xmi file to detect, via the relationships defined in the model (excludes and requires), which pairs should be removed.

Table 5. Visited pairs and test cases in Customizable AETG

	Visited pairs by value										
Step	ludo	chess	trivial	checkers	dice	no dice	person	computer	2	2- >2	Test Case
1	5	4	5	4	6	6	8	8	8	6	{ludo, dice, person, 2}
2	2	4	5	4	3	6	5	8	5	6	{trivial,dice,computer,2->2}
3	2	4	2	4	0	6	5	5	5	3	{chess,no dice,person,2}
4	2	1	2	4	0	3	3	5	3	3	{checkers, no dice,computer,2}
5	2	1	2	1	0	1	3	2	1	3	{trivial,dice,person,2->2}
6	2	1	1	1	0	1	1	2	1	2	{ludo,dice,computer,2->2}
7	0	1	1	1	0	1	1	1	1	1	{chess,no dice,computer,2}
8	0	0	1	1	0	1	1	0	1	1	{trivial, dice, person, 2}
9	0	0	0	1	0	1	1	0	0	1	{checkers,no dice, person,2}
	0	0	0	0	0	1	0	0	0	1	{chess,no dice, person, 2}

Table 6. Test cases that visit each pair in Customizable AETG

type-dice	test case	type-opponent	test case	type-players	test case	dice-opponent	test case	dice-players	test case	opponent-players	test case
ludo-dice	1,6	ludo-person	1	ludo-2	1	dice-person	1,5,8	dice-2	1,8	person-2	1,3,8,9
trivial-dice	2,5,8	ludo-computer	6	ludo-2,more2	6	dice-computer	2,6	dice-2,more2	2,5,6	person-2,more2	5
chess-no	3,7	chess-person	3	chess-2	3,7	no-person	3,9	no-2	3,4,7, 9	computer-2	4,7
checkers-no	4,9	chess-computer	7	trivial-2	8	no-computer	4,7	no-2,more2		computer-2,more2	2,6
		trivial-person	5,8	trivial-2,more2	2,5						
		trivial-computer	2	checkers-2	4,9						
		checkers-person	9								
		checkers-computer	4								

Then, the user is ready to select any of the provided algorithms (left side of Figure 7) and obtain the results. If s/he selects the Customizable Pair AETG algorithm, the algorithm shows the results.

We will illustrate how the results are reached describing the steps followed by the *Customizable AETG* in Figure 6. The first step in the algorithm is "Build pairTables for S, the set of parameters, the pairTables does not include the restricted pairs", the pairTables is shown in Table 6. At the beginning, the column corresponding to the test case that visits this pair is blank. Table 5 shows the visited pairs in each step of the algorithm. In the first step, the ludo value appears in 5 unvisited pairs (see Table 6). When the combination # 0 ={ludo, dice, person,2}, is selected, Table 6 is updated and for step 2, the ludo value appears now in 2 unvisited pairs.

The algorithm selects the value for each parameter that visits the most pairs. In step 2, it first selects computer because this value appears in 8 pairs; the selected test case up to now is {-,-,computer,-}. Then for the rest of the parameters, the algorithm selects the value that visits the most pairs. The first parameter selected is *Type* and the value trivial is selected because it appears in 5 pairs. The test case is now {trivial,-,computer,-}. For the parameter *Dice*, the value no dice appears 6 times, but the pair (trivial, no dice) does not exist in pairsTable, so the value dice is selected. The test case is {trivial, dice,computer,-}. For the parameter player, value 2, moreThan2 appears 6 times and is selected. The test case is {trivial, dice, computer, 2-MoreThan2}. Once the test case is selected, Table 6 is updated with the visited pairs for the test case.

The algorithm continues 9 more steps and the test cases selected are shown in Table 5. In the last step, only one pair is unvisited, this pair is (no dice, 2-MoreThan2). This pair is unreacheable by a combination of pairs, so in step 10 the

algorithm selects {chess,no dice, person, 2}. Due to the fact that this pair does not visit any unvisited pair, the algorithm stops.

Using the CustomizedAETG algorithm the test cases obtained are shown in Table 5, this mean that the test engineer must test the followings products in the line (where CF refers the set of common features to all the products in the line):

Product 1 = CF U {ludo, dice, person, 2}
Product 2 = CF U {trivial, dice, computer, 2, MoreThan2}
Product 3 = CF U {chess, person,2}
Product 4 = CF U {checkers, computer,2}
Product 5 = CF U {trivial, dice, person, 2, MoreThan2}
Product 6 = CF U {ludo, dice, computer, 2, MoreThan2}
Product 7 = CF U {chess, computer, 2}
Product 8 = CF U {trivial, dice, person, 2}
Product 9 = CF U {checkers, person, 2 }

5 Conclusions

This paper describes the application of combinatorial testing to the context of Software Product Lines. Products proceeding from a SPL consist of different types of combinations of the variants and variation points composing the line. Since exhaustive testing is not viable and, furthermore, many of the possible combinations will not belong to any of the final products, several authors have also approached combinatorial testing strategies for SPL testing, especially applying pairwise coverage. However, even some combinations proceeding from this kind of coverage criterion will not be present in any product (in the Board Games example, neither chess nor checkers will match with more than two players). Thus, the AETG algorithm for pairwise coverage has been modified to remove the unfeasible products from the final suite.

The modified version of the algorithm has been included on a web page, which furthermore makes it possible to upload a feature model described in xmi. The tool loads the variants and variation points and is capable of applying a variety of algorithms. In current SPL practice, there are pairs of combinations which the tester is more interested in testing. Therefore, we are also improving the algorithm to give weight to each pair, in order to more exhaustively test the most important pairs.

Acknowledgments. This research was financed by the projects: PRALIN (PAC08-0121-1374) and MECCA (PII2I09-00758394) from the "Consejería de Ciencia y Tecnología, JCCM" and the project PEGASO/MAGO (TIN2009-13718-C02-01) from MICINN and FEDER. Beatriz Pérez has a grant from JCCM Orden de 13-11-2008.

References

1. Clements, P., Northrop, L.: Software Product Lines - Practices and Patterns. Addison Wesley, Boston (2001)
2. Perez Lamancha, B., Polo, M., Piattini, M.: An automated model-driven testing framework for Model-Driven Development and Software Product Lines. In: 5th Inter. Conference on Evaluation of Novel Approaches to Software Engineering (2010) (to be published)

3. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005)
4. Cohen, D.M., et al.: The combinatorial design approach to automatic test generation. IEEE Transactions on Software Engineering 13(5), 83–89 (1996)
5. Bryce, R., Lei, Y., Kuhn, D., Kacker, R.: Combinatorial testing. In: Software Engineering and Productivity Technologies, pp. 196–208 (2010)
6. Griss, M.: Implementing product-line features by composing component aspects. In: Software Product Line Conference, pp. 222–228 (2000)
7. Kang, K., Cohen, S., Hess, J., Novak, W., Spencer, A.: Feature-oriented domain analysis (FODA) feasibility study. SEI Technical Report CMU/SEI-90-TR-21 (1990)
8. Kang, K., Kim, S., Lee, J., Kim, K., Kim, G., Shin, E.: FORM: A feature oriented reuse method with domain specific reference architectures. Annals of Software Engineering 5(1), 143–168 (1998)
9. Griss, M., Favaro, J., d'Alessandro, M.: Integrating feature modeling with the RSEB. In: Fifth International Conference on Software Reuse, p. 76 (1998)
10. Pérez Lamancha, B., Polo Usaola, M., Piattini, M.: Towards an Automated Testing Framework to Manage Variability Using the UML Testing Profile. In: 4th International Workshop on Automation of Software Test, pp. 10–17 (2009)
11. Benavides, F., Ruiz-Cortés, A.: Feature Model to Orthogonal Variability Model Transformations. A First Step. Actas de los Talleres de las Jornadas de Ing. del Software y BBDD 3(2), 81–90 (2009)
12. Grindal, M., Offutt, J., Andler, S.: Combination testing strategies: A survey. Software Testing Verification and Reliability 15(3), 167–200 (2005)
13. Mandl, R.: Orthogonal Latin squares: an application of experiment design to compiler testing. Communications of the ACM 28(10), 1058 (1985)
14. Williams, A.: Determination of test configurations for pair-wise interaction coverage. In: 13th International Conference on Testing Communicating Systems, pp. 59–74 (2000)
15. Perrouin, G., et al.: Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In: Third International Conference on Software Testing, Verification and Validation, pp. 10–17 (2010)
16. McGregor, J.D.: Testing a Software Product Line. Carnegie Mellon University, Software Engineering Institute. Technical report (2001)
17. Cohen, M., Dwyer, M., Shi, J.: Coverage and adequacy in software product line testing. In: ISSTA workshop on Role of software architecture for testing and analysis, pp. 53–63 (2006)
18. Thum, T., Batory, D., Kastner, C.: Reasoning about edits to feature models. In: 31st International Conference on Software Engineering (2009)

Increasing Functional Coverage by Inductive Testing: A Case Study

Neil Walkinshaw[1], Kirill Bogdanov[2], John Derrick[2], and Javier Paris[3]

[1] Department of Computer Science, The University of Leicester, Leicester, UK
[2] Department of Computer Science, The University of Sheffield, Sheffield, UK
[3] Department of Computer Science, University of A Coruña, A Coruña, Spain

Abstract. This paper addresses the challenge of generating test sets that achieve functional coverage, in the absence of a complete specification. The inductive testing technique works by probing the system behaviour with tests, and using the test results to construct an internal model of software behaviour, which is then used to generate further tests. The idea in itself is not new, but prior attempts to implement this idea have been hampered by expense and scalability, and inflexibility with respect to testing strategies. In the past, inductive testing techniques have tended to focus on the inferred models, as opposed to the suitability of the test sets that were generated in the process. This paper presents a flexible implementation of the inductive testing technique, and demonstrates its application with case-study that applies it to the Linux TCP stack implementation. The evaluation shows that the generated test sets achieve a much better coverage of the system than would be achieved by similar non-inductive techniques.

1 Introduction

The quality of a test set is conventionally measured by the extent to which it exercises the System Under Test (SUT). Various different notions of 'coverage' have been developed for different testing techniques, such as the proportion of branches / basic-blocks covered in structural source code testing, or the number of mutants killed in mutation testing (a comprehensive overview of coverage measures is provided by Zhu *et al.* [1]). Functional coverage is conventionally measured with respect to a program specification. The goal of a functional test set generator is to generate a finite set of tests that is sufficiently large and diverse to fully exercise the specification, and identify any implementation faults or failures in the process.

Unfortunately, specifications that fully encapsulate the desired system behaviour are rarely available, because they are often deemed to be too time-consuming to construct and maintain. If they exist at all, they are usually partial; for instance, it is possible to obtain a list of the main interface functions for a system, perhaps coupled with some of the more important pre-/post-conditions. However, it is unrealistic to presume the existence of a complete and up-to-date specification that could be used to ensure that the underlying system is functionally correct.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 126–141, 2010.

Inductive testing [2,3,4,5,6,7,8,9,10,11,12,13] offers a partial solution to this problem. It is based on the observation that testing and inductive inference are two sides of the same coin. In software testing the challenge is to identify a finite set of tests that will fully exercise some software system. In inductive inference the challenge is to work out what the (hidden) system is from a finite sample of examples of its behaviour. **The success of techniques in either area depends on the depth and breadth of the set of examples or tests**. In testing, a broader test set is more likely to lead to some unexpected / faulty state. In inductive inference, a broader set of examples will lead to a more accurate inferred model. Inductive testing exploits this symmetry between the two areas by inferring specifications from tests; the inferred model shows what has been tested already, so that the test-generator can attempt to find new, contradicting test cases. The process is iterative, and terminates once no further tests can be found that conflict with the inferred specification.

Although the idea of inductive testing is well-established (Weyuker's early work on this [2] dates back to 1983), its application has been restricted to small systems, and has not been widely adopted. This is mainly due to the fact that inductive inference techniques tend to scale poorly, coupled with the fact that some techniques are tied to very specific and expensive systematic test-generation approaches (e.g. Angluin membership queries [14]). Besides the scalability and flexibility problems, reluctance to adopt such techniques could also be due to the fact that they do not explicitly answer the following question: *To what extent does inductive testing improve on the coverage achieved by conventional testing techniques?*.

If it can be shown that inductive testing techniques can (a) be applied to large, realistic systems and (b) that they achieve better functional coverage than conventional black-box testing techniques, such techniques are bound to appeal to the broader testing community. This paper details a case-study that is intended to illustrate those two points. It uses an inductive testing framework that has been developed by the authors [13], which is flexible (allows the user to select their own testing algorithm / implementation), and can infer models from incomplete test sets with the help of heuristic inference algorithms [15,16]. The practicality of the technique is underpinned by the fact that it incorporates the Erlang QuickCheck testing framework, which means that it can be applied to a wide range of Erlang applications and network protocols. In our case, we demonstrate its applicability by automatically constructing a test set for the Linux TCP/IP stack. The main contributions of this paper are as follows:

- The practical applicability of inductive testing is demonstrated in a practical context on a real black-box system – the Linux TCP/IP stack.
- It is shown how the system can be combined with a network-protocol testing framework to enable the testing of arbitrary network protocols.
- The *functional coverage*, and average depth of the tests is measured, and compared against the results from an equivalent non-inductive testing technique.

Section 2 discusses the background; it discusses the general problem of achieving functional test coverage, the inductive testing technique, and its problems. Section 3 presents a high-level view of the inductive testing technique. Section 4 shows how we have implemented the various aspects of the technique, and shows how this has been applied to automatically generate test sets for the Linux TCP/IP stack. It presents results that show how the coverage and depth of test sets generated by the inductive testing technique compare favourably against test sets generated by an equivalent non-inductive version of the technique. Finally, section 5 concludes, and discusses future work.

2 Background

2.1 Context: Achieving Functional Test Coverage without a Model

This paper is concerned with the challenge of producing a test set for a system that is a black-box, and for which there is at best only a partial specification. Exhaustively enumerating every input is infeasible for most realistic programs, and the conventional approach of using random inputs is only likely to exercise those states of the system that are easy to reach by chance. Such approaches often fail to reach those areas of the state-space that are most likely to elicit unexpected program behaviour, such as an unhandled exception or a straightforward program failure.

As opposed to conventional model-based testing, the aim is not to demonstrate that the behaviour of the system is functionally correct. This is impossible without a complete and reliable specification. Instead the aim is to identify a set of test cases that fully exercise the SUT, in the hope that one of these will elicit program behaviour that is obviously incorrect, such as a program failure. Conventional coverage-driven testing approaches aim to fully exercise the SUT in terms of its source code or specification [1]. In the absence of either of those, the aim in our case is to obtain a test set that fully exercises SUT in terms of its observable behaviour.

The definition of "observable behaviour" depends on the characteristics of the SUT. The observable behaviour of a network protocol would be the sequences of outputs that are returned in response to sequences of inputs. The observable behaviour of a continuous function would be the numerical value that is returned for a particular input value. The ultimate aim is to explore the full range of functionality offered by the SUT, where this functionality is manifested in the variation of outputs in response to different inputs.

2.2 Inductive Testing

A test set is deemed to be *adequate* if it achieves a given level of coverage. Many coverage measures exist to suit different testing techniques [1], such as source code branch coverage, or transition coverage in state machines, etc. In our setting we cannot count on access to a specification or on access to the source code.

One solution to the above problem is offered by a technique that was first outlined by Weyuker [2] almost 30 years ago. She related the challenge of identifying an adequate test set to the machine-learning field of inductive inference, where the challenge is to infer a model from a finite set of examples. She observed that the two problems are in effect two sides of the same coin. An accurate model can only be inferred with a sufficiently broad set of examples. Importantly from a testing perspective, a better model will ultimately result in a more comprehensive test set.

Weyuker suggested that this intuitive symmetry could be exploited, and it is this idea that forms the basis for what is referred to here as inductive testing. The idea is as follows: a set of tests can be considered *adequate* if, when fed to an inductive inference engine, the resulting model is equivalent to the SUT[1]. Since then, the idea of combining inductive inference with testing has reappeared in various guises [3,4,5,6,7,8,9,10,17,11,12,13].

Problem: flexibility and scale. Current inductive testing approaches are hampered by the fact they are often tied to very specific systematic testing strategies, which in turn lead to problems in terms of scalability. Testing strategies are designed to produce a test set that will, when used as a basis for inference, produce a model that is *exact*. However, obtaining a sufficiently large test set to do so is often infeasible in practice, especially when tests are expensive to execute. Though cheaper, heuristic approaches have been proposed [13], their applicability has not been demonstrated with respect to a realistic black-box system. None of the proposed techniques have been explicitly evaluated in terms of their ability to achieve functional coverage of substantial black-box systems.

3 A Flexible Inductive Testing Technique

This section describes an inductive testing technique that is designed to be flexible. It permits the developer to choose a model-based testing technique to suit the circumstances (i.e. the complexity of the software system). To account for incomplete test sets, it adopts heuristics to infer models. Every time a model is inferred, the selected testing strategy is invoked to generate a test set from the model, which is executed on the SUT. This process continues until a set of tests has been generated that infer a model for which no further conflicting tests can be generated.

For this paper, we will restrict discussion to systems that can be modelled and tested as deterministic Labelled Transition Systems (LTS's).

Definition 1 (Labelled Transition System (LTS)). *A LTS is a quadruple* (Q, Σ, Δ, q_0)*, where* Q *is a finite set of states,* Σ *is a finite alphabet,* $\Delta : Q \times \Sigma \rightarrow$

[1] Although this is the essential idea, she considered a different problem-setting to the one considered here; her setting presumed that the system was white-box, that there was also a specification, and that the inductive inference engine was inferring executable source code as opposed to arbitrary models.

```
   Input: Prog, Alphabet
   Data: Alphabet, Pos, Neg, test, fail, failedLTS
   Uses: inferLTS(T+, T−), generateTests(LTS)
   Uses: runTest(t, Prog), generateInit(Alphabet)
   Result: LTS
1  Pos ← ∅;
2  Neg ← ∅;
3  LTS ← generateInitLTS(Alphabet);
4  Test ← generateTests(LTS);
5  foreach test ∈ Test do
6  |  (trace, pass) ← runTest(test, Prog);
7  |  if pass then
8  |  |  Pos ← Pos ∪ {trace};
9  |  |  if trace ∈ Σ* \ L(LTS) then
10 |  |  |  LTS ← inferLTS(Pos, Neg);
11 |  |  |  Test ← generateTests(LTS);
12 |  |  |
13 |  else
14 |  |  Neg ← Neg ∪ {trace};
15 |  |  if trace ∈ L(LTS) then
16 |  |  |  LTS ← inferLTS(Pos, Neg);
17 |  |  |  Test ← generateTests(LTS);
18 |  |  |
19 |  end
20 end
21 return LTS
```

Algorithm 1. Basic iterative algorithm

Q is a partial function and $q_0 \in Q$. This can be visualised as a directed graph, where states are the nodes, and transitions are the edges between them, labelled by their respective alphabet elements.

Some parts of this section will refer to the *language* of the LTS. This is defined as follows.

Definition 2 (The Language of an LTS). *For a state p and a string w, the extended transition function $\hat{\delta}$ returns the state p that is reached when starting in state p and processing sequence w [18]. For the base case $\hat{\delta}(q, \epsilon) = q$. For the inductive case, let w be of the form xa, where a is the last element, and x is the prefix. Then $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$.*

Given some LTS A, for a given state $q \in Q$, $L(A, q)$ is the language of A in state q, and can be defined as: $L(A, q) = \{w | \hat{\delta}$ is defined for $(q, w)\}$. The language of a LTS A can be defined as: $L(A) = \{w | \hat{\delta}$ is defined for $(q_0, w)\}$.

The inductive testing process is presented in algorithm 1. It takes the alphabet of the machine as input (i.e. the message types that can label transitions), and uses these to generate its initial most general machine (line 3). This is then fed

to a model-based tester to generate test cases (line 4). For each test case, if it has passed, its trace *trace* is added to the set of traces that are positive Pos and if it fails it is added to the set of negative traces Neg. In either case, the test case is checked against the current LTS model to make sure that it produced the correct result. If this isn't the case, a conflicting test case has been discovered, and a new LTS is inferred to account for the new traces (lines 10 and 16). When this is the case, a new set of test cases $Test$ is generated, and the whole process iterates. The rest of this section provides a more in-depth description of the key components in algorithm 1. The $generateInitLTS$ function is described below, followed by a more detailed description of the processes used to test and infer the models.

To begin with the inductive testing process an initial model is needed. In the algorithm this is generated by the $generateInitLTS$ function on line 3. No prior knowledge is required about the (hidden) state transition structure of the subject system, but it does assume that the set of labels (or possible software inputs) is known. An initial LTS is a simple transition system that is produced where any sequence in Σ^* is valid. This will always consist of a single state, with one looping transition that is labelled by all of the elements in Σ. Formally, $Q = \{q_0\}$, $\forall \sigma \in \Sigma$, $\delta(q_0, \sigma) = q_0$. The tests that are generated from this will can be used by the inference process to produce more refined models.

3.1 Test Generation and Execution

The $generateTests$ function represents a conventional model-based test set generator. It takes as input a model in the form of a LTS and generates a set of tests (paths through the state machine) that are expected to be either be possible or impossible in the implementation.

Given a LTS A and a test case t, the execution of a test case by the $runTest$ function results in a tuple $(test, Pass)$, where $test$ is the test (if it failed, its last element is the point at which it failed), and $Pass$ states whether it passed or failed. The algorithm then matches this against the expected outcome; if $Pass == false$ but $t \in L(A)$, or $Pass == true$ but $t \notin L(A)$, there is a conflict and A has to be re-inferred.

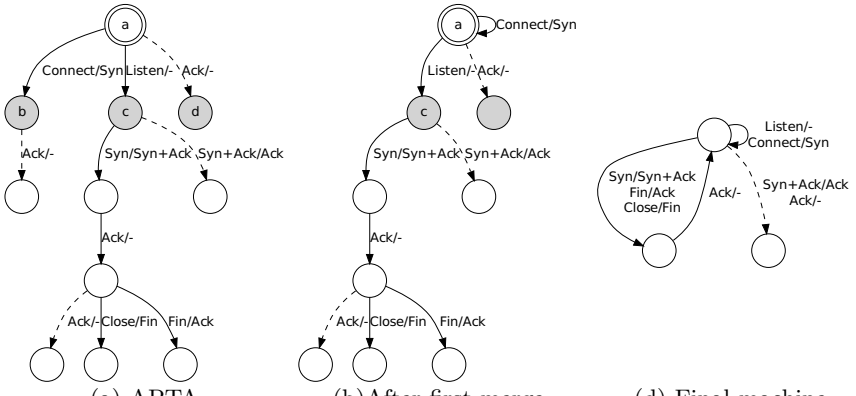
The $generateTests$ function can invoke a range of model-based testing algorithms. One the one hand there are 'formal' testing techniques that aim to guarantee certain levels of coverage of the given state machine. On the other hand, it is possible to adopt less rigorous but cheaper alternatives.

3.2 Model Inference

The inference process is denoted by the $inferLTS$ function. Its purpose is to infer an LTS from a set of test cases. The Evidence Driven State-Merging (EDSM) method [15] is currently accepted to be the leading approach for inferring LTS's from sparse sets of examples.

Conceptually, given a collection of test executions, one would expect that the executions traverse the same states in the SUT multiple times. State-merging

Pos:
$< Listen/-, Syn/Syn + Ack, Ack/-, Close/Fin >$
$< Listen/-, Syn/Syn + Ack, Ack/-, Fin/Ack >$
Neg:
$< Listen/-, Syn + Ack/Ack >$
$< Listen/-, Syn/Syn + Ack, Ack/-, Ack/- >$
$< Connect/SynAck/- >$
$< Ack/- >$



(a) APTA (b)After first merge (d) Final machine

Fig. 1. Augmented Prefix Tree Acceptor and illustration of merging. Dashed transitions are deemed to be invalid, multiple transitions between states are drawn as single transitions with multiple labels.

approaches arrange the presented set of traces into one large tree-structured state machine that exactly represents the executions, and then proceed to merge those states that are deemed to correspond to the same SUT state, with the aim of producing the minimal, most general LTS. The benefit of EDSM approach versus other traditional approaches is that it uses heuristics to choose suitable state-pairs, weighing up the likelihood that two states are equivalent in terms of their outgoing paths of transitions (i.e. their future behaviour). This has been shown to substantially increase the accuracy of the final machine [15,16].

A brief illustration will be provided with respect to a small part of the TCP example from the case study. Let us assume that we have reached a point where six tests have been executed, resulting in the traces Pos and Neg shown in figure 1. These can be aggregated into a single tree - referred to as an *augmented prefix tree acceptor (APTA)* [15], shown in (a). The tree is constructed so that any identical execution-prefixes lead to the same state, and any unique suffixes form unique branches in the tree. The APTA represents the most specific and precise LTS possible, and exactly corresponds to the provided sets of executions, not accepting any other sequences.

The goal of the inference is to identify states in this tree that are actually equivalent, and to merge them. The merging process takes two states q and q'. In effect, the state q' is removed, all of its incoming transitions are routed to

q instead and all of its outgoing transitions are routed from q. Every time a pair of states is merged, the resulting state machine may be non-deterministic (new transitions from q may carry the same labels as old transitions). Non-determinism is eliminated by recursively merging targets of non-deterministic transitions. For a more detailed description, the reader is referred to previous work by Dupont *et al.*[16].

The merging process is iterative - many subsequent merges are required to reach the final machine. At each iteration, a set of state-pairs is selected using the Blue-Fringe algorithm [15] (an effective search-windowing strategy that restricts the set of state pairs to be evaluated at each iteration to a subset that are most likely to be suitable merge-candidates). Each candidate pair is assigned a heuristic score, which indicates the likelihood that the states are equivalent. The score is computed by comparing the extent to which the suffixes of each state overlap with each other[2]. A pair of states is incompatible if a sequence is possible from one state, but impossible from the other - this leads to a score of -1. Any pairs with non-negative scores can potentially be merged. Once the scores have been computed, the pair with the highest score is merged, and the entire process of score-based ranking and merging starts afresh, until no further pairs can be merged.

To provide an intuition of the scoring process, we refer back to the example prefix-tree in Figure 1 (a). State 'a' is marked red (a double-circle in the figure), and the states 'b', 'c' and 'd' are marked blue (shaded in the figure - this is the 'blue fringe'). The EDSM algorithm considers any red-blue pair with a non-negative score as a possible merge-candidate. Pairs (a,c) and (a,d) have a score of 0, because they share no overlapping outgoing label-sequences. Pair (a,b) has a score of 1 (the event $Ack/-$ is impossible from both states) and, since this pair produces the highest score, it is selected to be merged. The result is shown in Figure 1 (b). This process continues until no further valid merges can be identified, and the final machine is shown in (c).

4 Testing the Linux TCP Stack

To illustrate the inductive testing approach, and to demonstrate its ability to improve functional coverage, we use it to explore the behaviour of the Linux TCP stack [19,20]. For this study we assume a certain, limited degree of prior knowledge about the system. We know the set of messages that can be sent to the stack, and how they affect its data-state (these are specified in the TCP RTP documents [19]). For the sake of assessing our inductive testing technique, we will assume that there is no prior knowledge of the order in which these messages can be sent. We also assume that we have no access to the internals of the stack itself, but we can reliably reset it (by restarting the Linux networking service).

The challenge for us is to generate a set of test cases that extensively exercises the functionality of the stack, by eliciting a range of behaviours that is as diverse

[2] The Blue-Fringe algorithm ensures that the suffixes of one state are guaranteed to form a tree (i.e. a graph without loops), which facilitates this score computation.

as possible. This section will show how the induction testing approach is able to elicit a much broader range of behaviour than the standard alternative of attempting random input combinations.

4.1 Inductive Testing Infrastructure

The case study exactly follows algorithm 1. Here we show how the various algorithm functions have been implemented. Specifically we describe the implementation of the three key functions: $generateTests$, $runTest$, and $inferLTS$. The implementation elaborates on earlier work by the authors [13], and can in principle be applied to test any system where the behaviour is based on an LTS (if this is not the case, the $inferLTS$ and $generateTests$ functions have to be adapted accordingly).

This case study involves running tests by sending messages over a network. Due to its comprehensive networking infrastructure, we use the Erlang programming language and its Open Telecom Platform (OTP) libraries [21]. This is exploited by the network-testing interface developed by Paris and Arts [20] (described in detail below). This distributed mechanism can be coupled with the powerful QuickCheck model-based testing infrastructure [22] (also described below), to provide a comprehensive network protocol testing framework.

Generating tests with QuickCheck (the *generateTests* function). For the model-based test set generation, we use the QuickCheck framework [22], a tool that has proved popular for the model-based testing of Erlang and Haskell programs. It has become one of the standard testing tools used by Erlang developers. The model is conventionally provided by a developer, either as a set of simple temporal logic properties, or as an abstract state machine.

We use abstract state machine models in this case study. Abstract state machine models are constructed by taking a labelled transition system (see definition 1). Each transition function in Δ is then associated with a transformation on a data state M, along with a set of optional pre-/post-conditions that have to hold when the function is executed.

QuickCheck generates tests with the help of *generator* functions. These are functions that, for a given data type, will produce a suitable random input. QuickCheck has several built-in generators, which can be used to produce random inputs for simple data types, such as integers, or lists of integers. To illustrate the use of these generators, a very simple property is provided below. The `prop_reverse` property takes a list of integers as input. It tests the `lists:reverse` function by ensuring that, when applied twice to a list, it returns the original list. The call to `list(int())` generates a list of random length, populated with random integers. For a more extensive example, including an abstract state machine example, the reader is referred to Walkinshaw *et al.*[13].

```
prop_reverse() ->
      ?FORALL(Xs,list(int()),
      lists:reverse(lists:reverse(Xs)) == Xs).
```

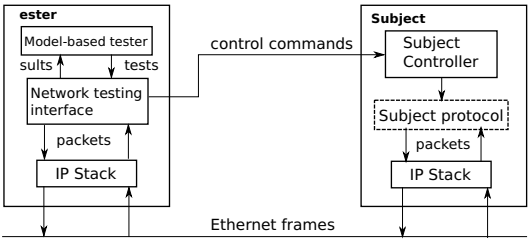


Fig. 2. Communication between tester and subject

The network-testing interface (the *runTest* function). Paris and Arts [20] developed a network protocol testing framework that enables model-based testing of network protocols on remote machines. The basic process is shown in figure 2. Two channels are used to interact with the system under test. One channel is the network itself, messages are synthesised by the tester and sent to the subject, and responses from the subject are sent back, which are monitored by the tester (using a network sniffer). The other channel is an Erlang communication channel that is linked to an Erlang process that is running on the SUT. It is used to control the SUT when necessary. For example if we want to test the TCP protocol in a scenario where the SUT initiates communication, this can be triggered by sending a command via this channel to the controller on the SUT, which will in turn make the stack send the corresponding message across the network link to the tester.

The network-testing interface is implemented on top of QuickCheck. The abstract state machine model keeps track of the necessary data elements (i.e. the last message received from the SUT, the Port number and IP address). For each possible message that can be sent to the SUT, a QuickCheck generator function is used to construct and send the suitable TCP packet and, if appropriate, to wait for a response from the SUT and update the data state in the abstract state machine. For a full description, the reader is referred to Paris and Arts [20].

For the sake of extensively testing TCP stacks with this framework, Paris and Arts produced a comprehensive QuickCheck model of the stack. For the inductive testing process discussed in this paper, we use a skeleton version of this model, having removed the LTS. This is instead replaced with the "universal" LTS, where every function is always possible (as generated by $generateInitLTS$ – see definition in section 3).

LTS Inference (the *inferLTS* function). Walkinshaw *et al.* have developed the openly-available StateChum model inference framework [23,13] that was used to infer the models. It implements the Blue-Fringe state merging approach that is presented in section 3.2. A front-end was developed that, given a template QuickCheck model, would append the appropriate inferred labelled transition system, and so produce a suitable model that could be used to generate test sets for the following iteration. Usually, the output LTS generated by StateChum is displayed on-screen. To enable its use in an inductive testing context, an

extension for StateChum has been developed that converts the output LTS into a QuickCheck model.

4.2 Exploring the TCP Stack Behaviour by Inductive Testing

The three functions (*generateTests*, *runTests* and $inferLTS$) are linked together by a simple Unix Bash script. Test sets are executed until a test fails. When this is the case, a new model is inferred, and the testing process starts afresh. The process will either terminate when no more conflicting tests can be found or by simply putting a limit on the number of test runs.

The basic rationale for inductive testing is that the ability to infer a model provides a picture of what has already been tested, enabling the test set generator to generate more elaborate, diverse test sets the next time round. To assess whether this is the case we compare the test sets that are generated by inductive testing to those that are generated by an identical set-up[3], but without the ability to replace the model with inferred models (i.e. where the model is always stuck on the most general model produced by the $generateInitLTS$ function).

For both runs, the limit was set to 285 iterations (i.e. 285 test sets could be generated and executed). This limit was chosen because it corresponded to a couple of hours of test runs (allowing for time-outs when unexpected packet-sequences resulted in deadlocks). In practice, it would be up to the developer to select a suitable limit, based on the constraints of the project.

Measuring coverage and depth. Every time a test is executed, the sequence is recorded to a text file, and it is prefixed by a "+" if it passed, or a "-" if it failed (a failure leads to the termination of the current test set execution and the generation of a new one). At any given point we are interested in the complete set of tests, i.e. all of the tests that were generated in all previous iterations leading up to that point.

The suitability of a test set is measured by the extent to which it covers the functionality of the SUT, coupled with its ability to reach "deep" states – states that require an elaborate combination of inputs to reach, and are hard to reach by random inputs. The metrics that are used to assess these are presented below:

Functional coverage. There is no accepted means of measuring functional coverage without access to an existing specification. The conventional way to estimate it so far has been to measure code-coverage, however this merely provides a crude approximation of the extent to which the actual functional behaviour of the program has been covered. Indeed, it is the recognition of this fact [2] that spawned the idea of inductive testing in the first place. The model that we infer from the test sets provides a functional perspective on the test sets, and in this work we use the inferred model as a basis for measuring the functional coverage. A larger model means that the test set exercises a broader range of SUT behaviour,

[3] There are several systematic black-box testing techniques that would outperform this naïve approach, but we choose this one for the sake of control, to ensure that the only factor to make a difference in the assessment is the ability to infer models.

because it is able to distinguish between lots of different states (otherwise the inference approach would simply merge them together). So, to assess functional coverage we count the number of individual transitions in the model – these can be of two kinds; conventional state transitions, and state transitions that lead to a failure / termination state.

Although this measure seems to be more appropriate than code-coverage, it should still only be interpreted as an indicator. The EDSM inference technique operates on heuristics, and is therefore open to error. The addition of a test to a test set could under certain circumstances produce a machine that is slightly smaller than the previous version. Nonetheless, as will be shown in the results, such dips tend to be very small, and the measure still presents a good overview of the breadth of the test set.

Test depth. It is generally acknowledged that longer test sequences tend to lead to a higher level of coverage [24]. A long sequence of steps can be required to put a system into a configuration where it is possible to reach particular parts of the system that would remain unreachable otherwise. Every time a test set is executed in this case study, the average length of its tests is recorded. Longer test sequences imply that a test set is reaching states in the system that are harder to reach.

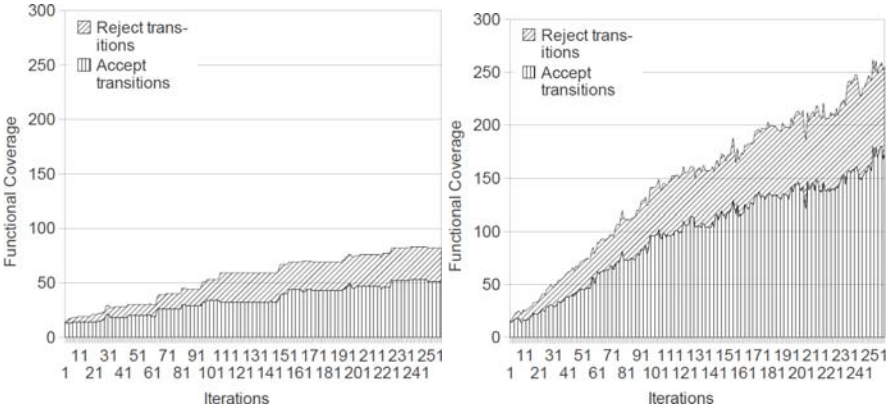
4.3 Results

Figures 3 (a) and (b) compares the functional coverage achieved by inductive testing, and compares it to the level of coverage that is achieved by naïve non-inductive testing. The coverage is plotted for every iteration. This is split between transitions that lead to a failing / terminating state, and transitions that lead to a normal state.

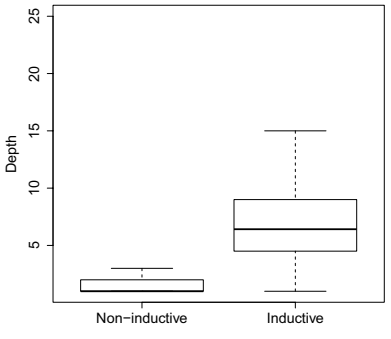
The charts show that the inductive testing technique is better at exploring new system behaviour. It finds a much larger number of unique tests, and does so at a much faster rate. The non-inductive approach never surpasses 83 transitions, whereas this level is achieved within the first 60 iterations by the inductive testing approach. The charts show clearly that, by relying on random tests alone, it is much harder to identify a diverse test set that thoroughly exercises the SUT. The coverage of the system only increases very slowly as the process iterates.

The difference between the test sets generated by the two approaches is also illustrated in figure 4. This figure shows the APTA's that are generated from the final test sets. Figure (a) is generated by the non-inductive version; although it represents a similar number of tests, many of them are simple repetitions, and do not lead to unexplored behaviour, meaning that the APTA is much smaller. Figure (b) on the other hand is much larger; there are far fewer repetitions amongst the tests because the inferred model encourages the discovery of new test sequences that build on the outcomes of previous tests.

Inductive testing leads to the generation of longer test sequences, which has been shown to be a significant factor in the ability to achieve high levels of test coverage [24]. Without the ability to build on the knowledge gained through



(a) Non-inductive testing (b) Inductive testing



(c) Box-plot that compares average depths of test cases

Fig. 3. Functional coverage and comparison of average test case depth

previous test cases, the non-inductive approach relies on randomly finding sequences to find "deep" states – states that can only be reached by a particular test sequence. In inductive testing, these sequences are remembered; they form part of the inferred model, making it easy for subsequent tests to revisit and explore these deep states. Consequently, there is a substantial difference in the average test sequence length, as shown in the box plots in figure 3(c). The limits of the box denote the upper and lower quartiles, the line through the middle represents the median, the whiskers mark the maximum and minimum values, and the small circles denote outliers.

Summary. The use of test outputs to infer a model of system behaviour is a valuable tool for the further exploration of system behaviour, and is practical. This is demonstrated in the results above. With no prior knowledge about the possible sequences in which messages can occur, the inductive testing approach has nonetheless managed to build an extensive and diverse set of test cases that

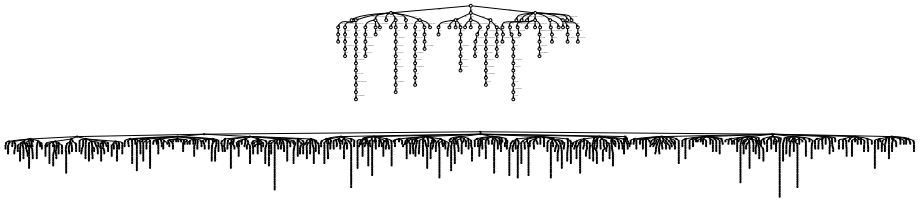


Fig. 4. Visual comparison of the APTAs generated for the non-inductive tests (on top) and the inductive tests (below)

exercise the TCP stack. In comparison to the non-inductive approach, the set of test cases is much more diverse, and also manages to consistently reach states at a greater depth.

5 Conclusions and Future Work

The main aims of this paper were to show that inductive testing can be applied to realistic black-box systems, and to demonstrate that inductive testing can achieve a better functional coverage of the system than conventional non-inductive strategies. The presented inductive testing technique is flexible; it is not necessarily tied to a specific inference or testing technique, but a framework is provided that enables the two to interact, and to feed off each other. For our case study, we selected a realistic system; the Linux TCP/IP stack. We used an heuristic inductive inference technique that has been shown to deal well with incomplete test sets [15,16,23], and we combined this with a simple black-box network testing framework [20] that is built on top of the well-established QuickCheck Erlang testing framework [22].

The approach is flexible; different combinations of inference and testing techniques could produce better results. However, the main purpose was to demonstrate that inductive testing is a practical technique, and can produce better, more efficient test sets than naïve black-box alternatives. The best combinations of techniques will invariably depend on the characteristics of the SUT, and is one of the main areas we intend to investigate in future work.

The test set generation technique that was used to generate the test cases (both for the non-inductive and inductive cases) was deliberately simple, to ensure that any improvements in the results were not an artefact of the test-generation technique, but were entirely due to the inferred model. In practice, it makes sense to combine the model induction with a more sophisticated test-generation procedure. The QuickCheck framework has several facilities that can enable this. For example, it is possible to associate probabilities with different transitions in the LTS, to ensure that certain areas of the machine are tested more often than others. Future work will look into exploiting such features, to emphasise the exploration of new areas in the SUT that have not been explored by previous tests.

In our case study, the number of iterations were restricted to 285. This sufficed to serve its purpose of showing that inductive testing produces better test sets than non-inductive testing. However, it does raise the question of when the process should be terminated in general; when is the test set good enough? This is the question that originally motivated Weyuker to devise the inductive testing process [2]. She envisaged that test generation should continue until no tests could be found that conflict with the inferred model (or program in her case). In practice, the complexity of the SUT and the selected testing technique may make it unrealistic to achieve this goal. The convention is to declare a time-limit, and to simply execute as many tests as possible [17]. Even if this approach is adopted, this work has shown that the inferred model can be used to provide an estimation of functional coverage during the testing process, and there is a strong argument to be made for the fact that this is more suitable than the convention of using code-coverage as an approximation.

Acknowledgements. Walkinshaw and Bogdanov are supported by the EPSRC REGI (EP/F065825/1) and STAMINA (EP/H002456/1) grants. Derrick and Walkinshaw are supported by the EU FP7 ProTest project.

References

1. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys 29(4), 366–427 (1997)
2. Weyuker, E.: Assessing test data adequacy through program inference. ACM Transactions on Programming Languages and Systems 5(4), 641–655 (1983)
3. Bergadano, F., Gunetti, D.: Testing by means of inductive program learning. ACM Transactions on Software Engineering and Methodology 5(2), 119–145 (1996)
4. Zhu, H., Hall, P., May, J.: Inductive inference and software testing. Software Testing, Verification, and Reliability 2(2), 69–81 (1992)
5. Zhu, H.: A formal interpretation of software testing as inductive inference. Software Testing, Verification and Reliability 6(1), 3–31 (1996)
6. Harder, M., Mellen, J., Ernst, M.: Improving test suites via operational abstraction. In: Proceedings of the International Conference on Software Engineering ICSE 2003, pp. 60–71 (2003)
7. Xie, T., Notkin, D.: Mutually enhancing test generation and specification inference. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 60–69. Springer, Heidelberg (2004)
8. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 175–189. Springer, Heidelberg (2005)
9. Raffelt, H., Steffen, B.: Learnlib: A library for automata learning and experimentation. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 377–380. Springer, Heidelberg (2006)
10. Bollig, B., Katoen, J., Kern, C., Leucker, M.: Smyle: A tool for synthesizing distributed models from scenarios by learning. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, Springer, Heidelberg (2008)
11. Shahbaz, M., Groz, R.: Inferring mealy machines. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009, vol. 5850, pp. 207–222. Springer, Heidelberg (2009)

12. Raffelt, H., Merten, M., Steffen, B., Margaria, T.: Dynamic testing via automata learning. STTT 11(4), 307–324 (2009)
13. Walkinshaw, N., Derrick, J., Guo, Q.: Iterative refinement of reverse-engineered models by model-based testing. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 305–320. Springer, Heidelberg (2009)
14. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 75, 87–106 (1987)
15. Lang, K., Pearlmutter, B., Price, R.: Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In: Honavar, V.G., Slutzki, G. (eds.) ICGI 1998. LNCS (LNAI), vol. 1433, pp. 1–12. Springer, Heidelberg (1998)
16. Dupont, P., Lambeau, B., Damas, C., van Lamsweerde, A.: The QSM Algorithm and its Application to Software Behavior Model Induction. Applied Artificial Intelligence 22, 77–115 (2008)
17. Pacheco, C., Lahiri, S., Ernst, M., Ball, T.: Feedback-directed random test generation. In: Proceedings of the International Conference on Software Engineering (ICSE 2007), pp. 75–84. IEEE Computer Society, Los Alamitos (2007)
18. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley, Reading (2001)
19. Postel, J.: Transmission control protocol. Technical Report 793, DDN Network Information Center, SRI International, RFC (September 1981)
20. Paris, J., Arts, T.: Automatic testing of tcp/ip implementations using quickcheck. In: Erlang 2009: Proceedings of the 8th ACM SIGPLAN workshop on Erlang, pp. 83–92. ACM, New York (2009)
21. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (July 2007)
22. Claessen, K., Hughes, J.: Quickcheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proceedings of the International Conference on Functional Programming (ICFP), pp. 268–279 (2000)
23. Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Reverse Engineering State Machines by Interactive Grammar Inference. In: 14th IEEE International Working Conference on Reverse Engineering, WCRE (2007)
24. Arcuri, A.: Longer is better: On the role of test sequence length in software testing. In: Proceedings of the International Conference on Software Testing, Verification and Validation, ICST 2010 (2010)

FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution

Kiran Lakhotia[1], Nikolai Tillmann[2],
Mark Harman[1], and Jonathan de Halleux[2]

[1] University College London, CREST Centre, Department of Computer Science, Malet Place, London, WC1E 6BT, UK
[2] Microsoft Research, One Microsoft Way, Redmond WA 98052, USA

Abstract. Recently there has been an upsurge of interest in both, Search–Based Software Testing (SBST), and Dynamic Symbolic Execution (DSE). Each of these two approaches has complementary strengths and weaknesses, making it a natural choice to explore the degree to which the strengths of one can be exploited to offset the weakness of the other. This paper introduces an augmented version of DSE that uses a SBST–based approach to handling floating point computations, which are known to be problematic for vanilla DSE. The approach has been implemented as a plug in for the Microsoft Pex DSE testing tool. The paper presents results from both, standard evaluation benchmarks, and two open source programs.

1 Introduction

It is widely believed that automating parts of the testing process is essential for successful and cost effective testing. Of the many approaches to automated test data generation in the literature, two important and widely studied schools of thought can be found in the work on Search–Based Software Testing (SBST) and Dynamic Symbolic Execution (DSE). This paper proposes an augmented approach to tackle the problem of floating point computations in DSE. It has been implemented as an extension to the Microsoft DSE testing tool Pex.

SBST and DSE have different strength and weaknesses. For instance, SBST provides a natural way to express coverage based test adequacy criteria as a multi–objective problem [6], while DSE is better suited to discover the structure of the input to the system under test. Yet there has been little work in trying to combine SBST and DSE.

Inkumsah and Xie [8] were the first authors to propose a framework (EVACON) combining evolutionary testing with DSE. Their framework targets test data generation for object oriented code written in JAVA. They use two existing tools, eToc [18], an evolutionary test data generation tool, and jCUTE [16], a DSE tool. eToc constructs method sequences to put a class containing the method under test, as well as non–primitive arguments, into specific desirable states. jCUTE is then used to explore the state space around the points reached by eToc.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 142–157, 2010.

Majumdar and Sen [10] combined random testing with DSE in order to address the state problem in unit testing. A random search is used to explore the state space, while DSE is used to provide a locally exhaustive exploration from the points reached by the random search.

Neither of the two approaches addresses the issue of floating point computations in DSE, which originate from limitations of most constraint solvers. Constraint solvers approximate constraints over floating point numbers as constraints over rational numbers. Because computers only have limited precision, solutions which are valid for rational numbers are not always valid when mapped to floating point numbers. Thus, Botella et al. [2] proposed a constraint solver based on interval analysis in order to symbolically execute programs with floating point computations. A limitation of their work is that it does not consider combinations of integers and floating point expressions.

The approach proposed in this paper differs from previous work in that it provides a general framework for handling constraints over floating point variables. It is also the first paper to propose a combination of DSE with SBST in order to improve code coverage in the presence of floating point computations.

The rest of the paper is organized as follows: Section 2 presents a brief overview of SBST and DSE, and the two search algorithms implemented. Section 3 describes the DSE tool Pex and how it has been extended with arithmetic solvers for floating point computations. The empirical study used to evaluate the solvers, analysis of the results, and threats to validity are presented in Section 4. Section 5 concludes.

2 Background

2.1 Dynamic Symbolic Execution

In DSE [5,17] the goal is to explore and execute the program to achieve structural coverage, based on analysis of the paths covered thus far. DSE combines symbolic and concrete execution. Concrete execution drives the symbolic exploration of a program, and runtime values can be used to simplify path constraints produced by symbolic execution to make them more amenable to constraint solving. Consider the example shown in Figure 1 and suppose the function is executed with the inputs `a=38` and `b=100`. The execution follows the `else`-branch at the `if`-statement. The corresponding path condition is $(int)\ Math.Log\ (a_0) \neq b_0$, where a_0 and b_0 refer to the symbolic values of the input variables `a` and `b` respectively. In order to explore a new execution path, the path condition is modified, for example by inverting the last constraint, *i.e.* $(int)\ Math.Log\ (a_0) = b_0$. However, suppose that the expression `(int)Math.Log(a)` cannot be handled by a particular constraint solver. In order to continue testing, DSE replaces the expression `(int)Math.Log(a)` with its runtime value, for example 3. The path condition can thus be simplified to $3 == b_0$, which can now be solved for b_0. Executing the program with the (updated) input values `a=38` and `b=3` traverses the `then`-branch of the `if`-statement.

```
void testme2(double a, int b)
{
   if( (int)Math.Log(a) == b )
      //
}
```

Fig. 1. Example used for demonstrating dynamic symbolic execution and search–based testing

2.2 Search–Based Testing

The field of SBST began in 1976 with the work of Miller and Spooner [11], who applied numerical optimization techniques to generate test data which traverses a selected path through the program. In SBST, an optimization algorithm is guided by an objective function to generate test data. The objective function is defined in terms of a test adequacy criterion. Recall the example from Figure 1 and suppose the condition in the `if`-statement must be executed as true. A possible objective function is $|(int)Math.Log(a) - b|$. When this function is 0, the desired input values have been found. Different functions exist for various relational operators in predicates [9] and objective functions have been developed to generate test data for a variety of program structures, including branches and paths [19].

2.3 Alternating Variable Method

A simple but effective optimization technique [7], known as the Alternating Variable Method (AVM), was introduced by Korel [9] in the 1990s. It is a form of hill climbing and works by continuously changing an input parameter to a function in isolation. Initially all (arithmetic type) inputs are initialized with random values. Then, so called *exploratory moves* are made for each input in turn. These consist of adding or subtracting a delta from the value of an input. For integral types the delta starts off at 1, *i.e.*, the smallest increment (decrement). When a change leads to an improved fitness value, the search tries to accelerate towards an optimum by increasing the size of the neighborhood move with every step. These are known as *pattern moves*. The formula used to calculate the delta added or subtracted from an input is: $\delta = 2^{it} * dir * 10^{-prec_i}$, where it is the repeat iteration of the current move (for pattern moves), dir either -1 or 1, and $prec_i$ the precision of the i^{th} input variable. The precision applies to floating point variables only (*i.e.* it is 0 for integral types). It denotes a scale factor for the size of a neighborhood move. For example, setting the precision ($prec_i$) of an input to 1 limits the smallest possible move to ± 0.1. Increasing the precision to 2 limits the smallest possible move to ± 0.01, and so forth.

Once no further improvements can be found for an input, the search continues optimizing the next input parameter, and may recommence with the first input if necessary. In case the search stagnates, *i.e.* no move leads to an improvement, the search restarts at another randomly chosen location in the search–space.

This is known as a *random restart* strategy and is designed to overcome local optima and enable the AVM to explore a wider region of the input domain for the function under test.

2.4 Evolution Strategies

Evolution Strategies (ES) date back to the 1960s and were popularized in the early '70s by Rechenberg [14] and Schwefel [15]. They belong to the family of Evolutionary Algorithms (EA). In an EA, a population of individuals *evolves* over a number of generations. Between each generation, genetic operators are applied to the individuals, loosely representing the effects of mating (crossover) and mutation in natural genetics. The net effect of these operations is that the population becomes increasingly dominated by better (more fit) individuals.

In an ES, an individual has at least two components: an object vector (*i.e.* containing the inputs to the function under test) and a strategy parameter vector. The strategy parameters 'evolve' themselves as the search progresses. Their role is to control the strength of the mutations of the object vector, so that the ES can self–adapt to the underlying search landscape. For example, if the search is far away from a global optimum, large mutations enable it to make faster progress towards an optimum. Conversely, the closer the search is to an optimum, smaller mutations may be necessary to reach the optimum.

3 Implementation

We will now describe how the AVM from Section 2.3 and the ES from Section 2.4 have been implemented as a Pex extension called FloPSy (search-based **Flo**ating **P**oint constraint solving for **Sy**mbolic execution). The extension is open source and available online from the codeplex website; URL: http://pexarithmeticsolver.codeplex.com.

3.1 Pex

Pex [17] is a test input generator for .NET code, based on DSE; test inputs are generated for parameterized unit tests, or for arbitrary methods of the code under test.

Pex can symbolically represent all primitive values and operations, including floating point operations and calls to pure functions of the .NET `Math` class. Note that unlike traditional DSE, Pex only simplifies constraint systems by using concrete values when the code under test invokes an environment-facing method, *i.e.* a method that is defined outside of .NET, and not part of the `Math` class.

Pex implements various heuristics to select execution paths. The constraint solver Z3 [3] is used to decide the feasibility of individual execution paths. Z3 is a Satisfiability Modulo Theories (SMT) solver, which combines decision procedures for several theories. However, Z3 does not have a decision procedure for floating point arithmetic. When Pex constructs constraint systems for Z3, all floating

point values are approximated by rational numbers, and most operations are encoded as uninterpreted functions. Pex supports *custom arithmetic solvers* to work around this restriction.

Path selection and path constraint solving. Given a program with input parameters, Pex generates test inputs in an iterative fashion. At every point in time, Pex maintains a representation of the already explored execution tree defined by the already explored execution paths of the program. The branches of this tree are annotated by the branch conditions (expressed over the input parameters). In every step of the test input generation, Pex selects an already explored path prefix from the tree, such that, not all outgoing branches of the last node of the path prefix have been considered yet. An outgoing branch has been considered when it has been observed as part of a concrete execution path, when it has been classed as infeasible, or when constraint solving timed out. Pex then constructs a constraint system that represents the condition under which the path prefix is exercised, conjoined with the negation of the disjunction of all outgoing branch conditions. If a solution of this constraint system can be obtained, then we have found a new test input which – by construction – will exercise a new execution path which was not part of the already explored execution tree. The path is then incorporated into the tree.

The successive selection of path prefixes plays a crucial role for the effectiveness of test generation. If the program contains unbounded loops or recursion, the execution tree might be infinite, and thus the selection of path prefixes must be fair in order to eventually cover all paths. Even if loops and recursion are bounded, or if the program does not contain loops and recursion, a fair selection is still important to achieve high code coverage fast. Pex implements various heuristic strategies to select the next path prefix [21], incorporating code coverage metrics and fitness functions.

Each constraint system is a Boolean-valued expression over the test inputs. Pex' expression constructors include primitive constants for all basic .NET data types (integers, floating point numbers, object references), and functions over those basic types representing particular machine instructions, *e.g.* addition and multiplication, and the functions of the .NET `Math` class. Pex uses tuples to represent .NET value types ("structs") as well as indices of multi-dimensional arrays, and maps to represent instance fields and arrays, similar to the heap encoding of ESC/Java [4]: An instance field of an object is represented by a *field map* which associates object references with field values. For each declared field in the program, there is one location in the state that holds the current field map value. An array type is represented by a class with two fields: a length field, and a field that holds a mapping from integers (or tuples of integers for multi-dimensional arrays) to the array elements. Constraints over the .NET type system and virtual method dispatch lookups are encoded in expressions as well. Most of Pex' expressions can be mapped directly into SMT theories for which Z3 has built-in decision procedures, in particular propositional logic, fixed sized bit-vectors, tuples, arrays, and quantifiers. Floating point values are functions that are a notable exception.

Extension mechanism for floating point constraint solving. If some constraints refer to floating point operations, then Pex performs a two-phase solving approach: First, all floating point values are approximated by rational numbers, and most operations are encoded as uninterpreted functions, and it is checked whether the resulting constraint system is satisfiable; if so, a model, *i.e.* a satisfying assignment, is obtained. (Note that this model includes a model for the uninterpreted functions, which might not reflect the actual floating point semantics.) Second, one or more custom arithmetic solvers are invoked in order to correct the previously computed model at all positions which depend on floating point constraints.

Consider for example the following method.

```
void foo(double[] a, int i){
  if (i >= 0 && i < a.Length &&
      Math.Sin(a[i]) > 0.0) {
    ...
  }
}
```

In order to enter the `then`-branch of the `if`-statement, the three conjuncts must be fulfilled. Using Z3, we may obtain a model, *i.e.* a satisfying assignment, where $i = 0$ and a.`Length` $= 1$ and $a[i] = 0/1$ (0 or 1) and `Math.Sin`$(0/1) = 1/1$ is a solution of the constraint system, when approximating floating point values with rational numbers, and treating `Math.Sin` as an uninterpreted function (which gets an interpretation as part of the model).

As this model does not hold with the actual interpretation of `Math.Sin` over floating point values, in a second phase custom arithmetic solvers are invoked. Here, there is a single relevant arithmetic variable x, which represents the heap location `a[i]`, and a single relevant arithmetic constraint `Math.Sin`$(x) > 0.0$. The relevant constraints and variables are computed by a transitive constraint dependency analysis, starting from the constraints which involve floating point operations. To avoid reasoning about indirect memory accesses by the custom arithmetic solver, each symbolic heap location, *e.g.* `a[i]` is simply treated as a variable, ignoring further dependent constraints relating to the sub-expressions of the symbolic heap location.

Default behavior for floating point constraints. By default, Pex version 0.91 employs two custom arithmetic solvers: a solver based on trying a fixed number of (pseudo) random values, and a solver based on an early implementation of the AVM method. To conduct the experiments presented later in Section 4, all custom solvers can be selectively enabled and disabled.

3.2 Custom Solvers

The extension for the custom arithmetic solvers can be enabled by including the `PexCustomArithmeticSolver` attribute at a class, or, at the assembly level (via `[assembly: PexCustomArithmeticSolver]`) of the assembly being tested with Pex.

Table 1. Relational operators used in Pex and corresponding distance functions. K is a failure constant (set to 1).

Operator	Pex representation	Distance function
$a = b$	$a = b$	if $\|a - b\| = 0$ then 0 else $\|a - b\|$
$a \neq b$	$(a = b) = 0$	if $\|a - b\| \neq 0$ then 0 else K
$a < b$	$a < b$	if $a - b < 0$ then 0 else $a - b$
$a \leq b$	$(b < a) = 0$	if $a - b \leq 0$ then 0 else $a - b$
$a > b$	$b < a$	if $b - a < 0$ then 0 else $b - a$
$a \geq b$	$(a < b) = 0$	if $b - a \leq 0$ then 0 else $b - a$

Fitness Function. The fitness function used by the custom solvers is defined in terms of the comparison operators that appear in the constraints, and is similar to what is commonly referred to as the *branch distance* measure in SBST [19]. The inputs to the fitness function are the original set of constraints, alongside the current model constructed by the custom solvers. The function computes a distance value for each constraint, and the overall fitness is the sum of these values. The computation of the distance value depends on the comparison operators in the constraints, as shown in Table 1.

AVM. The AVM starts by building a vector of symbolic variables that forms the basis of the exploratory moves. An index variable is used to keep track of the element in the vector currently being optimized. Further, each element has an associated *precision* variable. The precision is used to control the size of a neighborhood move, as described in Section 2.3, and is initialized to `0` for all variables, regardless of type (*i.e.* `float`,`double`, etc.).

Neighborhood Moves
Every delta is computed as a double precision variable before being converted to the type of the variable (*i.e.* decimal, integer etc.) currently being modified. For an exploratory move, *dir* (from Section 2.3) starts off as -1. If the move is rejected, *dir* is changed to 1. If neither exploratory move is accepted (*i.e.* produces a lower fitness value than the current best model), the AVM moves on to the next element in its vector and resets *dir* to -1. Conversely, if a move is accepted, the AVM continues with pattern moves for as long as there is an improvement in fitness values. Every pattern move simply increases the variable *it* from Section 2.3 by 1. When a pattern move is rejected, the AVM restarts with exploratory moves for the first element in its vector.

Once the AVM reaches the last element in its vector, and each exploratory move of that element is rejected, the search is stuck (either on a plateau or local optima).

Precision of Floating Point Variables
One possible cause for the search to get stuck is that the size of the neighborhood move is too coarse. Therefore, before performing a random restart, the AVM checks if the precision of a variable (see Section 2.3) can be increased. In C#, single–precision variables (*i.e.* `float`) contain about 7 decimal digits of precision,

while double–precision variables (*i.e.* `double`) contains about 15 decimal digits of precision. The AVM only changes the precision ($prec_i$) for a variable, if adding the value 10^{-prec_i} has an effect, or, $prec_i$ is less than 7 or 15 for single and double precision type inputs respectively.

Random Restarts
The main strategy for overcoming local optima is to perform a random restart. Two types of restarts are considered. The first is a global, and the second a local restart. In a global restart, all variables in the AVM's vector are assigned new random values. This is likely to place the starting point for the next hill climb far away from the local optimum where the search got stuck. While this is desirable in many cases, it is not an ideal strategy when a global optimum is surrounded by many local optima. Chances are the search will just get stuck at the same local optimum again. Therefore, the AVM also uses a local restart, which is designed to stay in the vicinity of the current search space while still being able to escape from the optimum.

In a local restart, a random number r (between 0 and 1) is created for each variable. This number is 'scaled' by the formula $10^{-prec_i} * r$, where $prec_i$ is the current precision of the i^{th} variable. The 'scaled' random number is then added to the existing value of the variable. If a local restart does not enable the search to make further progress, a global restart is performed. Thus, the AVM alternates between local and global restarts.

ES. Most EAs allow a wide range of configuration options and ES are no different. A user can configure the solver through various environment variables. `es_solver_parents` controls the size of the parent population μ (default: 15); `es_solver_offspring` sets the size of the offspring population (default: 100); `es_solver_recomb` sets the recombination strategy; `es_solver_mut` sets the mutation strategy.

The ES solver starts by creating an initial parent population of μ individuals. Each individual contains a model and a strategy parameter vector such that every variable in the model has a corresponding strategy parameter. The parameters are initialized to 1 and all the variables in the models are assigned random values.

The main loop of the ES solver involves recombination, mutation and selection steps. If a user specified a recombination strategy, two parents are repeatedly recombined to produce a single offspring until the offspring population is full. Then, a mutation operator is applied to each offspring in turn. The mutation operators first mutate the strategy parameter(s), and then the input variables. Once all offspring have been evaluated, they are combined with the parent population and ranked according to their fitness value. The top ranked individuals are then chosen to replace the parent population, thus forming the next generation of parents. We will now describe the reproduction operators in more detail.

Recombination. One of the genetic operators in an ES is the recombination operator where two or more parents are combined to produce one offspring. The ES solver supports the following recombination strategies:

Discrete
In a discrete recombination, two parent individuals are chosen uniformly from the parent population. Then, each variable in the offspring is assigned the value from either parent. Parents have an equal probability of contributing towards the offspring. The same applies for the strategy parameter vector, that is, the offspring receives each strategy parameter from either parent with equal probability.

Global Discrete
This recombination strategy is similar to discrete recombination. The only difference is that for each variable (and each strategy parameter), a new set of parents is chosen (uniformly).

Intermediate
During an intermediate recombination, two parents are chosen uniformly. For each variable in the offspring, the values of the corresponding parent variables are added together, and the sum is multiplied by 0.5, before being assigned to the offspring. The same is done for each strategy parameter.

Global Intermediate
Similarly, in a global intermediate recombination, each variable and each strategy parameter are chosen from a new set of parents, before combining them in the same way as described in the intermediate strategy.

Mutation. Traditionally mutation was the main means of reproduction in ES. The custom solver supports the following mutation strategies:

Single
In a single mutation strategy an individual only uses one strategy parameter, σ, for all variables. In addition, the mutation operator contains a learning parameter τ. It implements the following equations:

$$\tau := 1/\sqrt{n} \tag{1}$$

$$\sigma' := \sigma * \exp(\tau * N(0,1)) \tag{2}$$

$$v_i := v_i + \sigma' * N_i(0,1), \qquad i = 1, \ldots, n \tag{3}$$

First, the strategy parameter σ of an offspring is mutated to produce σ'. $N(0,1)$ denotes a random number from a standard univariate normal distribution. The mutated strategy parameter is then used to control the mutation of the variables. For each variable (from $1, \ldots, n$), a new random number from a standard univariate normal distribution is multiplied by σ', and then added to the existing value of the variable.

Multi
This mutation strategy contains two learning parameters, a global one, τ_0, and a local one, τ. Further, each variable uses its own strategy parameter. The following equations are implemented:

$$\tau_0 := 1/\sqrt{2 * n} \tag{4}$$

$$\tau := 1/\sqrt{2 * \sqrt{n}} \tag{5}$$

$$\sigma_i' := \sigma_i * \ \exp\ (\tau_0 * N(0,1) + \tau * N_i(0,1)), \qquad i = 1, \ldots, n \tag{6}$$

$$v_i := v_i + \sigma_i' * N_i(0,1), \qquad i = 1, \ldots, n \tag{7}$$

4 Empirical Study

The empirical study was split into two parts. The first part contains a set of benchmark functions which are commonly used to evaluate optimization algorithms [12]. The second part consists of two real world programs comprising 152,372 lines of C# code (as reported by the SLOCCount tool [20]).

4.1 Benchmark Functions

Details of the benchmark subjects are recorded in Table 2. The optimization problem is to find the minimum of each function (*i.e.* 0). The functions were formulated as a single `if`-statement, with the `then`-branch declared as a test data generation goal for Pex. 100% block coverage of the methods indicates that the Pex goal has been reached.

Table 2. Summary of the benchmark functions and their C# representation

Function	C# representation
Beale	(1.5 - x1 * (1 - x2)) == 0
Freudenstein And Roth	(-13 + x1 + ((5 - x2) * x2 - 2) * x2) + (-29 + x1 + ((x2 + 1) * x2 - 14) * x2) == 0
Helical Valley Function	double theta(double x1,double x2) { if(x1 > 0) return Math.Atan(x2 / x1) / (2 * Math.PI); else if (x1 < 0) return (Math.Atan(x2 / x1) / (2 * Math.PI) + 0.5); else return 0; } (10 * (x3 - 10 * theta(x1, x2))) == 0 && (10 * (Math.Sqrt(x1 * x1 + x2 * x2) - 1)) == 0 && x3 == 0
Powell	(Math.Pow(10, 4) * x1 * x2 - 1) == 0 && (Math.Pow(Math.E, -x1) + Math.Pow(Math.E, -x2) - 1.0001) == 0
Rosenbrock	Math.Pow((1 - x1), 2) + 100 * (Math.Pow((x2 - x1 * x1), 2)) == 0
Wood	(10 * (x2 - x1 * x1)) == 0 && (1 - x1) == 0 && (Math.Sqrt(90) * (x4 - x3 * x3)) == 0 && (1 - x3) == 0 && (Math.Sqrt(10) * (x2 + x4 - 2)) == 0 && (Math.Pow(10, -0.5) * (x2 - x4)) == 0

4.2 Real World Open Source Programs

Details for the two open source programs are shown in Table 3. Both programs are written in C# and were chosen specifically because they contain arithmetic operations over floating point variables. Alglib [1] is a numerical analysis and data processing library, and QLNet [13] is a library for quantitative finance operations.

Table 3. Details of the open source programs

Program	SLOC	Pex Methods	Tested
Alglib	62,271	608	514
QLNet	90,101	3,123	3,068
Total	152,372	3,731	3,582

4.3 Experimental Setup

The benchmark functions, Alglib, and QLNet source files were assembled into a single Visual Studio (VS) project each. We then created three test projects via the *Pex* → *Create Parameterized Unit Tests* command. This generates a `PexMethod` for every public method in the original source code. A `PexMethod` represents a parametrized unit test and serves as an entry point for the DSE.

If the arguments to a test function contain complex type objects, Pex may need help in constructing meaningful inputs in order to achieve a higher level of code coverage. Pex provides various mechanisms to that effect such as factory methods and the Moles framework. For the experiments, none of the generated parameterized unit tests were modified, and no factory methods or 'moled' objects were used.

The fitness budget for the random search used by Pex, the AVM and ES was limited to 100,000 fitness evaluations. The ES was configured to have 15 parents, produce 100 offspring per generation, use a *Global Discrete* recombination strategy and the *Single* mutation operator described in Section 3.2.

Setup for Benchmarks
Pex contains a number of options to bound its exploration. For example, the default time out for Pex's constraint solver is 1 second. We increased this limit to 10,000 seconds for the constraint solvers (`/mcst` option) and also increased the time limit for an exploration to the same amount. Then, we ran Pex for each `PexMethod` with: 1) only the constraint solver Z3 enabled, 2) a random search and Z3 enabled, 3) the AVM and Z3 enabled, 4) the ES and Z3 enabled.

Apart from the configuration with only Z3 enabled, we repeated the runs 30 times for each benchmark, due to the stochastic nature of both the AVM and the ES. A list of 30 random number seeds was used to seed the random number generator in Pex (by setting the environment variable `er_random_seed`). Repeating experiments for a stochastic algorithm samples its behavior for a given problem, and thus reduces the risk that any observed change in effectiveness is due to chance.

Setup for Open Source Programs
For the open source programs it was necessary to restrict the execution time of Pex and its constraint solvers to make the study scalable. Many of the functions tested contained unbounded loops and complex constraints over arithmetic types, which slow down the exploration of a program. Therefore, we allowed Pex 60 seconds per `PexMethod` (`/to` option), and set the time out for its constraint solvers to 20 seconds (`/mcst` option).

Further, because the programs comprised 3,731 functions, we did not repeat the experiments 30 times for each function. We believe that the risk of any observed difference when using the custom solvers being due to chance, is sufficiently mitigated by using such a large pool of functions.

4.4 Analysis

Benchmarks

Figure 2 summarizes the results in terms of block coverage for the benchmark functions. Overall, the ES solver is the most effective, achieving 100% block coverage for 5 out of 6 functions. It is also the only algorithm that finds the optimum of the Rosenbrock function, which has a large, almost flat valley near the optimum. The AVM fails to reach the Pex goal for the Rosenbrock function, and also fails in 3 out of 30 runs for the Beale function. Neither solver achieves full coverage of the Powell function.

These figures show that the custom solvers can improve the coverage of Pex for floating point computations. However, they also show that the mapping between rational numbers and floating point numbers is not always a problem in practice. For 3 out of 6 functions Pex was able to achieve 100% block coverage without using any heuristics on top of Z3.

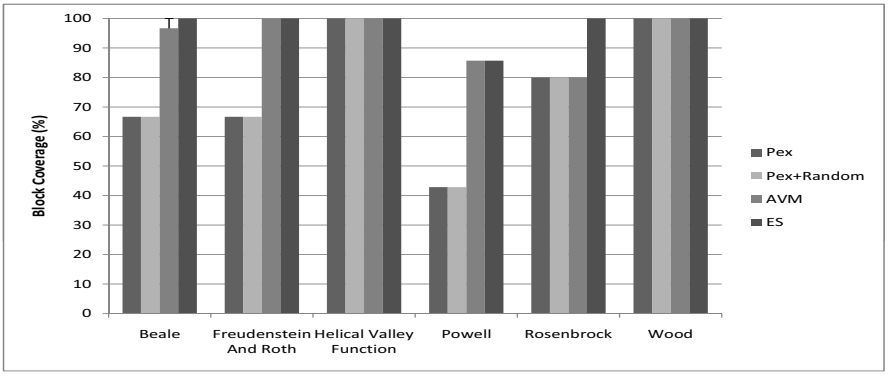


Fig. 2. Block coverage achieved with the different configurations of Pex on the benchmark problems. 100% coverage means the testing goal has been reached.

Open Source Programs

Next, we looked at two open source programs to gain a better understanding of what benefits the custom solvers might offer to a user in practice. During this study, the random search in Pex remained enabled, and we turned on the AVM and ES solvers on top of it; *i.e.* they only get invoked when Z3 or the random search fail to find a solution.

Table 4 reports the coverage for Pex (with a random search), Pex and the AVM, and Pex and the ES solver enabled. Coverage was measured using the reports produced by Pex and shows the ratio of blocks covered to the total number of blocks discovered by Pex (during its DSE). The 'Failed Methods' column

Table 4. Block coverage achieved by Pex, Pex+AVM, and Pex+ES for the open source programs. Coverage is reported as a percentage of blocks discovered by Pex and blocks covered. The 'Failed Methods' columns indicate the number of methods for which Pex was terminated after 1 minute. Any blocks covered in those methods are not counted. The 'Time' column reports the total wall clock time spent inside the AVM and ES solvers.

Program	**Pex**		**Pex+AVM**			**Pex+ES**		
	Cov.	Failed Methods	Cov.	Failed Methods	Time	Cov.	Failed Methods	Time
Alglib	43.86%	32	41.04%	40	02:42:42	44.13%	77	01:53:15
QLNet	43.54%	24	44.46%	31	01:06:02	46.84%	45	00:54:39

indicates the number of methods for which Pex was terminated abnormally after 60 seconds. Those methods do not contribute to the coverage reported, *i.e.* blocks covered during runs which were terminated, are ignored.

Strikingly, the results show that the coverage with the AVM solver enabled, is less than with the custom solvers disabled. The ES solver only marginally increases the coverage compared to Pex's default solvers (*i.e.* Z3 and random search). When the custom solvers are enabled, they will, on average, take around 5 seconds to either find a solution or exhaust their fitness budget. However, as can be seen from Table 5, there are instances where the AVM (438 times) and ES (310 times) were terminated after 20 seconds. Obviously, any time spent in one of the custom solvers is no longer available to Pex to explore different parts of the program. A random search is very fast, and if it fails, Pex can quickly 'move on'. Thus, it can spend more time exploring execution paths which do not depend on floating point computations, thereby increasing the overall level of coverage.

The results from Table 4 suggest that the custom solvers proposed in this paper should only be enabled if the code is known to produce many constraints over floating point variables. In such an event, sufficient resources, in terms of fitness evaluations as well as runtime, should be allocated for the solvers to be effective (*e.g.* as was the case for the benchmark functions). One has to also bear in mind that we do not know how many of the constraints passed to the custom solvers were infact infeasible. Since the solvers have no way of checking, infeasible constraints will cause the solvers to exhaust either their fitness budget or their time out, thus wasting valuable exploration time.

Another interesting question is how often constraints over floating point variables are a problem for DSE in practice. Table 5 shows that, for Alglib, over a third of its methods contain constraints over floating point numbers that could not be solved by either Z3 or a random search. For the QLNet library on the other hand, this figure is much smaller at around 3% of its methods. This suggests that it very much depends on the type of application being tested. Nevertheless, the number of constraints per application (first column in Table 5) which cannot be solved by a random search or Z3, is large enough to present a problem in practice, especially, since this number will be higher if Pex is given more exploration time.

The final part of the study was to analyze the data types of the variables involved in floating point constraints, and the arithmetic operators over them.

Table 5. This table shows how often either the AVM or ES solvers have been invoked by Pex. 'UM' stands for 'Unique Methods' and shows the number of methods that contained one or more constraints which could not be solved by Z3 or a random search. The 'Succ.' columns show how often the AVM or ES solvers were successful, while 'Timeouts' counts the number of times the AVM or ES were terminated after 20 seconds. The 'Fail' column lists the number of unsuccessful attempts by the AVM or ES to find a solution. The numbers in brackets show the number of methods for which the custom solvers were invoked, but did not receive any variables to optimize from Pex.

Program	**Pex+AVM**				**Pex+ES**			
	Inv. (UM)	Succ.	Timeouts	Fail	Inv. (UM)	Succ.	Timeouts	Fail
Alglib	1828 (220)	646	398	784 (269)	1170 (172)	521	153	496 (269)
QLNet	557 (93)	37	40	480 (0)	495 (87)	10	0	485 (0)
Success rate	28.64%				31.89%			

Figure 3 shows that constraints passed to the AVM and ES solvers only included variables of type `double` and integral types (*e.g.* `short`, `int` etc.). Thus, it is important that any approach dealing with constraints over floating point variables can handle mixed floating point and integral type constraints.

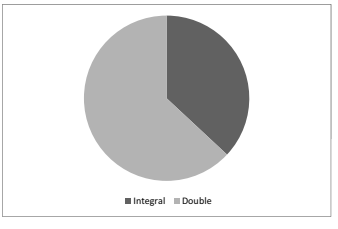


Fig. 3. Distribution of the data types of the variables passed to the AVM and ES solvers

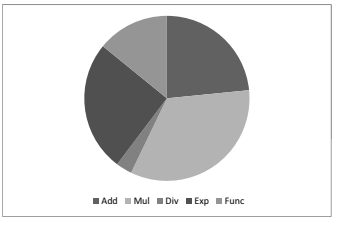


Fig. 4. Distribution of arithmetic operators in the constraints passed to the AVM and ES solvers

Figure 4 summarizes the observed arithmetic operators in the constraints attempted by the AVM and ES solvers. Most commonly the constraints involved multiplication of input variables, followed by the power operator (*e.g.* x^y). These two operators are likely to produce non–linear constraints, and thus are especially hard for constraint solvers. The third most frequent operator was addition of two variables, followed by system library calls involving floating point variables, such as `Double.IsNaN`.

4.5 Threats to Validity

Naturally there are threats to validity in any empirical study such as this. The first issue to address is the internal validity of the experiments, *i.e.* whether there has been a bias in the experimental design that could affect the obtained

results. One potential source of bias comes from the settings used for Pex and the custom solvers. For the open source programs, the time out for Pex explorations was set at 60 seconds, while constraint solvers were given 20 seconds per invocation. These limits, together with the fitness budget of 100,000 evaluations, were necessary to control the scope of the study. However, as a result, the benefit of using the custom arithmetic solvers was diminished, because the functions tested were so complex that they required longer execution times and bigger fitness budgets.

Another potential source of bias comes from the inherent stochastic behavior of the meta–heuristic search algorithms used in the custom solvers. The most reliable (and widely used) technique for overcoming this source of variability is to perform tests using a sufficiently large sample of result data. In order to ensure a large sample size, experiments for the benchmark functions were repeated 30 times. The open source programs comprised 3,731 functions, thus also providing a large pool of data from which to draw observations.

A further source of bias includes the selection of the functions used in the empirical study, which could potentially affect its external validity, *i.e.*, the extent to which it is possible to generalize from the results obtained. The benchmark functions have been used to evaluate different optimization algorithms before, thus were considered a good candidate to test Pex and each custom solver (implementing an optimization algorithm) in the extreme. The open source programs were chosen because they represent non-trivial sized programs and because they contain complex floating point arithmetic. As with any empirical study such as this, caution is required before making any claims as to whether these results would be observed on other functions. More experiments are needed to validate or refute such claims.

5 Conclusions

This paper has presented an extension to DSE for floating point computations, based on SBST techniques. The extension has been implemented as a plug in for the Microsoft Pex DSE testing tool. Results from a set of benchmark functions show that it is possible to increase the effectiveness of what might be called "vanilla DSE". However, a study on two open source programs also shows that for the solvers to be effective, they need to be given adequate resources in terms of wall clock execution time, as well as a large fitness budget. Otherwise any increase in coverage is, at best, marginal. More experiments are needed to check if longer exploration times for Pex make the custom solvers more effective.

Acknowledgments

Kiran Lakhotia is funded by EPSRC grant EP/G060525/1. Mark Harman is supported by EPSRC Grants EP/G060525/1, EP/D050863, GR/S93684 & GR/T22872 and also by the kind support of Daimler Berlin, BMS and Vizuri Ltd., London.

References

1. Alglib. Alglib, http://www.alglib.net/
2. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Softw. Test, Verif. Reliab 16(2), 97–121 (2006)
3. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Conference on Programming language design and implementation, pp. 234–245 (2002)
5. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Conference on Programming Language Design and Implementation, pp. 213–223. ACM, New York (2005)
6. Harman, M., Lakhotia, K., McMinn, P.: A multi-objective approach to search-based test data generation. In: GECCO 2007, pp. 1098–1105 (2007)
7. Harman, M., McMinn, P.: A theoretical and empirical study of search based testing: Local, global and hybrid search. IEEETransactions on Software Engineering 36(2) (to appear, 2010)
8. Inkumsah, K., Xie, T.: Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In: ASE 2007, pp. 425–428 (2007)
9. Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering 16(8), 870–879 (1990)
10. Majumdar, R., Sen, K.: Hybrid concolic testing. In: ICSE 2007, pp. 416–426. IEEE Computer Society, Los Alamitos (2007)
11. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering 2(3), 223–226 (1976)
12. Moré, J.J., Garbow, B.S., Hillstrom, K.E.: Testing unconstrained optimization software. ACM Trans. Math. Software 7(1), 17–41 (1981)
13. QLNet. QLNet, http://www.qlnet.org/
14. Rechenberg, I.: Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Frommann-Holzboog, Stuttgart (1973)
15. Schwefel, H.-P.: Numerical optimization of Computer models. John Wiley & Sons Ltd., Chichester (1981)
16. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
17. 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)
18. Tonella, P.: Evolutionary testing of classes. In: ISSTA 2004, pp. 119–128 (2004)
19. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43(14), 841–854 (2001)
20. Wheeler, D.A.: More than a gigabuck: Estimating GNU/Linux's size (2001), http://www.dwheeler.com/sloc/
21. Xie, T., Tillmann, N., de Halleux, P., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: International Conference on Dependable Systems and Networks, DSN 2009 (2009)

Test Data Generation for Programs with Quantified First-Order Logic Specifications

Christoph D. Gladisch

University of Koblenz-Landau
Department of Computer Science
Germany
gladisch@uni-koblenz.de

Abstract. We present a novel algorithm for test data generation that is based on techniques used in formal software verification. Prominent examples of such formal techniques are symbolic execution, theorem proving, satisfiability solving, and usage of specifications and program annotations such as loop invariants. These techniques are suitable for testing of small programs, such as, e.g., implementations of algorithms, that have to be tested extremely well.

In such scenarios test data is generated from test data constraints which are first-order logic formulas. These constraints are constructed from path conditions, specifications, and program annotation describing program paths that are hard to be tested randomly. A challenge is, however, to solve quantified formulas. The presented algorithm is capable of solving quantified formulas that state-of-the-art satisfiability modulo theory (SMT) solvers cannot solve. The algorithm is integrated in the formal verification and test generation tool KeY.

1 Introduction

Testing has been influenced in the last decade by formal methods. Prominent examples of such formal techniques are symbolic execution, theorem proving, satisfiability solving, and the usage of formal specifications and program annotations such as loop and class invariants. Formal testing techniques can achieve a high code coverage or they can generate a low number of tests that very likely exhibit software faults. Such techniques generate test data constraints which are first-order logic (FOL) formulas. These constraints are constructed from path conditions, specifications, and program annotation and describe program paths that are hard to be tested randomly.

Satisfiability modulo theory (SMT) solvers are state-of-the-art techniques for generating models of FOL formulas. A model is a FOL interpretation in which a formula evaluates to true. A major bottleneck is, however, the handling of quantifiers (see, e.g., [21,22]). SMT solvers can often create models for quantified formulas if *one* theory is involved. Quantifiers and multiple theories, however, often lead to problems that are not in the decidable fragments of the solvers. In such cases an SMT solver cannot generate a model for the formula.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 158–173, 2010.

Java + JML

```
public class C{
 private String[] s;
 /*@ invariant
      s.length>=10;*/
 ...
}
```

Java + JML

$$\left(\begin{array}{l}\forall o : \texttt{C}.o \neq \texttt{null} \rightarrow \\ \quad \forall i : \texttt{int}.0 \leqslant i \leqslant \texttt{length}(\texttt{s}(o)) \rightarrow \\ \qquad \texttt{acc}_{[]}(\texttt{s}(o), i) \neq \texttt{null}\end{array}\right) \quad (1)$$

$$\forall o : \texttt{C}.o \neq \texttt{null} \rightarrow \texttt{length}(\texttt{s}(o)) \geqslant 10 \quad (2)$$

Fig. 1. (left) A field declaration and a class invariant; (right) Quantified formulas occurring in test data constraints generated by KeY

For example, Figure 1 shows a Java class with a field declaration and a Jml [18] specification of a class invariant. From the field declaration and the class invariant the tool KeY [2,16] generates the formulas (1) and (2), respectively. These formulas are part of test data constraints. In this approach Java-fields and arrays are modelled as uninterpreted functions in first-order logic, hence, FOL interpretations and program states are the same concept. Formula (1) follows Jml's semantics and expresses that the elements of the array field `s` are not `null`. Formula (2) expresses the class invariant, that for all objects of class `C` the array `s` has 10 or more elements.

When generating a test for some method of class `C`, the test data constraints have to be satisfied by the test data. The problem is, however, that state-of-the-art SMT solvers, concretely we have tested Z3 [5], CVC3 [1], Yices [9], are not capable to solve (1) or (2). Although SMT solvers can solve quantified formulas in certain cases, (1) and (2) are not in the decidable logic fragment of the solvers. Note, that a different translation of the code in Figure 1 could create formulas that are solvable by an SMT solver, but the general problem of solving quantified formulas remains.

The contribution of this paper is not a technique to derive test cases or test data constraints. Those techniques are cited in Section 1.1. Instead, our contribution is the handling of quantified formulas for solving provided test data constraints. We propose a model generation algorithm that is not explicitly restricted to a specific class of formulas and which can therefore solve more general formulas than SMT solvers can solve.

The basic idea of our approach is to generate a partial FOL model in which a quantified formula that we want to eliminate evaluates to true. SMT solvers can then solve the remaining ground formulas. The representation of a model in our approach is a program. Our technique generates candidate programs and checks if a candidate program satisfies the test data constraint. For instance, in order to satisfy formula (2) we could generate the following Java statement

```
for (C o : Cs){o.s = new String [10] ;}        (3)
```

where `Cs` is a collection of objects of class `C`, and verify it against formula (2). A programming language such as Java is, however, not *directly* suitable for this

task because (a) function and predicate symbols are usually not part of such languages and (b) loop invariants would have to be generated for the verification proof. A language and a calculus that are suitable for our purpose exist, however, in the verification system KeY. The language consists of so-called *updates*.

Structure of the paper. In the following we describe the background of our work as well as related work. In Section 2 the underlying formalism of our approach is introduced. The main section is Section 3 where we describe our algorithm. In Section 4 we report on experiments with our approach and provide conclusions and further research plans in Section 5.

1.1 Background and Related Work

The work presented in the paper has been developed in the KeY project [16]. The KeY system [2,16] is a verification and test generation tool for JAVA. The tool can automatically generate JUnit tests from proof structures [10]. The test data is generated from FOL constraints which combine execution path conditions with annotations such as method specification, class invariants, and loop invariants [13,14]. Hence, the approach is a grey-box testing technique. So far we have used the theorem prover Simplify [7] to generate test data but the so-generated test data is not always guaranteed to satisfy the constraints as will be shown below. On the other hand, we found that more recent SMT solvers such as Z3 [5], CVC3 [1], Yices [9] are not capable to solve the constrains either, which was the motivation for this work.

There exist several other tools that follow similar ideas as the KeY tool to generate test data constraints and have therefore similar requirements on test data generation. KUnit [6] is an extension of Bogor/Kiasan which combines symbolic execution, model checking, theorem proving, and constraint solving. Check'n'Crash generates JUnit tests for assertions that could not be proved using ESC/Java2 [4]. Java PathFinder is an explicit-state model checker and features the generation of test inputs [24]. Non-trivial FOL formulas may also occur in functional testing or model-based testing. A survey of search-based test data generation techniques is given in [19]. These techniques are powerful for traditional testing approaches but they do not handle test data generation for constraints with quantified FOL formulas.

The main contribution of our work is the handling of quantifiers. One has to distinguish between different quantifiers in different contexts, namely between those that can be skolemized and those that cannot be skolemized. The tricky cases are the handling of (a) existential quantification when showing validity and (b) universal quantification when generating models. In order to handle case (a) some instantiation(s) of the quantified formulas can be created *hoping* to complete the proof. Soundness is preserved by any instantiation. The situation in case (b) which occurs when generating test data is, however, worse when using instantiation-based methods, because these methods are sound only if a complete instantiation of the quantified formula is guaranteed.

A popular instantiation heuristic is E-matching [21] which was first used in the theorem prover Simplify. E-matching is, however, not complete in general.

In general a quantified formula $\forall x.\varphi(x)$ cannot be substituted by a satisfiability preserving conjunction $\varphi(t_0) \wedge \ldots \wedge \varphi(t_n)$ where $t_0 \ldots t_n$ are terms computed via E-matching. For this reason Simplify may produce unsound answers (see also [17]) as shown in the following example.

$$\forall h.\forall i.\forall v.select(store(h,i,v),i) = v \tag{4}$$

$$\forall h.\forall j.0 \leqslant select(h,j) \wedge select(h,j) \leqslant 2^{32} - 1 \tag{5}$$

Formula (4) is an axiom of the theory of arrays and (5) specifies that all array elements of all arrays have values between 0 and $2^{32} - 1$. The first axiom is used to specify heap memory in [20]. Formula (5) seems like a useful axiom to specify that all values in the heap memory have lower and upper bounds, as it is the case in computer systems. However, the conjunction (4) $\wedge$ (5) is inconsistent, i.e. it is false, which can be seen when considering the following instantiation $[h := store(h_0, k, 2^{32}), j := k]$, (see [20]). Simplify, however, produces a counter example for $\neg((4) \wedge (5))$, which means that it satisfies the *false* formula (4) $\wedge$ (5). E-matching may be used for sound satisfiability solving when a complete instantiation of quantifiers is ensured. For instance, completeness of instantiation via E-matching has been shown for the Bernays-Schönfinkel class in [11]. An important fragment of FOL for program specification which allows a complete instantiation is the Array Property Fragment [3].

Quantifier elimination techniques, in the *traditional* sense, replace quantified formulas by *equivalent* ground formulas, i.e. without quantifiers. Popular methods are, e.g., the Fourier-Motzkin quantifier elimination procedure for linear rational arithmetic and Cooper's quantifier elimination procedure for Presburger arithmetic (see, e.g., [12] for more examples). These techniques are, in contrast to the proposed technique, not capable of eliminating the quantifier in, e.g., (1) or (2). Since first-order logic is only semi-decidable, equivalence preserving quantifier elimination is possible only in special cases. The transformation of formulas by our technique is not equivalence preserving. The advantage of our technique is, however, that it is not restricted to a certain class of formulas.

Finally, Finite Model Finding methods such as [25] regard the finite domain version of the satisfiability problem in first-order logic. Our approach handles, however, also infinite domains.

2 KeY's Dynamic Logic with Updates

The KeY system is based on the logic JAVA CARD DL, which is an instance of Dynamic Logic (DL) [15]. Dynamic Logic is an extension of first-order logic with modal operators. The ingredients of the KeY system that are needed in this paper are first-order logic (FOL) extended by the modal operators *updates* [23].

Notation. We use the following abbreviations for syntactic entities: $\boldsymbol{V}$ is the set of (logic) variables; $\boldsymbol{\Sigma^f}$ is the set of function symbols; $\boldsymbol{\Sigma^f_r} \subset \Sigma^f$ is the set of rigid function symbols, i.e. functions with a fixed interpretation such as, e.g., '0', 'succ', '+'; $\boldsymbol{\Sigma^f_{nr}} \subset \Sigma^f$ is the set of non-rigid function symbols, i.e. uninterpreted functions; $\boldsymbol{\Sigma^p}$ is the set of predicate symbols; $\boldsymbol{\Sigma}$ is the signature

consisting of $\Sigma^f \cup \Sigma^p$; $\boldsymbol{Trm_{FOL}}$ is the set of FOL terms; $\boldsymbol{Trm} \supset Trm_{FOL}$ is the set of DL terms; $\boldsymbol{Fml_{FOL}}$ is the set of FOL formulas; $\boldsymbol{Fml} \supset Fml_{FOL}$ is the set of DL formulas; $\boldsymbol{U}$ is the set of updates; $\doteq$ is the equality predicate; and $=$ is syntactic equivalence. The following abbreviations describe semantic sets: $\boldsymbol{\mathcal{D}}$ is the FOL domain or universe; $\boldsymbol{\mathcal{S}}$ is the set of states or equivalently the set of FOL interpretations. To describe semantic properties we use the following abbreviations: $\boldsymbol{val_s(t)} \in \mathcal{D}$ is the valuation of $t \in Trm$ and $\boldsymbol{val_s(u)} \in \mathcal{S}$ is the valuation of $u \in U$ in $s \in \mathcal{S}$; $\boldsymbol{s \vDash \varphi}$ means that φ is true in state $s \in \mathcal{S}$; $\boldsymbol{\vDash \varphi}$ means that φ is valid, i.e. for all $s \in \mathcal{S}$, $s \vDash \varphi$; and $\equiv$ is semantic equivalence.

Updates capture the essence of programs, namely the state change computed by a program execution. States and FOL interpretations are the same concept. An update changes the interpretation of symbols Σ^f_{nr} (such as uninterpreted functions). Hence, updates represent partial states and can be used to represent (partial) models of formulas. The set Σ^f_r represents rigid functions whose interpretation is fixed and cannot be changed by an update.

For instance, the formula $(\{a := b\}a = c) \in Fml$, where $a \in \Sigma^f_{nr}$ and $b, c \in \Sigma^f$ consists of the (function) update $a := b$ and the *application* of the update modal operator $\{a := b\}$ on the formula $a = c$. The meaning of this *update application* is the same as that of the weakest precondition $wp(a := b, a = c)$, i.e. it represents all states such that after the assignment $a := b$ the formula $a = c$ is true which is equivalent to $b = c$.

Definition 1. *Syntax. The sets U, Trm and Fml are inductively defined as the smallest sets satisfying the following conditions. Let $x \in V$; $u, u_1, u_2 \in U$; $f \in \Sigma^f_{nr}$; $t, t_1, t_2 \in Trm$; $\varphi \in Fml$.*

- *Updates. The set U of updates consists of: function updates $(f(t_1, \ldots, t_n) := t)$, where $f(t_1, \ldots, t_n)$ is called the* location term *and t is the* value term*; parallel updates $(u_1 \parallel u_2)$; conditional updates* (`if` φ; u)*; and quantified updates* (`for` x; φ; u)*.*
- *Terms. The set of Dynamic Logic terms includes all FOL terms, i.e. $Trm \supset Trm_{FOL}$; and $\{u\}t \in Trm$ for all $u \in U$ and $t \in Trm$.*
- *Formulas. The set of Dynamic Logic formulas includes all FOL formulas, i.e. $Fml \supset Fml_{FOL}$; $\{u\}\varphi \in Fml$ for all $u \in U$, $\varphi \in Fml$; and sequents $\Gamma \Longrightarrow \Delta$, where $\Gamma \subset Fml$ is called antecedent and $\Delta \subset Fml$ is called succedent.*

A sequent $\Gamma \Longrightarrow \Delta$ is equivalent to the formula $(\gamma_1 \wedge \ldots \wedge \gamma_n) \rightarrow (\delta_1 \vee \ldots \vee \delta_m)$, where $\gamma_1, \ldots, \gamma_n \in \Gamma$ and $\delta_1, \ldots, \delta_m \in \Delta$. Sequents are normally, e.g. in [2], not included in the set of formulas. However, in this work it is convenient to include them to the set of formulas as *syntactic sugar*.

Definition 2. *Semantics. We use the notation from Def. 1, further let $s, s' \in \mathcal{S}$; $v, v_1, v_2 \in \mathcal{D}$; $x, x_i, x_j \in V$; and $\varphi(x)$ and $u(x)$ denote a formula resp. an update with an occurrence of x.*

Terms and Formulas:

- *$val_s(\{u\}t) = val_{s'}(t)$, where $s' = val_s(u)$*
- *$val_s(\{u\}\varphi) = val_{s'}(\varphi)$, where $s' = val_s(u)$*

Updates:

- $val_s(f(t_1, \ldots, t_n) := t) = s'$, *where* $s' = s$ *except the interpretation of* f *is changed such that* $val_{s'}(f(t_1, \ldots, t_n)) = val_s(t)$
- $val_s(u_1; u_2) = s'$, *there is* s'' *with* $s'' = val_s(u_1)$ *and* $s' = val_{s''}(u_2)$
- $val_s(u_1 \,||\, u_2) = s'$. *We define* s' *by the interpretation of terms* t. *Let* $v_0 = val_s(t)$, $v_1 = val_s(\{u_1\}t)$, *and* $v_2 = val_s(\{u_2\}t)$.

$$\textit{If } v_0 \neq v_2 \textit{ then } val_{s'}(t) = v_2 \textit{ else } val_{s'}(t) = v_1.$$

- $val_s(\texttt{if}\ \ \varphi;\, u) = s'$, *if* $val_s(\varphi) = true$ *then* $s' = val_s(u)$, *otherwise* $s' = s$.
- *Intuitively, a quantified update* $(\texttt{for}\ x;\, \varphi(x);\, u(x))$ *is equivalent to the infinite composition of parallel updates (parallel updates are associative):*

$$\ldots \,||\, (\texttt{if}\ \ \varphi(x_i);\, u(x_i)) \,||\, (\texttt{if}\ \ \varphi(x_j);\, u(x_j)) \,||\, \ldots$$

satisfying some global order $\succ$ *such that* $\beta(x_i) \succ \beta(x_j)$, *where* $\beta : V \to \mathcal{D}$.

A complete and formal definition of quantified updates cannot be given in the scope of this paper; we refer the reader to [23,2] for a complete definition of the language and the simplification calculus. In the following some examples are shown of how updates, terms, and formulas are evaluated in KeY respecting the given semantics in Def 2.

- $\{f(1) := a\}f(2) = f(1)$ simplifies to $f(2) = a$.
- $\{f(b) := a\}P(f(c))$ simplifies to $(b \doteq c \to P(a)) \wedge (\neg b \doteq c \to P(f(c)))$.
- $\{f(a) := a\}f(f(f(a)))$ simplifies to a.
- $\{u_1; f(t_1, \ldots, t_n) := t\}$ is equivalent to $\{u_1 \,||\, f(\{u\}t_1, \ldots, \{u\}t_n) := \{u\}t\}$.
- $\{f(1) := a \,||\, f(2) := b\}f(2) = f(1)$ simplifies to $b = a$.
- $\{f(1) := a \,||\, f(1) := b\}f(2) = f(1)$ simplifies to $f(2) = b$, i.e. the last update *wins* in case of a conflict.
- $\{\texttt{if}\ \ \varphi;\, f(b) := a\}P(f(c))$ simplifies to $\varphi \to \{f(b) := a\}P(f(c))$.
- $\{\texttt{for}\ x;\, 0 \leqslant x \wedge x \leqslant 1;\, f(x) := x\}$ is equivalent to $\{f(1) := 1 \,||\, f(0) := 0\}$.

3 Test Data Generation for Quantified Formulas

The basic idea of our approach is to generate a partial FOL model in which a quantified formula that we want to eliminate evaluates to true. A set of quantified formulas can be eliminated, i.e. evaluated to true, by successive extensions of the partial model. This approach can be continued also on ground formulas to generate complete models. While this basic idea is simple, the interesting questions are: how to represent the interpretations, how to generate (partial) models, and what calculus is suitable in order to evaluate formulas under those (partial) interpretations.

A suitable language for this purpose with a simplification calculus are updates. In order to generate test data that satisfies a test data constraint $\varphi \in Fml_{FOL}$, our approach is to generate an update u, such that $\{u\}\varphi$ evaluates to true. If such an update exists, then φ is satisfiable and the update represents a test preamble initializing a state in which φ evaluates to true (see Section 3.3).

Example 1. Referring to Figure 1, models for satisfying formulas (2) and (1) can be represented by the following update applications, respectively.

$$\{\texttt{for}\ o : \texttt{C};\ \neg o \doteq \texttt{null};\ \texttt{l}(\texttt{s}(o)) := 10\}(2) \tag{6}$$

$$\{\texttt{for}\ o : \texttt{C};\ \neg o \doteq null;\ (\texttt{for}\ i : \texttt{int};\ 0 \leqslant i \leqslant \texttt{l}(\texttt{s}(o));\ \texttt{acc}_{[]}(\texttt{s}(o), i) := obj_C(i))\}(1)$$

where `l` is an abbreviations of `length`, $\texttt{acc}_{[]}(\texttt{s}(o), i)$ encodes the array access `o.s[i]`, and $obj_C : \texttt{int} \to \texttt{C}$ is an injective function from numbers to objects.

Our technique uses heuristics for construction of candidate updates u and then verifies $\{u\}\varphi$. The advantage of using updates is the availability of the powerful update simplification calculus to automatically verify $\{u\}\varphi$. In particular it is not required to generate loop invariants in order to handle quantified updates. In this way different heuristics can be used in a search procedure for u while guaranteeing correctness of the test preamble in the end. The technique requires a theorem prover for FOL and an implementation of updates.

Definition 3. ***Procedure Th.*** *The procedure* `Th` *represents a theorem prover.*

- *Given a formula $\vartheta \in Fml$ as input,* $\texttt{Th}(\vartheta)$ *returns a set $\Theta \subset Fml_{FOL}$.*
- *If $\Theta = \emptyset$, then $\vDash \vartheta$.*
- *Each $\vartheta' \in \Theta$ is either a literal or a quantified formula.*
- *The prover may use only local equivalence rules. Global rules ensure that satisfiability of $\neg\vartheta'$ for any $\vartheta' \in \Theta$ ensures satisfiability of $\neg\vartheta$. Local equivalence rules ensure additionally that $\vDash (\neg\vartheta') \to (\neg\vartheta)$.*

The KeY system uses a sequent calculus. Therefore, in the following sections we assume that the set Θ returned by `Th` consists of sequents (see Def. 1). In the following two algorithms are described. Algorithm 1 in Section 3.1 extracts information from (quantified) formulas for update construction and invokes a theorem prover to verify $\{u\}\varphi$. Algorithm 1 queries Algorithm 2 to construct candidate updates based on information obtained from Algorithm 1. Algorithm 2 is described in Section 3.2. In order to keep the pseudo-code small we use indeterministic choice points, marked by the keyword **choose**, and assume a backtracking control-flow wrt. to these choice points. In this way we also separate the basic algorithm from concrete search heuristics. If a choice at a choice point cannot be made, e.g. when trying to select an element of an empty set, then the algorithms backtracks or stops with the result "unknown" resp. "$\emptyset$".

3.1 The Model Search Algorithm

Assume we want to generate test data resp. a model satisfying the input formula ϕ_{in}. The Algorithm 1 reformulates this problem as counter example generation for φ' (Line 1). In Line 4 the algorithm attempts to show $\vDash \varphi'$ (Line 4). If φ' is valid, then $\Phi = \emptyset$ and the algorithm stops (Line 5) because φ' has no counter example and ϕ_{in} is unsatisfiable. The other case is that the proof attempt of φ' results in a set of open, i.e. unproved, proof obligations Φ (Line 5). In this case it is unknown if a model of ϕ_{in} exists or not. The proof obligations Φ result from case distinctions

Algorithm 1. modelSearch(ϕ_{in})

1: $\varphi' := \neg\phi_{in}$
2: solution := $\bot$
3: **loop**
4: $\quad\Phi := \mathtt{Th}(\varphi')$
5: $\quad$**choose** $\varphi \in \Phi$
6: $\quad$**if** φ is ground **then**
7: $\quad\quad$**if** GROUNDPROC($\neg\varphi$) = ("sat", groundmodel) **then**
8: $\quad\quad\quad$**return** ("sat", groundmodel, solution)
9: $\quad\quad$**else**
10: $\quad\quad\quad$**backtrack** or **return** ("unknown", $\bot$, $\bot$)
11: $\quad\quad$**end if**
12: $\quad$**end if**
13: $\quad$normalize φ such that quantified formulas appear only in the antecedent of φ
14: $\quad$**choose** a quantified formula $\forall x.\phi(x)$ in φ, i.e. let $\varphi = (\Gamma, \underline{\forall x.\phi(x)} \Longrightarrow \Delta)$
15: $\quad\varphi' := (\Gamma, \underline{true} \Longrightarrow \Delta)$
16: $\quad\psi := (\Gamma \Longrightarrow \underline{\forall x.\phi(x)}, \Delta)$
17: $\quad\Psi := \mathtt{Th}(\psi)$
18: $\quad$**while** $\Psi \neq \emptyset$ **do**
19: $\quad\quad$**choose** $\psi' \in \Psi$
20: $\quad\quad\Upsilon :=$ formulaToUpdate(ψ')
21: $\quad\quad$**choose** $(u, \alpha) \in \Upsilon$
22: $\quad\quad$solution := append (u, α) to solution
23: $\quad\quad\varphi' := (\alpha \rightarrow \{u\}\varphi')$
24: $\quad\quad\psi := (\alpha \rightarrow \{u\}\psi)$
25: $\quad\quad\Psi := \mathtt{Th}(\psi)$
26: $\quad$**end while**
27: **end loop**

in the proof structure created by Th and contain valuable information, because they describe situations in which φ' potentially has counter examples.

In Line 5 the algorithm selects a formula $\varphi \in \Phi$. The goal is to create a counter example for φ, i.e. satisfy $\neg\varphi$, in order to satisfy ϕ_{in}. Ground formulas should be preferred at this choice point because they can be efficiently checked by a ground procedure such as an SMT solver. Otherwise, we assume φ is not ground. After normalization at Line 13 the antecedent of φ contains at least one universally quantified formula and all formulas of the succedent are ground. This normalization can be easily achieved by the equivalence $(\Gamma \Longrightarrow \exists x.\phi(x), \Delta) \equiv (\Gamma, \forall x \neg\phi(x) \Longrightarrow \Delta)$. A counter example for φ must satisfy the formulas in the antecedent, i.e. Γ and $\forall x.\phi(x)$. The algorithm selects a quantified formula $\forall x.\phi(x)$ from the antecedent of φ (Line 14) for which a model is generated in the following.

The core idea of this algorithm is to create an update u, such that $\{u\}\forall x.\phi(x)$ evaluates to true, and in this way to eliminate the quantified formula. The weakest condition under which $\forall x.\phi(x)$ evaluates to true in φ can be expressed as

$$\underbrace{(\Gamma, \forall x.\phi(x) \Longrightarrow \Delta)}_{\varphi} \leftrightarrow \underbrace{(\Gamma, true \Longrightarrow \Delta)}_{\varphi'} \tag{7}$$

which simplifies by equivalence transformations to

$$\underbrace{\Gamma \Longrightarrow \forall x.\phi(x), \Delta}_{\psi} \tag{8}$$

Any model of (8) is also a model of (7), which means that in such states $\forall x.\phi(x)$ evaluates to true. Hence, in Line 15 the formula φ' is constructed where the quantified formula is replaced by true. Substituting φ by φ' in subsequent computation is sound only if (7) or equivalently (8) is valid. Therefore formula (8) is assigned to ψ in Line 16 and is checked by `Th` in Line 17. If ψ can be proved, then $\Psi = \emptyset$ and the algorithm continues in Line 4 where φ' (now without the quantified formula) is used instead of φ. Otherwise, if the proof of (8) of does not close (Line 17), then the result is a set of proof obligations Ψ.

The formulas Ψ (Line 19) describe potential states in which $\forall x.\phi(x)$ does not evaluate to true. The goal is therefore to construct an update u (Line 20) such that for each formula $\psi' \in \Psi$, $\vDash \{u\}\psi'$. If this is the case, then also $\vDash \{u\}\psi$ which allows us to eliminate the quantified formula by the equivalence (7). Instead of satisfying this condition in one step, our heuristic is rather to extend u iteratively. In each iteration of the inner loop one formula $\psi' \in \Psi$ is selected in Line 19 and Ψ is updated in Line 25 until Ψ *eventually* decreases to $\emptyset$.

Remark. For the construction of the updates it is sometimes necessary to introduce and axiomatize fresh function symbols. For instance, it may be required to introduce a fresh function $notZero \in \Sigma^f$ with the axiom $\neg(notZero \doteq 0)$. An update is therefore associated with an axiom α (see Line 21).

The goal of the inner loop is to generate an update u and a formula α (Line 20) and to check if

$$\alpha \rightarrow \{u\}\psi \tag{9}$$

evaluates to true. Formula 9 (see Line 24) is a weakening of (8). The procedure `formulaToUpdate` which is described in Section 3.2 generates candidate pairs (u, a) that are likely to satisfy (9).[1] In (8), respectively (9), the quantified formula occurs negated wrt. (7). An important consequence of this negation is that in Lines 17 and 25 the theorem prover can skolemize the quantified formula (8) resulting in

$$\Gamma \Longrightarrow \phi(sk), \Delta \tag{10}$$

where $sk \in \Sigma_r^f$ is a fresh symbol. In this way the formula $\phi(sk)$ can be simplified by the calculus and information contained in the structure of $\phi(sk)$ is extracted to the sequent level, i.e. the boolean structure of $\phi(sk)$ is flattened (see Def. 3). This information occurs in the formulas $\psi' \in \Psi$ (Line 19). The task of generating a pair (u, α) from ψ' for satisfying (9) by the procedure `formulaToUpdate` is considerably simpler than generation of the pair from the whole unsimplified quantified formula $\forall x.\phi(x)$.

[1] Note that since the procedure `formulaToUpdate` uses only one formula $\psi' \in \Psi$ to construct the pair (u, α), formula 9 may not evaluate true. In this case the inner loop continues iteration and unsolved formulas $\psi' \in \Psi$ reapear in the next iteration to be solved.

Example 2. Let $\varphi = (A, \forall x.\phi(x) \Longrightarrow B)$, where $A, B \in Fml_{FOL}$ and $\phi(x) = (f(x) > x \wedge g(x) < f(x))$. Generating a model for φ in one step is complicated because the quantified formula cannot be skolemized. In contrast, in $\psi = (A \Longrightarrow \forall x.\phi(x), B)$ the quantified formula is negated (because $(F \Longrightarrow) \equiv (\Longrightarrow \neg F)$) and can therefore be skolemized. $\texttt{Th}(\psi)$ yields $\Psi = \{(A \Longrightarrow f(sk) > sk, B), (A \Longrightarrow g(sk) < f(sk), B)\}$. The structure of each $\psi' \in \Psi$ is simpler than of ψ. The procedure `formulaToUpdate` can then generate, e.g., the following updates with axioms to satisfy the formulas in Ψ respectively:

$$\{((\texttt{for}\ x;\ true;\ f(x) := x - 1), true), ((\texttt{for}\ x;\ true;\ g(x) := f(x) + 1), true)\}.$$

Checking formula (9) as described above is imporant because it is equivalent to

$$(\alpha \rightarrow \{u\}\varphi) \leftrightarrow (\alpha \rightarrow \{u\}\varphi') \tag{11}$$

which in turn is a weakening of (7). Accordingly, in Line 23 the formula φ' is updated. If (9) is valid, which is checked in Line 25, then the inner loop terminates and the outer loop continues execution. Hence, the original counter example generation problem for φ is replaced by the counter example generation problem for $\alpha \rightarrow \{u\}\varphi'$ where the quantified formula is eliminated, i.e. replaced by true. This is sound because if (9) is valid, then (11) is valid and therefore a counter example for $\alpha \rightarrow \{u\}\varphi'$ is a counter example for $\alpha \rightarrow \{u\}\varphi$. The latter implies that φ has a counter example as well which is formalized by the following proposition.

Proposition 1. *Let $u \in U$, $\alpha, \varphi' \in Fml$. If there is an $s \in \mathcal{S}$ such that $s \vDash \neg(\alpha \rightarrow \{u\}\varphi)$, then there is $s' \in \mathcal{S}$ such that $s' \vDash \neg\varphi$.*

Proof. Assume there is $s \in \mathcal{S}$ such that $s \vDash \neg(\alpha \rightarrow \{u\}\varphi)$, which implies that $s \vDash \alpha$ and $s \vDash \neg\{u\}\varphi$. The following is an equivalence in Dynamic Logic: $\neg\{u\}\phi \equiv \{u\}\neg\phi$ (see [2]). Using this equivalence we obtain the statement: there is $s \in \mathcal{S}$ such that $s \vDash \{u\}(\neg\varphi)$. According to Def. 2, there is $s' \in \mathcal{S}$ such that $s' = val_s(u)$, and $s' \vDash \neg\varphi$. ∎

3.2 Update Generation For Satisfying Quantified Formulas

In this section we describe Algorithm 2 which is used by Algorithm 1 to construct updates for satisfying quantified formulas. According to Section 3.1 this algorithm receives as input a sequent ψ' that is an open proof obligation of $\texttt{Th}\ (\psi)$. Algorithm 2 is queried for each open proof obligation and is expected to generate an update and axiom pair (u, α) that is *likely* to satisfy

$$\alpha \rightarrow \{u\}\psi'$$

Considering Example 2, if $\psi' = (A \Longrightarrow f(sk) > sk, B)$, then a suitable (u, α) pair is, e.g., $((\texttt{for}\ x;\ true;\ f(x) := x - 1), true)$.

As each pair (u, α) satisfies one of the open proof obligations $\psi' \in \texttt{Th}(\psi)$, a series of such pairs *eventually* satisfies formula (9). The algorithm returns a set of

Algorithm 2. formulaToUpdate(ψ')

1: let sk = the skolem function of ψ'
2: let $(\Gamma \Longrightarrow \Delta) = \psi'$
3: let $(\Gamma_{sk} \Longrightarrow \Delta_{sk}) \subset (\Gamma \Longrightarrow \Delta)$ (according to the description)
4: **choose** $\vartheta_{sk} \in (\neg\Gamma_{sk} \cup \Delta_{sk})$ (negation is applied to each element in Γ)
5: **choose** $\vartheta'_{sk} \in$ `solve`(ϑ_{sk})
6: **choose** $(u, \alpha) \in$ `concretize`(ϑ'_{sk})
7: **choose** $(u', \alpha) \in$ `toQuanUpd`$(sk, (u, \alpha), (\Gamma_{sk} \Longrightarrow \Delta_{sk}), \vartheta_{sk})$
8: **return** (u', α)

alternative solutions for each sequent ψ'. Important to note is that soundness of the approach is guaranteed by any pair (u, α) because the inner loop of Algorithm 1 does not terminate until a model is generated for ψ.

According to Def. 3 the sequent ψ' has been simplified such that all formulas on the sequent level are either quantified formulas or atoms. We are interested in atoms that were derived from $\forall x.\phi(x)$. Therefore our heuristic is to categorize those atoms in ψ' as highly relevant for the construction of the update u that have an occurrence of the skolem symbol sk, that was introduced in (10). Let ψ'_{sk} be defined as

$$\Gamma_{sk} \Longrightarrow \Delta_{sk}$$

such that it coincides with ψ', except that all quantified formulas and formulas that do not contain an occurrence of sk are removed in ψ'_{sk} (Line 3 of Algorithm 2). Hence, all formulas in Γ_{sk} and Δ_{sk} are ground formulas with an occurrence of sk. Following the Example 2, $(\Gamma_{sk} \Longrightarrow \Delta_{sk})$ is either $(\Longrightarrow f(sk) > sk)$ or $(\Longrightarrow g(sk) < f(sk))$.

The goal is to create an update u such that $\vDash (\{u\}\psi'_{sk})$. Note that $(\{u\}\psi'_{sk}) \rightarrow (\{u\}\psi')$. In order to evaluate $\{u\}\psi'_{sk}$ to true the update u must either evaluate an atom in Γ_{sk} to $false$, or an atom in Δ_{sk} to $true$. We refer to the chosen atom, whose interpretation we want to manipulate, as the *core atom*. Let ϑ_{sk} denote the chosen core atom (Line 4). The task is to construct a function update u such that $\{u\}\vartheta_{sk}$ evaluates $true$. This task is divided into two steps realized by the algorithms `solve` (Line 5) and `concretize` (Line 6).

Definition 4. ***Procedure Solve.*** *Given an atom ϑ_{sk}, whose top-level symbol is a relation $R \in \Sigma^p$, the procedure* `solve` *constructs a set of atoms such that for each atom $\vartheta'_{sk} \in \Theta$ holds*

- $\vartheta'_{sk} = R'(\underline{f(t_1, \ldots, t_n)}, v)$, *i.e. syntactic equivalence*
- $\vartheta'_{sk} \doteq \vartheta_{sk}$, *i.e. semantic equivalence*

where $R' \in \Sigma^p_r$, $f \in \Sigma^f_{nr}$, $f \neq sk$, and $t_1, \ldots, t_n, v \in Trm$.

The procedure `solve` creates normal forms for core atoms. For example, for the formula $\vartheta_{sk} = (f(sk) + 3 < g(sk) - sk)$, with "$<$"$\in \Sigma^p$, $f, g \in \Sigma^f_{nr}$, the procedure `solve` should generate, e.g., the following set:

$$\{\underline{f(sk)} < (g(sk) - sk - 3), \underline{g(sk)} > (f(sk) + 3 + sk)\} \tag{12}$$

Note that the procedure `solve` is part of our heuristic and the resulting set is not strictly defined, it may also be empty. Some core atoms may be not solvable by the procedure but the more results the procedure `solve` produces the better is the chance of generating a suitable update.

Definition 5. ***Procedure Concretize.*** *Let $R \in \Sigma^p_r$, $f \in \Sigma^f_{nr}$, and $t_1, .., t_n, v \in Trm$. Given an atom of the form $R(f(t_1, \ldots, t_n), v)$ the procedure* `concretize` *creates a set of pairs (u, α), with $u \in U$, $\alpha \in Fml$, such that:*

- $u = (f(t_1, \ldots, t_n) := value)$, *where* $value \in Trm$
- $\alpha \rightarrow \{u\}R(f(t_1, \ldots, t_n), v)$ *evaluates to true*

The procedure `concretize` creates for a given normalized core atom ϑ'_{sk} an update u that evaluates $\{u\}\vartheta'_{sk}$ to *true*. E.g., if the normalized core atom is of the form $t_1 = t_2$, using infix notation, then the result of the procedure `concretize` is simply $((t_1 := t_2), true)$. In some cases it is desired to introduce fresh symbols and to axiomatize them for the construction of the term *value*. Such axiomatizations are collected in the formula α.

For example, using the normalized core atom $\vartheta'_{sk} = (f(sk) < (g(sk) - sk - 3))$ from the solution set of the previous example, the procudure `concretize` may produce, e.g., the following solution:

$$\{(\underbrace{f(sk) := (g(sk) - sk - 3) + sk_2)}_{u}, \underbrace{sk_2 < 0}_{\alpha}\} \tag{13}$$

where $sk_2 \in \Sigma^f_r$ is a fresh constant. Using this solution, we can evaluate $\alpha \rightarrow \{u\}\vartheta'_{sk}$ as follows

$$\begin{aligned} &sk_2 < 0 \rightarrow \{f(sk) := (g(sk) - sk - 3) + sk_2\}(f(sk) + 3 < g(sk) - sk) \\ &sk_2 < 0 \rightarrow (g(sk) - sk - 3) + sk_2 + 3 < g(sk) - sk \\ &sk_2 < 0 \rightarrow sk_2 < 0 \end{aligned}$$

The next step uses the result computed by the procedures `solve` and `concretize` in order to create a quantified update (Line 7).

Definition 6. ***Procedure toQuanUpd.*** *Given a tuple (u, α), with $u \in U$ and $\alpha \in Fml$, a sequent $\Gamma_{sk} \Longrightarrow \Delta_{sk}$, a core atom ϑ'_{sk}, and a (skolem) function sk. The procedure creates the pair (u', α) where $u' \in U$ has the form (let $z \in V$)*

$$\texttt{for } z;\ \neg((\Gamma_{sk} \backslash \{\vartheta_{sk}\}) \Longrightarrow (\Delta_{sk} \backslash \{\vartheta_{sk}\}))[z \backslash sk];\ u[z \backslash sk]$$

The substitution $[x \backslash sk]$ deskolemizes all formulas and terms in order to quantify functions and predicates at those argument positions as they were quantified in the original quantified formula (see (8) vs. (10)). The guard $\neg((\Gamma_{sk} \backslash \{\psi_{sk}\}) \Longrightarrow (\Delta_{sk} \backslash \{\psi_{sk}\}))$ restricts the application of the update in order create small models as explained in the following.

For example, assume we want to construct an update that evaluates the formula $\forall x.(x > 4 \rightarrow (f(x)+3 < g(x)-x))$ to true. Algorithm 1 invokes Algorithm 2 with the following sequent ψ':

$$sk > 4 \Longrightarrow f(sk) + 3 < g(sk) - sk$$

Let $f(sk)+3 < g(sk)-sk$ be the core atom ϑ_{sk} chosen in Line 4, then according to the previous examples procedures `solve` and `concretize` produce the intermediate result (13) that serves as input to the procedure `toQuanUpd`. We obtain the guard

$$\neg((\{sk > 4\}\backslash\{\vartheta_{sk}\}) \Longrightarrow (\{\psi_{sk}\}\backslash\{\psi_{sk}\}))[z\backslash sk]$$

which simplifies to $\neg(z > 4 \Longrightarrow)$ and then to $z > 4$. The final result of the procedure `toQuanUpd` for this example is the (u, α)-pair:

$$((\texttt{for}\ z;\ z > 4;\ f(z) := (g(z) - z - 3) + sk_2), sk_2 < 0)$$

3.3 From Updates to a Test Preamble

A test preamble is part of a test harness and its goal is to initialize the program under test with a desired program state. Here we assume that the test preamble can directly write values to all relevant memory locations. For this purpose, e.g., the KeY tool uses Java's reflection API.

Assume the input formula for Algorithm 1 is a test data constraint ϕ_{in}. If the algorithm terminates with the answer "sat", then it also provides a ground model M, i.e. assignment of values to non-quantified terms, and a sequence S of update and axiom pairs $(u_m, a_m), \ldots, (u_0, a_0)$. By the construction of the algorithm the axioms are already satisfied by M. The conversion of M into a test preamble is a well-known technique and is not discussed here.

The choice of using updates to represent models of quantified formulas is also very convenient for test preamble generation. The reason is that updates can be viewed as a small imperative programming language with some special constructs. An algorithm that converts updates to a test preamble simply has to follow the semantics of updates (Def. 2). Conversion of function updates to assignments and conditional updates to `if`-statements is trivial. Parallel updates were not created by our algorithm, they were used only to define the semantics of quantified updates. Quantified updates are converted into loops, e.g. (6) is converted to (3). If a quantified update quantifies over integers, then the integer bounds have to be determined. If the update quantifies over objects, then our solution is iterate over all objects that were created by the preamble. This solution is, however, only an approximation as it does not initialize objects that are created later on during the execution of the program under test.

4 Evaluation

We have implemented our algorithm, i.e. the combination of Algorithms 1 and 2, as well as a converter from updates to a test preamble, in an experimental

Classes with invariants	T	A	B
Account	4	4	4
AccountMan_src	5	5	2
Currency_src	2	2	2
SavingRule	4	2	2
SpendingRule	4	2	2
Transfers_src	3	3	2
Total	22	18	14

Methods with Specifications	T	A	B
AccountMan_src::IsValid()	6	5	2
AccountMan_src::BOdelete()	6	5	2
AccountMan_src::isValidBank()	5	4	2
AccountMan_src::isValAcc()	5	5	2
AccountMan_src::getRef()	5	3	2
Total	27	22	10

Fig. 2. Evaluation of the model generation algorithm applied to a banking software with JML specifications; T: total number of quantified formulas in one conjunction that occurred as test data constraints; A, B: maximum number of quantified formulas solved in a conjunction; A: our model generation algorithm; B: SMT solvers

version of the KeY tool. The technique is currently realized as an interactive model generator. The implementation proposes candidate updates to be selected by the user. The reason for this is two-fold. On the one hand, the interaction with the algorithm enables us to study heuristics for the model generation as well as to identify and understand limitations of the algorithm. On the other hand, it is the paradigm of the KeY tool to combine automation and interaction. In more general test generation contexts a full automation of the model generator is of course expected, and possible.

In order to test the algorithm we have used several examples from different sources. In the beginning, we have used hand-crafted formulas in order to test and develop the algorithm. A crucial improvement was achieved by the generation of formula (8) that allows skolemization of the quantified formulas as described in Section 3.1. Earlier approaches to generate models without formula (8) were not successful.

To test our algorithm on more realistic tests, we used a small banking software that was the subject in a case study on JML-based software validation [8]. The banking software contains JML specifications with quantified formulas. When applying KeY's test generation techniques [10,13,14], the quantified formulas are encountered as test data constraints. Our goal was to test for how many of these formulas our algorithm can generate models. Figure 2 shows the results.

The left table of Figure 2 shows numbers of quantified formulas that stem from class invariants of the respective classes and the right table shows numbers of quantified formulas that stem from method preconditions and loop invariants. Note that additional quantified formulas are generated by the KeY tool as shown in the motivating example in Figure 1. The column T shows the total number of quantified formulas that occurred in test data constraints in a *conjunction*, i.e. a complete model must satisfy the whole conjunction. Columns A and B show the maximum number of quantified formulas that we found solvable in one conjunction which required us to test different combinations of the quantified formulas. Column A shows the results of our algorithm and column B shows respectively the best result achieved by any of the SMT solvers Z3, CVC3, and Yices. The evaluation shows that our algorithm can solve quantified formulas

that state-of-the-art SMT solvers cannot solve. Furthermore, our algorithm was able to generate models for almost all of the quantified formulas when it was applied to the quantified formulas in separation, i.e. not in a conjunction. This simplification did not make an improvement on the SMT side.

5 Conclusions and Future Work

In formal test generation techniques quantified formulas occur in test data constraints. We have therefore developed a model generation algorithm for quantified formulas. The algorithm is not guaranteed to find a solution but on the other hand it is not restricted to a particular class of formulas. In this way, the algorithm can solve formulas that are otherwise not solvable by satisfiability modulo theories (SMT) solvers alone which is confirmed by our experiments. The algorithms can be used as a precomputation step for SMT solvers. In this case the algorithm generates only a partial model that satisfies only the quantified formulas and returns a residue of ground formulas to be solved.

Models generated by the algorithm are represented as updates. Quantified updates are suitable to represent models of quantified formulas. Our algorithm systematically analyses quantified formulas and uses heuristics to generate candidate updates. The KeY system implements a powerful update simplification calculus that allows us to check if a quantified formulas evaluates to true in a model represented by an update. Furthermore, updates are a convenient model representation language for test generation, because they have program semantics and can be converted into a test preamble.

The current implementation of the algorithm queries the user to select candidate updates from a list. We are developing heuristics in order to automate the search. The kind of formulas that is solvable by our general approach depends on the model representation language. Quantified updates are, e.g., not expressive enough to represent models of recursive functions. Our future research plans are therefore to extend the expressiveness of updates and to extend the simplification calculus.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005)
4. Csallner, C., Smaragdakis, Y.: Check 'n' Crash: combining static checking and testing. In: ICSE, pp. 422–431. ACM, New York (2005)
5. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)

6. Deng, X., Robby, Hatcliff, J.: Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In: TAICPART-MUTATION 2007: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, Washington, DC, USA, pp. 3–12. IEEE Computer Society, Los Alamitos (2007)
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
8. du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., Lanet, J.-L.: Case study in JML-based software validation. In: Proceedings Automated Software Engineering, pp. 294–297. IEEE Computer Society, Los Alamitos (2004)
9. Dutertre, B., de Moura, L.: The YICES SMT solver. Technical report, Computer Science Laboratory, SRI International (2006), http://yices.csl.sri.com/tool-paper.pdf (visited July 2010)
10. Engel, C., Gladisch, C., Klebanov, V., Rümmer, P.: Integrating verification and testing of object-oriented software. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 182–191. Springer, Heidelberg (2008)
11. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
12. Ghilardi, S.: Quantifier elimination and provers integration. Electr. Notes Theor. Comput. Sci. 86(1) (2003)
13. Gladisch, C.: Verification-based test case generation for full feasible branch coverage. In: SEFM, pp. 159–168 (2008)
14. Gladisch, C.: Could we have chosen a better loop invariant or method contract? In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 74–89. Springer, Heidelberg (2009)
15. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. The MIT Press, London (2000)
16. KeY project homepage, http://www.key-project.org/
17. Kiniry, J.R., Morkan, A.E., Denby, B.: Soundness and completeness warnings in ESC/Java2. In: Proc. Fifth Int. Workshop Specification and Verification of Component-Based Systems, pp. 19–24 (2006)
18. Leavens, G., Cheon, Y.: Design by contract with JML (2006), http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf (visited May 2010)
19. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. 14(2), 105–156 (2004)
20. Moskal, M.: Satisfiability Modulo Software. PhD thesis, University of Wrocław (2009)
21. Moskal, M., Lopuszanski, J., Kiniry, J.R.: E-matching for fun and profit. Electr. Notes Theor. Comput. Sci. 198(2), 19–35 (2008)
22. Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Challenges in satisfiability modulo theories. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 2–18. Springer, Heidelberg (2007)
23. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 422–436. Springer, Heidelberg (2006)
24. Visser, W., Păsăreanu, C., Khurshid, S.: Test input generation with Java PathFinder. In: ISSTA, pp. 97–107. ACM, New York (2004)
25. Zhang, J., Zhang, H.: Extending finite model searching with congruence closure computation. In: Buchberger, B., Campbell, J. (eds.) AISC 2004. LNCS (LNAI), vol. 3249, pp. 94–102. Springer, Heidelberg (2004)

Efficient Distributed Test Architectures for Large-Scale Systems*

Eduardo Cunha de Almeida[1], João Eugenio Marynowski[1], Gerson Sunyé[2], Yves Le Traon[3], and Patrick Valduriez[4]

[1] Universidade Federal do Paraná, Brazil
{eduardo,jeugenio}@inf.ufpr.br
[2] INRIA - University of Nantes, France
gerson.sunye@univ-nantes.fr
[3] SnT, University of Luxembourg
yves.letraon@uni.lu
[4] INRIA - LIRM, Montpellier, France
patrick.valduriez@inria.fr

Abstract. Typical testing architectures for distributed software rely on a centralized test controller, which decomposes test cases in steps and deploy them across distributed testers. The controller also guarantees the correct execution of test steps through synchronization messages. These architectures are not scalable while testing large-scale distributed systems due to the cost of synchronization management, which may increase the cost of a test and even prevent its execution. This paper presents a distributed architecture to synchronize the test execution sequence. This approach organizes the testers in a tree, where messages are exchanged among parents and children. The experimental evaluation shows that the synchronization management overhead can be reduced by several orders of magnitude. We conclude that testing architectures should scale up along with the distributed system under test.

1 Introduction

There are increasing needs of dynamic virtual organizations such as professional communities where members contribute with their own data sources and computation resources, perhaps small ones but in high numbers, and may join and leave the organization at will. In particular, current solutions require heavy organization, administration and tuning which are not appropriate for large numbers of small devices. While current Grid solutions focus on data sharing and collaboration for statically defined virtual organizations with powerful servers, Peer-to-Peer (P2P) techniques focus on scalability, dynamism, autonomy and decentralized control. Both approaches can be combined and the synergy between P2P computing and Grid computing has been advocated to help resolve their respective deficiencies [1]. Grid and P2P systems are thus becoming key

* Work partially funded by the Datluge CNPq-INRIA project.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 174–187, 2010.

technologies for software development, but still lack an integrated solution to validate the final software.

Although Grid and P2P systems usually have a simple public interface, the interaction between nodes is rather complex and difficult to test. For instance, distributed hash tables (DHTs) [2,3], provide only three public operations (insert, retrieve and lookup), but need very complex interactions to ensure the persistence of data while nodes leave or join the system. Testing these three operations is rather simple. However, testing that a node correctly transfers its data to another node before leaving requires the system to be in a particular state. Setting a system into a given state requires the execution of a sequence of steps, corresponding to the public operation calls as well as the requests to join or leave the system, in a precise order. The same rationale can be applied to data grid management systems (DGMS) [4].

Testing a large scale distributed application implies dealing with distributed test scenarios: this means synchronizing all the tasks required to execute the test scenarios and collect/aggregate test verdicts.

Typical testing architectures for distributed software rely on a centralized test controller, which decomposes test cases in steps and deploy them across distributed testers. The controller also guarantees the correct execution of test steps through synchronization messages. These architectures are not scalable while testing large-scale distributed systems due to the cost of synchronization management, which may increase the cost of a test and even prevent its execution.

From the point of view of performance, this distribution of testing tasks across the distributed architecture is intrusive and may impact the behavior of the system under test itself. This phenomenon is already known in the testing community and corresponds to the case when the observer perturbs the experience, i.e., the test environments modifies the system's behavior. A second issue occurs with error handling and diagnosis: if the same error occurs on several nodes, only one log is required while the centralized testing architecture will generate redundant logs.

The issue addressed in this paper is the optimization of the test architecture for large-scale distributed dynamic systems: the goal is to improve the performance and reduce the impact of test tasks.

This paper studies three alternatives to build a testing architecture and synchronize the test execution sequences: a centralized solution, and two distributed architectures. The distributed approach organizes the testers in a tree, where messages are exchanged among parents and children. Two strategies for building the tree are compared, depending on whether we consider the physical nodes for aggregating testers or not. The experimental evaluation shows that the synchronization management overhead can be reduced by several orders of magnitude. For the tree-based approaches, empirical results reveal the tradeoff between load balancing (deploying too many logical nodes on the same physical node degrades its performance) and thus synchronization tasks simplification. We conclude that the distributed testing architectures scale up along with the distributed system under test, but that tuning the ideal load balancing is critical.

This paper is organized as follows. Section 2 discusses related work. In Section 3, we introduce some basic concepts in software testing and the requirements for a large-scale testing architecture. In Section 4, we discuss the centralized approach in detail, and present two distributed approaches and their trade-offs. In Section 5, we present initial results through implementation and experimentation. Section 6 concludes.

2 Related Work

A basic challenge for testing distributed systems is to synchronize the correct execution sequence of test case steps. To guarantee the correct execution at large-scale, a great deal of message exchange should be managed by some synchronization component.

The Joshua system [5] uses a centralized test controller which is responsible to prepare the test cases to be executed in a distributed manner. The BlastServer [6] system is similar to Joshua. It uses the client/server approach. A server component is responsible to synchronize events, while a client component provides the communication conduit between the server component and the client application. The execution of tests is based on a queue controlled by the server component. Clients requests are queued, then consumed when needed. This approach ensures that concurrent test sequences run to completion.

At large-scale, this approach will require a large effort from the controller to synchronize messages. For instance, one willing to stress test a DHT decides to insert a large amount of data (in the form of $\{key,value\}$) by several peers. This requires to send synchronization messages to all of these peers. Unlike our approach, the number of peers will directly impact the performance of the controller. Due to this impact, the test synchronization time will be greater than the test execution time.

A more scalable approach [7] uses a complete distributed tester architecture, which is closer to our approach. It divides test cases in small parts called *partial test cases* (PTC). Each PTC is assigned to a distributed tester and can be executed in parallel to another PTC with respect to a function that controls mutual exclusivity. The behavior of the distributed testers is controlled by a *Test Coordination Procedure* (TCP) which coordinates the PTCs execution by synchronization events. Through this approach, different nodes can execute different test steps, however, the same test step cannot be executed in parallel by different nodes.

The MRUnit system[1] is designed to unit testing of MapReduce systems (MR)[2] [8]. MR has two components: map and reduce. The map component reads a set of records (using several map instances), does a computation and outputs a set of records. The reduce component consumes this output (also using several reduce instances) to group all the results together, and presents the

[1] MRUnit Project, http://archive.cloudera.com/docs/mrunit

[2] MR are widely used by Google and Yahoo to compute very large amounts of data, such as crawled documents, web request log, data warehousing, etc.

answer to a MR computation. The MRUnit hooks a test driver to each component (the driver is analogous to the JUnit's). However, the driver only verifies the computation of the same job at maps/reduces. It does not verify complex computation where different maps compute different jobs.

The P2PTester [9] and the Pigeon [10] systems avoid the central controller to address the scalability issue. They rely on a communication layer to monitor the network and log the exchanged messages. Assuming that these platforms aim to verify correctness, they must check the log files after the test execution using an oracle approach. However, all of them require the inclusion of additional code in the SUT source code. This inclusion can be either manual, for instance using specific interfaces, like in P2PTester, or automatic, using reflection and aspect-oriented programming, like in Pigeon. The inclusion of additional code is error-prone since the added code may produce errors and contaminate the test results. Furthermore, it is hard to verify whether the error came from the system under test or the testing architecture.

3 Testing Large Scale Distributed Systems

Grid and P2P systems are distributed applications, and should be first tested using appropriate tools dedicated to distributed system testing. Distributed systems are commonly tested using conformance testing [11]. The purpose of conformance testing is to determine to what extent the implementation of a system conforms to its specification. The tester specifies the system using Finite State Machines [12,13,14], Labeled Transition Systems [15,16,17] and uses this specification to generate a test suite that is able to verify (totally or partially) whether each specified transition is correctly implemented. The tester then observes the events sent among the different nodes of the system and verifies that the sequence of events corresponds to the state machine (or to the transition system).

The classical architecture for testing a distributed system, illustrated by the UML deployment diagram presented in Figure 3, consists of a *test controller* which sends the test inputs, controls the synchronization of the distributed system and receives the outputs (or local verdicts) of each node of the system under test (SUT). Note that the tester controller and the testers execute on different *logical nodes* (i.e. independent process) that may be deployed on the same *physical node* (i.e. computer device). This architecture is similar to the ISO 9646 conformance testing architecture [18]. In many cases, the distributed system under test is perceived as a single application and it is tested using its external functionalities, without considering its components (i.e. black-box testing). The tester in that case must interpret results which include non-determinism since several input/outputs orderings can be considered as correct.

The observation of the outputs for a distributed system can also be achieved using the traces (i.e. logs) produced by each node. The integration of the traces of all nodes is used to generate an event timeline for the entire system. Most of these techniques do not deal with large scale systems, in the sense that they target a small number of communicating nodes. In the case of Grid and P2P

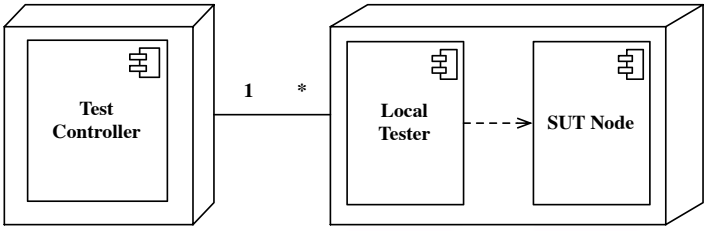


Fig. 1. Typical Centralized Tester Architecture

systems, the tester must observe the remote interface of peers to observe their behavior and must deal with a potentially large number of nodes. Writing test cases is then particularly difficult, because non-trivial test cases must execute test steps on different nodes. Consequently, synchronization among test steps is necessary to control the execution sequence of the whole test case.

Analyzing the specific features of Grid and P2P system, we remark that they are distributed systems, but the existing testing techniques for distributed systems do not address the issue of synchronization when a large number of nodes are involved. Moreover, the typical centralized tester architecture can be a bottleneck when building a testing framework for these systems.

3.1 Test Case Sample

A test case noted τ is a tuple $\tau = (S^\tau, V^\tau)$ where V^τ is a set of local verdicts and S^τ is a set of test steps. A test step is also a tuple $S = (\Psi^S, \theta^S, T^S)$ where Ψ is a set of instructions, θ is the interval of time in which the tests step should be executed and T is a set of testers that should execute this test step.

The Ψ set may contain three different kinds of instructions: (i) calls to the IUT public interface; (ii) calls to the tester interface and (iii) any statement in the test case programming language. The time interval θ sets the expected running time for Ψ and is necessary to avoid locks.

Let us illustrate these definitions with a simple distributed test case (see table 1). The aim of this test case is to detect errors on a Distributed Hash Table (DHT) implementation. More precisely, it verifies if a node correctly resolves a given query that retrieves data 3 seconds after its insertion, in less than 100 milliseconds and continues to do so in the future.

This test case involves three testers $T^\tau = \{t_0, t_1, t_2\}$ managing nine steps $S^\tau = \{s_1, ..., s_9\}$ on three nodes $N = \{n_0, n_1, n_2\}$. The goal of the first three steps is to populate the DHT. The only local verdict is given by t_0. If the data retrieved by t_0 is the same as the one inserted by t_2, then the verdict is *pass*. If the data is not the same, the verdict is $fail$. If p_0 is not able to retrieve any data, then the verdict is *inconclusive*.

Table 1. Simple test case

Test Step	Testers Ψ	(Instructions)	θ
(1)	0,1,2	Join the system;	
(2)	*	Pause;	
(3)	2	Insert the string "One" at key 1;	
		Insert the string "Two" at key 2;	
(4)	*	Wait 3 sec.	
(5)	0	Retrieve data at key 1;	100 msec.
		Retrieve data at key 2;	
(6)	1	Leave the system;	
(7)	0	Retrieve data at key 1;	100 msec.
		Retrieve data at key 2;	
(8)	0,2	Leave the system;	
(9)	0	Calculate a verdict;	

3.2 Requirements for Testing Architecture

In this section, we enumerate the requirements for large-scale, distributed testing architectures.

Scalability. The performance of the testing environment and specially of test step sequencing may impact the behavior of the system under test. Thus, in order to reduce the impact on the SUT, the architecture should scale at least as well as the SUT.

Test step sequencing/synchronization. This requirement ensures that test steps are executed in the correct sequence and that each test step has finished in all testers before the execution of the subsequent test step. Complex test step sequencing leads to an overhead of the synchronization management.

Volatility management. In some distributed systems, such as P2P, node volatility is a common behavior. The architecture must be able to simulate the entry and the departure of nodes in a fine way, without interferences to the sequencing process.

Shared variables. During a test, some variables are only known dynamically and by few nodes, e.g., node ids, number of nodes, etc. The architecture should provide a mechanism that allows testers to share variables.

Error/Log handling. Typical large scale systems often use the same software in several nodes, which generate similar logs. The architecture should be able to do both, reconstruct the global log timeline and detect and ignore duplicate logs.

Except for the test step sequencing, the centralized architecture does not fulfill the requirements enumerated above. The next section presents a distributed architecture that supersedes the classical centralized architecture and is able to meet these requirements.

4 Distributed Architecture

In this section, we present an alternative to the centralized testing architecture. Our proposal consists of organizing testers in a tree structure, similarly to the overlay network used by GFS-Btree [19]. The main idea is to drop the test controller and introduce the notion of *hybrid testers*, i.e., a node that is both, tester and controller.

When executing a test case, the root tester dispatches test steps to its child testers, which in turn, dispatch test steps to their children. Once a step is executed, the leaves (which are only testers), send their results to their parents, and so forth, until the root receives all results. Then, the root dispatch the next test steps in the same way.

The performance of the synchronization is highly related to the tree topology and to the distance between nodes. For instance, a high tree, i.e., with a small order, would increase the communication delay between the root and the leaves, while a wide tree, i.e., with a high order, would overload the hybrid testers.

If the distance between nodes is not taken into consideration, the communication delay between the root and the leaves may vary substantially. For instance, a path between the root and a given leaf could be composed of nodes belonging to the same physical node, while another path could be composed of intercalated nodes from two distant physical nodes.

Besides the expected impact in the synchronization performance, the distributed architecture also impacts on log handling, since hybrid testers are able to treat logs and errors before pulling them up to their parents.

In the next sections, we present two different approaches to build the tester tree. The first one builds a balanced tree, where testers are placed accordingly to their arriving time: the first nodes to connect are placed at the top of the tree. The second one introduces a more optimized structure, which avoids placing two hybrid testers at the same physical node.

4.1 Balanced Tree

In this approach, testers are organized in a tree of order m, where all nodes have at most m children and all children of a node containing less dans m children are leaves. Figure 2 presents an example of tester organization using a balanced Tree of order 2, containing 12 testers: 1 root, 6 hybrid and 5 leaf testers.

While in theory this organization should not give good performance results since the hybrid testers are concentrated in the nearest physical nodes, in practice it does, as we will see in Section 5. This because the leaf testers, which receive the test steps in last, are placed in low-charged physical nodes, i.e., without the controller overhead. Thus, they can execute test steps quicker than other testers and reduce the global execution time.

4.2 Optimized Tree

In order to better balance the testers through the physical nodes, we added an extra constraint to the above presented tree: physical nodes contains at most one

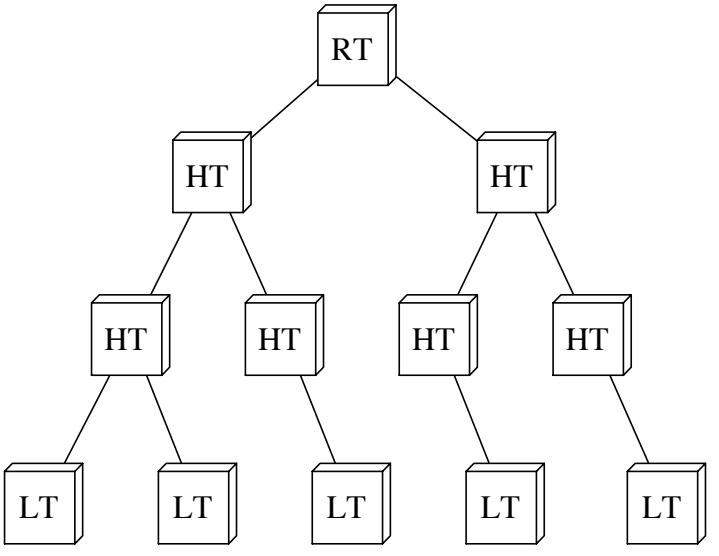


Fig. 2. Balanced Tree, 12 testers, order=2

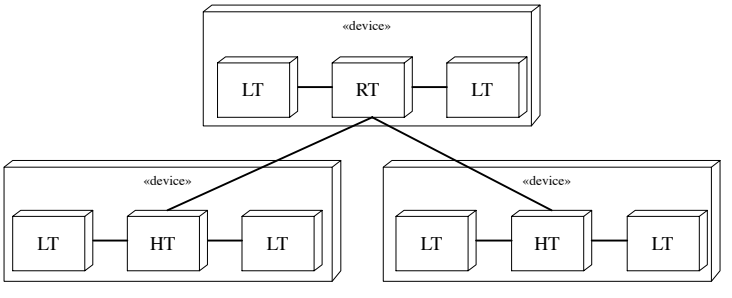


Fig. 3. Optimized Tree

hybrid tester. Our goal was to take advantage of the context of our experiments, a grid computer with an excellent network latency. Figure 3 presents an example of tester organization using a optimized Tree of order 2, containing 9 testers: 1 root, 2 hybrid and 6 leaf testers.

Similar to the precedent tree, hybrid testers controls up to m children. However, they also control all testers belonging to the same physical node, their dependents. Contrary to the number of children, the number of dependents is not fixed, it depends on the number of testers running in the same physical node. Dependents are always leaf testers.

5 Experimentation

In this section, we present the performance evaluation of the distributed with the centralized testing architecture. Our goal is to evaluate to which extent the distributed architecture reduces the overhead of test step synchronization management. We also evaluate the performance of the architecture in different configurations, varying the tree order, the number of testers and the number of test case steps.

All of our experiments were run on the Grid5000 platform[3] using several clusters running GNU/Linux (up to 256 nodes) connected by a 10Gbps network. We implemented our approaches in Java (version 1.5) using Remote Method Invocation (RMI) for the communication among testers. In all experiments, each tester is configured to run in a single Java virtual machine. The testers were allocated equally through the nodes up to 32 testers per physical node to obtain a large-scale testing environment. Since we can have full control over these clusters during experimentation, our experiments are reproducible. The implementation produced for this paper is open-source and can be found in our web page[4]. It was used to test two popular open-source P2P systems [20]: FreePastry and OpenChord.

5.1 Test Step Synchronization for Up to 8,192 Testers

To measure the response time of test step synchronization, we submitted a fake test case, composed of empty test steps, across a different range of testers. Then, for each step, we measured the whole execution time, which comprises remote invocations, execution of empty test steps and confirmations. First, we verify the performance at the centralized test architecture. Then, we compare the result of the centralized with the distributed architecture.

The evaluation works as follows. We deploy the fake test case through several testers. The testers register their test steps with the coordinator. Once the registration is finished, the coordinator executes all the test case steps and measures their execution time. The evaluation finishes when all steps are executed.

The fake test case contains 8 empty test steps (we choose this number arbitrarily) and is executed until a limit of 8192 testers running in parallel. Figure 4 presents the response time for synchronization for a varying number of testers. The centralized test controller showed a linear performance in terms of response time as the number of testers increases. Although this result was expected, its implementation is straightforward and can be even used while testing in small-scale environments. Our target, however, is testing in large-scale environments.

Figure 5 compares the response time of the centralized architecture solution with the distributed one, using both, the balanced and the optimized tree. We observe that the centralized architecture leads to an exponential increase of the overhead, which makes it unscalable. For small-scale (i.e. less than 1,024 nodes),

[3] The Grid5000 platform, http://www.grid5000.fr
[4] Peerunit project, http://peerunit.gforge.inria.fr

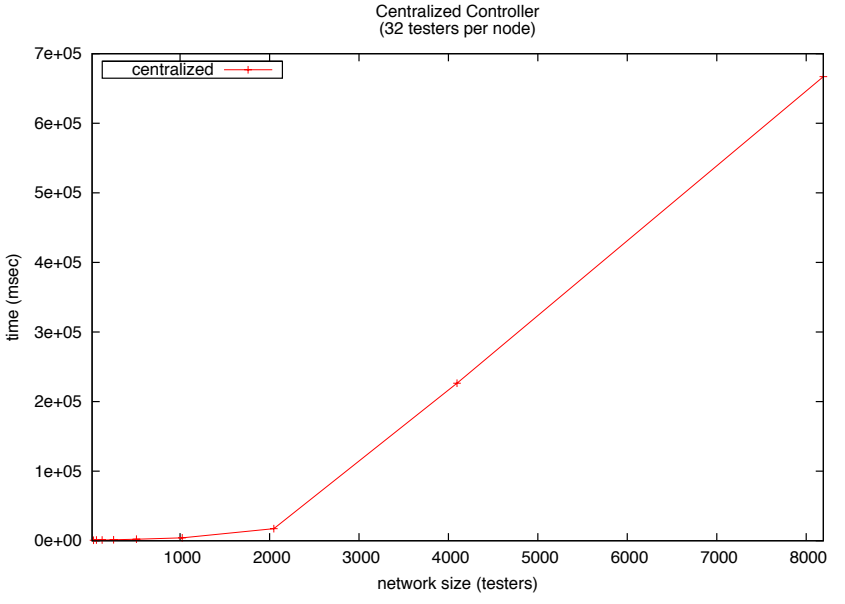


Fig. 4. Centralized Test Architecture

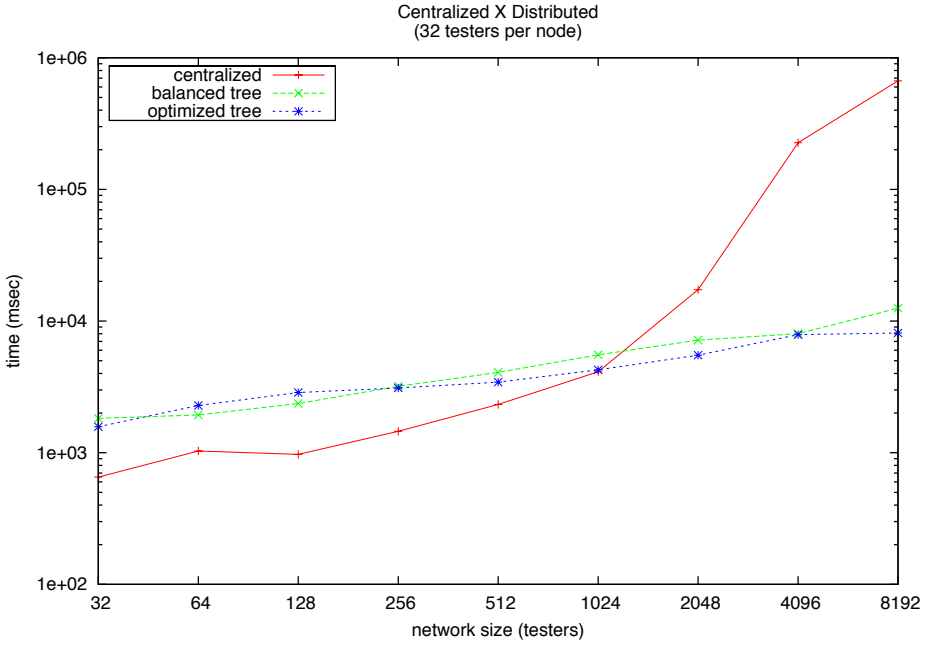


Fig. 5. Centralized X Distributed Architectures

the centralized architecture is more efficient than the distributed one. The distributed architecture, in both configurations, leads to a satisfying overhead when the number of testers increases. A disappointing phenomenon occurs, which is the very similar slopes we obtain with the two different trees. To understand the phenomenon, we study the parameters that impact the performance of these architectures, in particular the number of logical nodes deployed per physical node and the number of test steps.

5.2 Varying the Tree Order

To quantify the impact of the optimization gains on the optimized tree, we vary its order (m). The order is a parameter configured before the execution of the test case (the same test case used above, with 8 empty test steps). As expected, independently from the tree order, the time increases logarithmically.

However, as showed in Figure 6, the impact of varying the tree order is not evident. Beyond 128 testers and from $m = 2$ to $m = 8$ response time is inversely proportional to m and directly proportional to the tree height. Response time increases since the messages exchanged between the root and the leaves have a longer path among physical nodes. For instance, for 4,096 testers (256 physical nodes), the height of the tree[5] is 8 for $m = 2$ and 3 for m=8.

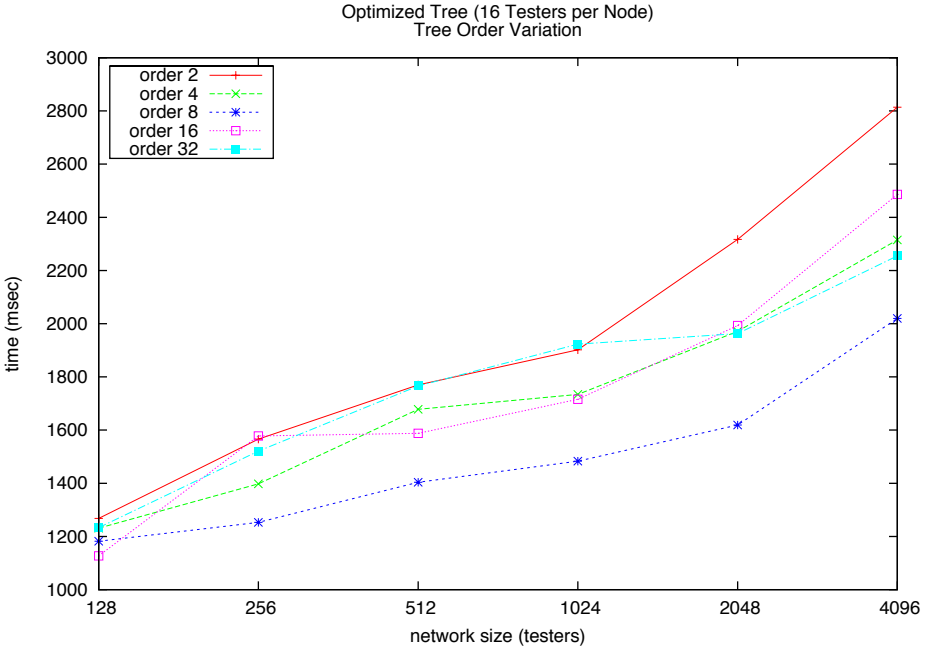


Fig. 6. Varying the tree order

[5] Considering only hybrid testers

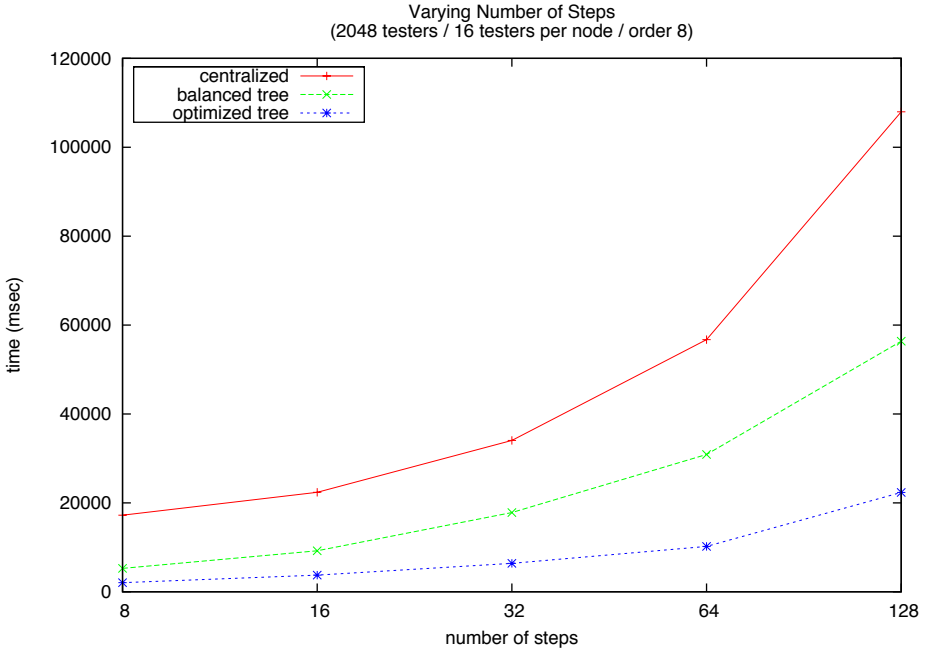


Fig. 7. Varying the number of test steps

Still beyond 128 testers, but from $m = 8$ to $m = 64$, response time is directly proportional to m. In these cases, the overhead of controlling more testers is more important than the communication overhead of heaving a higher tree, since the hight varies little. For instance, from 2 ($m = 64$, $m = 32$ and $m = 16$) to 3 ($m = 8$) for 4,096 testers.

5.3 Varying the Number of Test Steps

We also investigate the impact of the number of test steps. A larger number of test steps requires a larger effort from the controller to keep the execution sequence and dispatch more steps. We limit the results to 2048 testers to easy the reading (the overhead on the central controller increases exponentially as the testing scale grows larger).

Figure 7 shows that both architectures scale up linearly, as we expected. The distributed architecture using the optimized tree yields better results in several orders of magnitude.

6 Conclusion

In this paper, we presented a distributed architecture for testing large-scale distributed systems. The architecture organizes testers in a balanced tree and supersedes the traditional centralized test controller by several *hybrid* testers,

which are controllers and testers at the same time. After implementing the architecture, we conducted several experiments with up to 8.192 logical nodes as means to evaluate the overall performance of the architecture.

The experiments showed that our architecture scales up logarithmically, which is an important requirement for testing large-scale systems. The experiments also showed that several factors may impact the synchronization performance of the architecture: number of nodes, number of testers per node, the order of the tree, among others.

The tree order emerged as an important yet complex factor for fine-tuning the performance. The best value seems to come from an equilibrium between the height of the tree and number of testers controlled by each hybrid tester. We believe that the best tree order for a given test can be calculated before deploying the testbed.

In order to improve our architecture, we intend to implement a logging facility which is able to build a common timeline when merging logs from different nodes, and discard similar log entries.

Finally, we intend to demonstrate that false verdicts may be assigned (i.e., false-positives or false-negatives) due to low efficient testing architectures. For instance, a DHT routing mechanism, which is used to exchange messages, requires a very low time to update its entries. In large-scale testing environments, the DHT would perform this update faster than the test steps synchronization of the centralized testing architecture. This could lead to false-positive verdict.

References

1. Foster, I.T., Iamnitchi, A.: On death, taxes, and the convergence of peer-to-peer and grid computing. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 118–128. Springer, Heidelberg (2003)
2. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenkern, S.: A scalable content-addressable network. ACM SIGCOMM (2001)
3. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peertopeer lookup service for internet applications. ACM, New York (2001)
4. Jagatheesan, A., Rajasekar, A.: Data grid management systems. In: SIGMOD 2003: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 683–683. ACM, New York (2003)
5. Kapfhammer, G.M.: Automatically and transparently distributing the execution of regression test suites. In: Proceedings of the 18th International Conference on Testing Computer Software, Washington, D.C. (2001)
6. Long, B., Strooper, P.A.: A case study in testing distributed systems. In: Proceedings 3rd International Symposium on Distributed Objects and Applications (DOA 2001), pp. 20–30 (2001)
7. Ulrich, A., Zimmerer, P., Chrobok-Diening, G.: Test architectures for testing distributed systems. In: Proceedings of the 12th International Software Quality Week (1999)
8. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)

9. Dragan, F., Butnaru, B., Manolescu, I., Gardarin, G., Preda, N., Nguyen, B., Pop, R., Yeh, L.: P2ptester: a tool for measuring P2P platform performance. In: BDA conference (2006)
10. Zhou, Z., Wang, H., Zhou, J., Tang, L., Li., K.: Pigeon: A framework for testing peer-to-peer massively multiplayer online games over heterogeneous network. In: 3rd IEEE Consumer Communications and Networking Conference, CCNC (2006)
11. Schieferdecker, I., Li, M., Hoffmann, A.: Conformance testing of tina service components - the ttcn/ corba gateway. In: IS&N, pp. 393–408 (1998)
12. Chen, W.H., Ural, H.: Synchronizable test sequences based on multiple uio sequences. IEEE/ACM Trans. Netw. 3, 152–157 (1995)
13. Hierons, R.M.: Testing a distributed system: generating minimal synchronised test sequences that detect output-shifting faults. Information and Software Technology 43, 551–560 (2001)
14. Chen, K., Jiang, F.: A new method of generating synchronizable test sequences that detect output-shifting faults based on multiple uio sequences. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 1791–1797. Springer, Heidelberg (2007)
15. Jard, C.: Principles of distribute test synthesis based on true-concurrency models. Technical report, IRISA/CNRS (2001)
16. Pickin, S., Jard, C., Le Traon, Y., Jéron, T., Jézéquel, J.M., Le Guennec, A.: System test synthesis from UML models of distributed software. ACM - 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems (2002)
17. Jard, C., Jéron, T.: TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. Int. J. Softw. Tools Technol. Transf. (2005)
18. 9646-1, I.: Conformance Testing Methodology and Framework—Part 1: General Concepts (1994)
19. Li, Q., Wang, J., Sun, J.G.: Gfs-btree: A scalable peer-to-peer overlay network for lookup service. In: Li, M., Sun, X.-H., Deng, Q.-n., Ni, J. (eds.) GCC 2003. LNCS, vol. 3032, pp. 340–347. Springer, Heidelberg (2004)
20. de Almeida, E.C., Sunyé, G., Traon, Y.L., Valduriez, P.: A framework for testing peer-to-peer systems. In: 19th International Symposium on Software Reliability Engineering (ISSRE 2008). IEEE Computer Society Press, Redmond (November 2008)

Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction*

Fides Aarts[1,**], Bengt Jonsson[2], and Johan Uijen[1]

[1] Inst. f. Comp. and Inf. Sciences, Radboud University, Nijmegen, The Netherlands
{f.aarts,j.uijen}@cs.ru.nl
[2] Department of Computer Systems, Uppsala University, Sweden
bengt@it.uu.se

Abstract. In order to facilitate model-based verification and validation, effort is underway to develop techniques for generating models of communication system components from observations of their external behavior. Most previous such work has employed regular inference techniques which generate modest-size finite-state models. They typically suppress parameters of messages, although these have a significant impact on control flow in many communication protocols. We present a framework, which adapts regular inference to include data parameters in messages and states for generating components with large or infinite message alphabets. A main idea is to adapt the framework of predicate abstraction, successfully used in formal verification. Since we are in a black-box setting, the abstraction must be supplied externally, using information about how the component manages data parameters. We have implemented our techniques by connecting the LearnLib tool for regular inference with the protocol simulator ns-2, and generated a model of the SIP component as implemented in ns-2.

1 Introduction

Model-based techniques for verification and validation of communication protocols and reactive systems, including model checking and model-based testing [7] have witnessed drastic advances in the last decades, and are being applied in industrial settings (e.g., [20]). They require formal models that specify the intended behavior of system components, which ideally should be developed during specification and design. However, the construction of models typically requires significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. Automated support for constructing models of the behavior of implemented components would therefore be extremely useful, e.g., for regression testing, for replacing manual testing by model based testing, for producing models of standardized protocols, for analyzing whether an existing system is vulnerable to attacks, etc. Techniques, developed

* Supported in part by EC Proj. 231167 (CONNECT).
** Supported in part by the EC Progr. No. 214755 (QUASIMODO).

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 188–204, 2010.

for program analysis, that construct models from source code (e.g., [4,19]) are often of limited use, due to the presence of library modules, third-party components, etc., that make analysis of source code difficult. We therefore consider techniques for constructing models from observations of their external behavior.

The construction of models from observations of component behavior can be performed using regular inference (aka automata learning) techniques [3,11,22,31]. This class of techniques is now receiving increasing attention in the testing and verification community, e.g., for regression testing of telecommunication systems [18,21], for integration testing [17,23], security protocol testing [33], and for combining conformance testing and model checking [28,16]. One of the most used algorithms for regular inference, L^*, poses a sequence of *membership queries*, each of which observes the component's output in response to a certain input string, and produces a minimal deterministic finite-state machine which conforms to the observations. If the sequence of membership queries is sufficiently large, the produced machine will be a model of the observed component.

Since regular inference techniques are designed for finite-state models, previous applications to model generation have been limited to generating a moderate-size finite-state view of the system behavior, implying that the alphabet must be made finite, e.g., by suppressing parameters of messages. However, parameters have a significant impact on control flow in typical protocols: they can be sequence numbers, configuration parameters, agent and session identifiers, etc. The influence of data on control flow is taken into account by model-based test generation tools, such as ConformiQ Qtronic [20]. It is therefore important to extend inference techniques to handle message alphabets and state-spaces with structures containing data parameters with large domains.

In this paper, we present a general framework for generating models of protocol components with large or infinite structured message alphabets and state spaces. The framework is inspired by predicate abstraction [24,9], which has been successful for extending finite-state model checking to large and infinite state spaces. In contrast to that work, however, we are now in a black-box setting, where an abstraction cannot be defined based on the source code or model of a component, since it is not accessible. Instead, we must construct an externally supplied abstraction, which translates between a large message alphabet of the component to be modeled and a small finite alphabet of the regular inference algorithm. Via regular inference, a finite-state model of the abstracted interface is inferred. The abstraction can then be reversed to generate a faithful model of the component.

We describe how to construct a suitable abstraction, utilizing pre-existing knowledge about which operators are sufficient to express guards and operations on data in a faithful model of the component. We have implemented our techniques by connecting the LearnLib tool for regular inference with the protocol simulator ns-2, which provides implementations of standard protocols. We have used it to generate models of ns-2 protocol implementations.

Related Work. Regular inference techniques have been used for several tasks in verification and test generation, e.g., to create models of environment constraints

with respect to which a component should be verified [10], for regression testing to create a specification and test suite [18,21], to perform model checking without access to source code or formal models [16,28], for program analysis [2], and for formal specification and verification [10]. Groz, Li, and Shahbaz [23,32,17] extend regular inference to Mealy machines with data values, for use in integration testing. They use only a finite set of the data values in the obtained model, and do not infer internal state variables. Shu and Lee [33] learns the behavior of security protocol implementations for a finite subset of input symbols, which can be extended in response to new information obtained in counterexamples. Lorenzoli, Mariani, and Pezzé infer models of software components that consider both sequence of method invocations and their associated data parameters [25,26]. They use a passive learning approach where the model is inferred from a given sample of traces. They infer a finite control structure capturing possible sequences of method invocations, by an extension of the k-tails algorithm, and using Daikon [8] to infer guards and relations on method parameters. In contrast to their passive learning, we use an active learning approach where new queries may be supplied to the system; this is an added requirement but allows to generate a more informative sample.

In previous work, we have considered extensions of regular inference to handle data parameters. In [5], we show how guards on boolean parameters can be refined lazily. This technique for maintaining guards have inspired the more general notion of abstractions on input symbols presented in the current paper. We have also proposed techniques to handle infinite-state systems, in which parameters of messages and state variables are from an unbounded domain, e.g., for identifiers [6], and timers [13,12]. These extensions are specialized towards a particular data domain, and their worst-case complexities do not immediate suggest an efficient implementation. This paper proposes a general framework for incorporating a range of such data domains, into which techniques specialized for different data domains can be incorporated, and which we have also evaluated on realistic protocol models.

Organization. In the next section, we give basic definitions of Mealy machines. We present our inference and abstraction techniques in Section 3. The application to SIP is reported in Section 4. Section 5 contains conclusions and directions for future work.

2 Mealy Machines

Basic Definitions. We will use *Mealy machines* to model communication protocol entities. A *Mealy machine* is a tuple $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ where Σ_I is a nonempty set of *input symbols*, Σ_O is a nonempty set of *output symbols*, Q is a nonempty set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma_I \to Q$ is the *transition function*, and $\lambda : Q \times \Sigma_I \to \Sigma_O$ is the *output function*. The sets of states and symbols can be finite or infinite: if they are all finite we say that the Mealy machine is *finite*. Elements of Σ_I^* are called *input strings*, and elements

of Σ_O^* are called *output strings*. We extend the transition and output functions to input strings in the standard way, by defining:

$$\begin{array}{ll} \delta(q,\varepsilon) &= q \\ \delta(q,ua) &= \delta(\delta(q,u),a) \end{array} \qquad \begin{array}{ll} \lambda(q,\varepsilon) &= \varepsilon \\ \lambda(q,ua) &= \lambda(q,u)\lambda(\delta(q,u),a) \end{array}$$

where $u \in \Sigma_I^*$. We define $\lambda_{\mathcal{M}}(u) = \lambda(q_0, u)$ for $u \in \Sigma_I^*$. Two Mealy machines $\mathcal{M}$ and $\mathcal{M}'$ with the same set of input symbols are *equivalent* if $\lambda_{\mathcal{M}}(u) = \lambda_{\mathcal{M}'}(u)$ for all input strings u.

Intuitively, a Mealy machine behaves as follows. At any point in time, the machine is in some state $q \in Q$. When supplied with an input symbol $a \in \Sigma_I$, it responds by producing an output symbol $\lambda(q,a)$ and transforms itself to a new state $\delta(q,a)$. We use the notation $q \xrightarrow{a/b} q'$ to denote that $\delta(q,a) = q'$ and $\lambda(q,a) = b$; in this case $q \xrightarrow{a/b} q'$ is called a *transition* of $\mathcal{M}$.

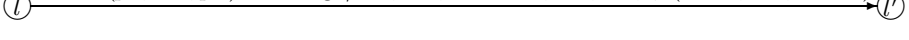
The Mealy machines that we consider are *deterministic*, meaning that for each state q and input symbol a exactly one next state $\delta(q,a)$ and output string $\lambda(q,a)$ is possible.

Symbolic Representation. In order to conveniently model entities of communication protocols, we should be able to describe messages as consisting of a message type with a number of parameters, and states as consisting of a control location and values of a set of state variables. We therefore introduce a symbolic representation of Mealy machines, similar to Extended Finite State Machines [29].

So, assume a set of *action types*. Each action type α has a certain *arity*, which is a tuple of *domains* (a domain is a set of allowed data values) $\mathcal{D}_{\alpha,1}, \ldots, \mathcal{D}_{\alpha,n}$ (where n depends on α). For a set I of action types, let Σ_I be the set of terms of form $\alpha(d_1, \ldots, d_n)$, where $d_i \in \mathcal{D}_{\alpha,i}$ is a data value in the appropriate domain for each i with $1 \leq i \leq n$. Assume a set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, to be used as placeholders for parameters of symbols.

Also, assume a set of *state variables*. Each state variable v has a domain of possible values, and a unique initial value. For a set V of state variables, let a *V-valuation* σ be a partial mapping from V to data values in their respective domains, and let σ_0^V be the *V-valuation* which maps each variable in V to its initial value. We extend V-valuations to expressions over state variables in the natural way; for instance, if $\sigma(v_3) = 8$, then $\sigma(2 * v_3 + 4) = 20$.

Definition 1. A *Symbolic Mealy machine* is a tuple $\mathcal{SM} = \langle I, O, L, l_0, V, \longrightarrow \rangle$, where I and O are disjoint finite sets of actions (*input actions* and *output actions*), where L is a finite set of *locations*, where $l_0 \in L$ is the *initial location*, where V is a finite set of *state variables*, and where $\longrightarrow$ is a finite set of *symbolic transitions*, each of form



in which l and l' are locations, $\alpha \in I$ and $\beta \in O$ are actions, $p_1, \ldots, p_n$ are distinct formal parameters, $v_1, \ldots, v_k$ are distinct state variables in V, in which g (the guard*) is a boolean expression over $p_1, \ldots, p_n$ and V, and in which $e_1, \ldots, e_k$ and*

${e^{out}}_1, \ldots, {e^{out}}_m$ are expressions over $p_1, \ldots, p_n$ and V. We assume that the arities of α and β and the domains of $v_1, \ldots, v_k$ are respected. For each input action $\alpha \in I$, each location $l \in L$, and each V-valuation σ, the set $\longrightarrow$ must contain exactly one symbolic transition of the above form for which $\sigma(g[d_1, \ldots, d_n/p_1, \ldots, p_n])$ is true. □

In the following, we will use $\overline{p}$ for $p_1, \ldots, p_n$ and $\overline{d}$ for $d_1, \ldots, d_n$.

Intuitively, a symbolic transition of the above form denotes that whenever a Symbolic Mealy machine (SMM for short) $\mathcal{SM}$ is in location l and some input symbol of form $\alpha(\overline{d})$ is received, such that the guard g is satisfied when the formal parameters $\overline{p}$ are bound to the data values $\overline{d}$, then the state variables among $v_1, \ldots, v_k$ are simultaneously assigned new values, an output symbol obtained by evaluating $\beta({e^{out}}_1, \ldots, {e^{out}}_m)$, is generated, and $\mathcal{SM}$ moves to location l'.

The meaning of a SMM $\mathcal{SM} = \langle I, O, L, l_0, V, \longrightarrow \rangle$ is defined by its denotation, which is the Mealy machine $\mathcal{M}_{\mathcal{SM}} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$, where Σ_I is obtained from I as described earlier, and similarly for Σ_O, where Q is the set of pairs $\langle l, \sigma \rangle$ consisting of a location $l \in L$ and a V-valuation σ, where q_0 is the pair $\langle l_0, \sigma_0^V \rangle$, and where δ and λ are such that for any symbolic transition in $\longrightarrow$ of form

$$l \xrightarrow{\alpha(p_1, \ldots, p_n) \textbf{ when } g \;/\; v_1, \ldots, v_k := e_1, \ldots, e_k \;;\; \beta({e^{out}}_1, \ldots, {e^{out}}_m)} l'$$

for any V-valuation σ and data values $\overline{d}$ with $\sigma(g[\overline{d}/\overline{p}])$ being true, it holds that

- $\delta(\langle l, \sigma \rangle, \alpha(\overline{d})) = \langle l', \sigma' \rangle$, where σ' is the V-valuation such that $\sigma'(v_i) = \sigma(e_i[\overline{d}/\overline{p}])$ for $1 \leq i \leq k$, and $\sigma'(v) = \sigma(v)$ if v is not among $v_1, \ldots, v_k$,
- $\lambda(\langle l, \sigma \rangle, \alpha(\overline{d})) = \beta(\sigma'(e_1^{out}[\overline{d}/\overline{p}]), \ldots, \sigma'(e_m^{out}[\overline{d}/\overline{p}]))$.

We use $\lambda_{\mathcal{SM}}$ to denote $\lambda_{\mathcal{M}_{\mathcal{SM}}}$, and say that $\mathcal{SM}$ and $\mathcal{SM}'$ are equivalent if $\lambda_{\mathcal{SM}}(u) = \lambda_{\mathcal{SM}'}(u)$ for all input strings u. We can similarly say that an SMM is equivalent to a Mealy machine.

Example. We consider a simplistic SMM, which models a component that services requests to set up a connection. Its sets of input and output actions are $I = \{REQ, CONF\}$ and $O = \{REPL, ACK, REJ\}$. The arity of REJ is the empty tuple (i.e., it has no parameters), and the arity of the other actions is the pair $\mathbb{N}, \mathbb{N}$ i.e., input symbols are of form $REQ(id, sn)$ and $CONF(id, sn)$ where id

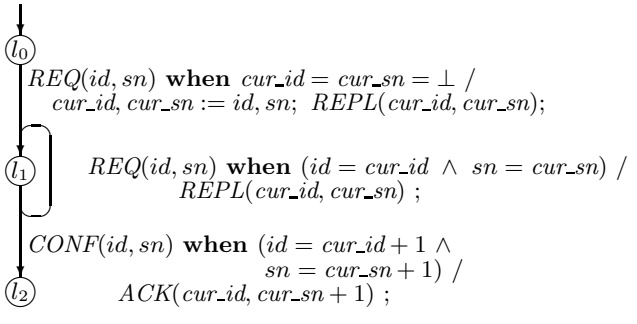


Fig. 1. Symbolic transitions of SMM in Example

and *sn* are natural numbers, and analogously for output symbols. There are two state variables, *cur_id* and *cur_id*, both ranging over $\mathbb{N} \cup \bot$, with $\bot$ (a distinguished symbol denoting "undefined") as initial values. The set of locations ($\{l_0, l_1, l_2\}$) and symbolic transitions are shown in Figure 1. We have suppressed symbolic transitions where the machine replies with the output symbol *REJ* and lead to a terminal error state (also not shown). For each location and input action, there is one such symbolic transition, guarded by the negation of the guard on the transition from the same location with the same input action. □

3 Inference of Symbolic Mealy Machines

3.1 The Setting of Inference

The problem considered in this paper is the following: Given an SMM $\mathcal{SM}$, how can a component, called the *Learner*, which communicates with $\mathcal{SM}$, infer an SMM equivalent to $\mathcal{SM}$ by observing how $\mathcal{SM}$ responds to a set of input strings. We use the same setting as Angluin's L^* algorithm [3]. There the *Learner* initially knows the static interface of $\mathcal{SM}$, i.e., the sets I and O of input and output actions together with their arities. It may then ask a sequence of *membership queries*; each one supplying a chosen input string $u \in (\Sigma_I)^*$ and observing the response $\lambda_{\mathcal{SM}}(u)$. After a "sufficient" number of membership membership queries the *Learner* can build a "stable" hypothesis $\mathcal{H}$ from the obtained information. The hypothesis $\mathcal{H}$ should of course agree with $\mathcal{SM}$ on the performed membership queries (i.e., $\lambda_{\mathcal{SM}}(u) = \lambda_{\mathcal{H}}(u)$ whenever u was supplied in a membership query), but must make suitable generalizations for other input strings. In order to increase confidence in the hypothesis $\mathcal{H}$, one can subject $\mathcal{SM}$ to thorough conformance testing or longer-term monitoring in order to search for input strings on which $\mathcal{SM}$ disagrees with $\mathcal{H}$. In the setting of L^*, this is idealized as an *equivalence query*, which asks whether $\mathcal{H}$ is equivalent to $\mathcal{SM}$, and which is replied with either *yes*, meaning that $\mathcal{H}$ is indeed equivalent to $\mathcal{SM}$, or with *no* and a *counterexample*, which is an input string $u \in \Sigma_I^*$ such that $\lambda_{\mathcal{SM}}(u) \neq \lambda_{\mathcal{H}}(u)$.

For finite Mealy machines the above problem is well understood. The L^* algorithm, which has been adapted to Mealy machines by Niese [27], generates hypotheses $\mathcal{H}$ that are the minimal Mealy machines that agree with the performed membership queries. It is implemented in the LearnLib tool [30], which also realizes approximate equivalence queries by test suites of user-controllable size.

3.2 Inference Using Abstraction

The L^* algorithm works only for finite Mealy machines. In order to use it for inferring models of large or infinite-state SMMs, we adapt ideas from predicate abstraction [24,9], which has been successful for extending finite-state model checking to large and infinite state spaces.

In the following, consider an SMM $\mathcal{SM} = \langle I, O, L, l_0, V, \longrightarrow \rangle$ with $\mathcal{M}_{\mathcal{SM}} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$, in which Σ_I, Σ_O, and Q may be large or infinite.

To apply regular inference to $\mathcal{SM}$, we should define an abstraction from Σ_I and Σ_O to (small) finite sets of *abstract* input and output symbols. For instance, in the SMM in Figure 1, symbols of form $REQ(id, sn)$ can be abstracted to symbols of form $REQ(\mathrm{ID}, \mathrm{SN})$, where ID and SN are from a small domain. Let us abstract a parameter value id by CUR if id is the "current" session identifier, and by OTHER otherwise. By the "current" session identifier, we mean the value of id received in the first symbol of form $REQ(id, sn)$. We abstract the parameter sn in a similar way. In this way, the input string $REQ(25, 4)$ $REQ(25, 7)$ is abstracted to $REQ(\mathrm{CUR}, \mathrm{CUR})$ $REQ(\mathrm{CUR}, \mathrm{OTHER})$, whereas the input string $REQ(42, 4)$ $REQ(25, 7)$ is abstracted to $REQ(\mathrm{CUR}, \mathrm{CUR})$ $REQ(\mathrm{OTHER}, \mathrm{OTHER})$. Thus, the abstraction of a symbol, such as $REQ(25, 7)$, in general depends on the previous history of symbols. In model checking using abstraction [24,9], this dependency is taken into account by letting the abstraction depend on internal state variables, such as *cur_id* and *cur_sn* in the SMM of Figure 1. However, we are now in a black-box setting where the state variables of the SMM are not accessible. Therefore, the abstraction must maintain a set of additional state variables that record relevant history information. In our example, they can be *abs_id* and *abs_sn*, where *abs_id* is assigned the value of the id parameter in the first input symbol of form $REQ(id, sn)$, and is thereafter used to decide whether id-parameters should be mapped to CUR or OTHER. Let us formalize.

Definition 2. Let I and O be disjoint finite sets of (input and output) actions. An $\langle I, O\rangle$-*abstraction* is a tuple $\mathcal{A} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, R, r_0, abstr_I, abstr_O, \delta^{\mathcal{R}}\rangle$, where

- $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$ are finite sets of *abstract* input and output symbols,
- R is a (possibly infinite) set of *local states*,
- $r_0 \in R$ is an *initial local state*,
- $abstr_I : R \times \Sigma_I \mapsto \Sigma_I^{\mathcal{A}}$ maps input symbols to abstract ones,
- $abstr_O : R \times \Sigma_O \mapsto \Sigma_O^{\mathcal{A}}$ maps output symbols to abstract ones, and
- $\delta^{\mathcal{R}} : R \times (\Sigma_I \cup \Sigma_O) \mapsto R$ updates the local state when a new input or output symbol occurs. □

Intuitively, an abstraction $\mathcal{A}$ maps input and output symbols to abstract ones, and updates its local state immediately after the occurrence of each symbol. We let $\mathcal{A}$ be implemented by a *Mapper* module, as shown in Figure 2. The *Mapper* maintains the local state r of the abstraction. Each abstract input symbol $a^{\mathcal{A}}$ supplied by the *Learner* (such as $REQ(\mathrm{CUR}, \mathrm{CUR})$), is translated by the

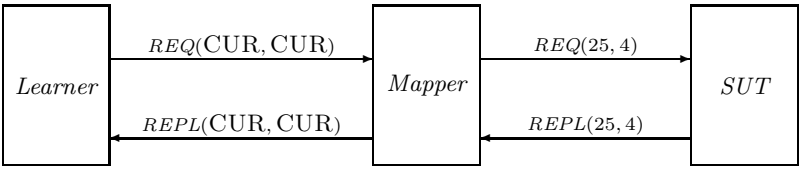


Fig. 2. Introduction of *Mapper* module

Mapper to a concrete input symbol a such that $a^{\mathcal{A}} = abstr_I(r, a)$, and sent to *SUT*. The corresponding reply b by *SUT* is translated to the abstract symbol $abstr_O(\delta^{\mathcal{R}}(r, a), b)$ and sent back to the *Learner*. Finally the local state r is updated to $\delta^{\mathcal{R}}(\delta^{\mathcal{R}}(r, a), b)$. To keep the notation simpler, we will in the following assume that the local state is updated only in response to input symbols (i.e., that $\delta^{\mathcal{R}}(r, b) = r$ for any output symbol b); the extension to the general case is straight-forward. We extend the transition and input abstraction function to input strings by:

$$\begin{array}{ll} \delta^{\mathcal{R}}(r, \varepsilon) &= q \\ \delta^{\mathcal{R}}(r, ua) &= \delta^{\mathcal{R}}(\delta^{\mathcal{R}}(r, u), a) \end{array} \qquad \begin{array}{ll} abstr_I(r, \varepsilon) &= \varepsilon \\ abstr_I(r, ua) &= abstr_I(r, u)\, abstr_I(\delta^{\mathcal{R}}(r, u), a) \end{array}$$

In particular, $abstr_I(r_0, u)$ is the abstraction of an arbitrary input string u.

The *Learner* interacts with the combination of the *Mapper* and the *SUT*, using the finite sets $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$. In general, this combination is not a (deterministic) Mealy machine, but rather some nondeterministic state machine, since each (abstract) input symbol $a^{\mathcal{A}}$ can be translated by the *Mapper* (in state r) to any input symbol a with $a^{\mathcal{A}} = abstr_I(r, a)$: different choices of a can, in general, cause the *SUT* to move to different states and subsequently cause different (abstract) output symbols to be generated. The states of this combination, denoted $Q^{\langle \mathcal{SM}, \mathcal{A} \rangle}$, is the set of pairs in $Q \times R$ of form $\langle \delta(q_0, u), \delta^{\mathcal{R}}(r_0, u) \rangle$ for some input string $u \in {\Sigma_I}^*$.

Although the combination of the *Mapper* and the *SUT* is in general nondeterministic, a well-designed *Mapper* will mask this nondeterminism so that the *Learner* perceives a deterministic Mealy machine, in the sense that a produced abstract output symbol is uniquely determined by the preceding sequence of abstract input symbols. We formalize this by defining $\mathcal{A}$ to be *adequate* for $\mathcal{SM}$ if $abstr_I(r_0, ua) = abstr_I(r_0, u'a')$ implies $abstr_O(\delta^{\mathcal{R}}(r_0, ua), \lambda(\delta(q_0, u), a)) = abstr_O(\delta^{\mathcal{R}}(r_0, u'a'), \lambda(\delta(q_0, u'), a'))$ for all input strings u, u' and symbols a, a'.

If $\mathcal{A}$ is adequate for $\mathcal{SM}$, then the *Learner* will perceive that the combination of the *Mapper* and the *SUT* is equivalent to a (deterministic) Mealy machine (which may or may not be finite-state). This deterministic Mealy machine can be defined by a (Nerode-like) quotient construction, as follows. Define the equivalence $\simeq$ on $Q^{\langle \mathcal{SM}, \mathcal{A} \rangle}$ by $\langle q, r \rangle \simeq \langle q', r' \rangle$ if for any input strings $u, u' \in {\Sigma_I}^*$ and input symbols $a, a' \in \Sigma_I$ we have that $abstr_I(r, ua) = abstr_I(r', u'a')$ implies $abstr_O(\delta^{\mathcal{R}}(r, ua), \lambda(\delta(q, u), a)) = abstr_O(\delta^{\mathcal{R}}(r', u'a'), \lambda(\delta(q', u'), a'))$. Intuitively, two elements of $Q^{\langle \mathcal{SM}, \mathcal{A} \rangle}$ are equivalent if they cannot be distinguished by the *Learner*, i.e., any two subsequent input strings that are identified by $abstr_I$ trigger two subsequent output strings that are identified by $abstr_O$. If $\mathcal{A}$ is adequate for $\mathcal{SM}$, define $\mathcal{A}\langle\langle \mathcal{SM} \rangle\rangle$ to be the MM $\langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\langle \mathcal{SM}, \mathcal{A} \rangle} / \simeq, [\langle q_0, r_0 \rangle]_{\simeq}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}} \rangle$, where for any $a \in \Sigma_I$ with $abstr_I(r, a) = a^{\mathcal{A}}$ we have

- $\delta^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}}) = [\langle \delta(q, a), \delta^{\mathcal{R}}(r, a) \rangle]_{\simeq}$, and
- $\lambda^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}}) = abstr_O(\delta^{\mathcal{R}}(r, a), \lambda(q, a))$.

For any $a^{\mathcal{A}} \in \Sigma_I^{\mathcal{A}}$ for which there is no $a \in \Sigma_I$ with $abstr_I(r, a) = a^{\mathcal{A}}$, we let $\lambda^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}})$ be a designated error symbol, and let $\delta^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}})$ be

a designated error state with a self-loop from which only the error symbol is output. The definition of $\simeq$ can be used to show that $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is well-defined.

If a finite Mealy machine $\mathcal{M}^{\mathcal{A}} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}}\rangle$ is produced by the *Learner*, then we must finally "reverse" the effect of the abstraction $\mathcal{A}$ to obtain an SMM $\mathcal{SM}$ such that $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$. In general, we then run into the problem that an abstract output symbol may correspond to several concrete output symbols, implying that there is not a unique deterministic SMM that causes the *Learner* to produce $\mathcal{M}^{\mathcal{A}}$. Therefore, define $\mathcal{A}$ to be *unambiguous* for $\mathcal{M}^{\mathcal{A}}$ if for all input symbols a and all $\langle q^{\mathcal{A}}, r\rangle \in Q^{\langle\mathcal{SM},\mathcal{A}\rangle}$, there is at most one output symbol b which satisfies

$$abstr_O(\delta^{\mathcal{R}}(r,a),b) = \lambda^{\mathcal{A}}(q^{\mathcal{A}}, abstr_I(r,a))$$

Intuitively, this means that we can deduce which output symbol is produced by $\mathcal{SM}$ by seeing only its abstraction.

If $\mathcal{A}$ is unambiguous for $\mathcal{M}^{\mathcal{A}}$, then define $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$ to be the Mealy machine $\langle \Sigma_I, \Sigma_O, Q^{\mathcal{A}} \times R, \langle q_0^{\mathcal{A}}, r_0\rangle, \delta, \lambda\rangle$, where

- $\delta(\langle q^{\mathcal{A}}, r\rangle, a) = \langle \delta^{\mathcal{A}}(q^{\mathcal{A}}, abstr_I(r,a)), \delta^{\mathcal{R}}(r,a)\rangle$, and
- $\lambda(\langle q^{\mathcal{A}}, r\rangle, a) = b$, where b is such that $abstr_O(\delta^{\mathcal{R}}(r,a),b) = \lambda^{\mathcal{A}}(q^{\mathcal{A}}, abstr_I(r,a))$.

Proposition 1. *If $\mathcal{A}$ is adequate for $\mathcal{SM}$ and unambiguous for $\mathcal{M}^{\mathcal{A}}$, and if $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$, then $\mathcal{SM}$ is equivalent to $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$.* □

The equivalence can be proven by observing that a state $\langle q^{\mathcal{A}}, r\rangle$ of $\mathcal{A}^{-1}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to a state q of $\mathcal{SM}$ if there is a common input string $u \in \Sigma_I$ which drives the state of $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$ to $\langle q^{\mathcal{A}}, r\rangle$, and the state of $\mathcal{SM}$ to q.

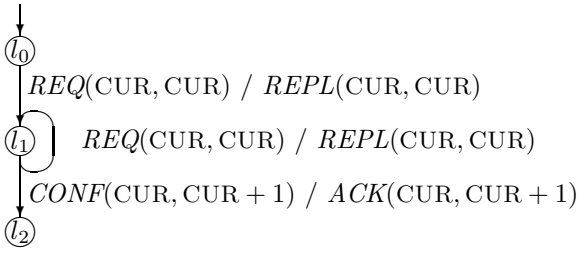
Example. Let us define an abstraction for the SMM in Figure 1. Since the infiniteness typically stems from the infinite domains of the parameters in symbols, the abstraction maps parameter values to small domains. We map each symbol form $REQ(id, sn)$ to an abstract symbol of form $REQ(\mathrm{ID}, \mathrm{SN})$, where $\mathrm{ID} \in \{\mathrm{CUR}, \mathrm{OTHER}\}$ and $\mathrm{SN} \in \{\mathrm{CUR}, \mathrm{CUR}+1, \mathrm{OTHER}\}$. The state of the abstraction is defined by two local variables that range over $\mathbb{N}$: *abs_id*, which is initially "undefined" (denoted $\perp$), and thereafter assigned to the *id* parameter of the first received *REQ* message, and *abs_sn*, which is also initially "undefined" and thereafter assigned to the *sn* parameter of the first received *REQ* message.

Table 1. Abstraction mappings for parameters

par	CUR	CUR + 1	OTHER
id	$cur\_id = \perp \wedge\ mtype = REQ$ $\vee$ $id = cur\_id \wedge\ cur\_id \neq \perp$		$id \neq cur\_id$ $\wedge\ cur\_id \neq \perp$
sn	$cur\_sn = \perp \wedge\ mtype = REQ$ $\vee$ $sn = cur\_sn \wedge\ cur\_sn \neq \perp$	$sn = cur\_sn + 1$ $\wedge\ cur\_sn \neq \perp$	$sn \neq cur\_sn \wedge sn \neq cur\_sn + 1$ $\wedge\ cur\_sn \neq \perp$

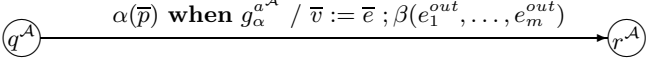
The input and output abstraction mappings $abstr_I$ and $abstr_O$ can be defined by supplying, for each parameter (being either *id* or *sn*) and each abstract parameter value D, a predicate which defines the set of parameter values that are mapped to D. We can organize these predicates into a table, as in Table 1. We use *mtype* to denote the action type of the symbol considered (being either *REQ*, *CONF*, *RESP*, or *ACK*). Thus, in total there are 12 abstract input symbols and 13 abstract output symbols (the symbol *REJ* is mapped to itself).

A possible result by the *Learner* is the Mealy machine $\mathcal{M}^{\mathcal{A}}$ in the figure below.



Each arc is labeled by an abstract input symbol followed by the abstract output symbol that the *Learner* observes in response. From the picture, we have excluded all arcs that contain the output symbol *REJ*: these all go to the terminal state l_2.

We can construct $\mathcal{A}^{-1}\langle\langle \mathcal{M}^{\mathcal{A}} \rangle\rangle$ from $\mathcal{M}^{\mathcal{A}}$, as an SMM, whose set of locations is $\{l_0, l_1, l_2\}$, whose state variables are the local variables of $\mathcal{A}$, and such that for each each transition $q^{\mathcal{A}} \xrightarrow{a^{\mathcal{A}}/b^{\mathcal{A}}} r^{\mathcal{A}}$ of $\mathcal{M}^{\mathcal{A}}$ there is a symbolic transition



where $g_\alpha^{a^{\mathcal{A}}}$ is the conjunction of constraints on parameter values $\overline{p}$ under which an input symbol $\alpha(\overline{p})$ is abstracted to $a^{\mathcal{A}}$, where $\overline{v} := \overline{e}$ is the update of local variables in the abstraction $\mathcal{A}$, and where $\beta(e_1^{out}, \ldots, e_m^{out})$ is composed from the expressions that cause an output symbol of form $\beta(d_1, \ldots, d_m)$ to be abstracted to $b^{\mathcal{A}}$.

When carrying this out on the finite Mealy machine $\mathcal{M}^{\mathcal{A}}$ obtained by the *Learner*, we obtain the SMM of Figure 1, but with location l_2 merged with the terminal error location.

3.3 Systematic Construction of Abstractions

The construction of a suitable abstraction is crucial for successful inference of an SMM $\mathcal{SM}$. In this subsection, we discuss techniques by which an abstraction can be constructed more systematically. We assume, as before, that the sets I and O of input and output actions of $\mathcal{SM}$, together with their arities, are known *a priori*. In the running example in the previous subsection, we see that typically the abstraction mapping for input symbols uses expressions that become guards in the resulting SMM, and that the abstraction mapping for output symbol uses

expressions that occur in output expressions of the SMM. We therefore assume that a set of guards and expressions, which is sufficient to construct a model of $\mathcal{SM}$, is also known *a priori*. This set can be seen as describing how state variables of $\mathcal{SM}$ can influence control flow through guards, and how they can be used in expressions that produce output symbols. We assume that the updates of state variables in $\mathcal{SM}$ do not need operators, i.e., they simply save some of the data values received in input parameters; operators that occur in updates to state variables can often be moved ("inlined") to the expressions in guards and output symbols where these state variables are used.

Under the above assumptions, we can construct an abstraction which maps combinations of parameterized input actions and guards in a possible SMM to abstract input symbols, and maps combinations of expressions in output symbols of a possible SMM to abstract output symbols, as in the running example. The updates to state variables will simply consist in assigning some input parameters to state variables: the problem here is to decide which input parameters will influence the future behavior of $\mathcal{SM}$, and must be remembered in state variables. In our experiments, we have made this decision based on observing the response of $\mathcal{SM}$ to selected input strings, i.e., by posing membership queries, and saving those parameter values that are used to produce future output. For parameter values on which the only performed operation is a test for equality, such as the *id* parameter of the running example, we have made these ideas more precise in our earlier work [6], as follows:

Consider an input string u, which contains a parameter value d. We observe the output of $\mathcal{M}$ in response to u and to selected continuations of u, and decide to store d in a state variable if there is some continuation v of u such that d is used to produce the response to v. More precisely, this happens if there is a fresh (i.e., previously unused) data value d' such that the response $\lambda(\delta(q_0, u), v)$ to v and the response $\lambda(\delta(q_0, u), v[d'/d])$ to $v[d'/d]$ (i.e., v where all occurrences of d have been replaced by d') satisfy $\lambda(\delta(q_0, u), v)[d'/d] \neq \lambda(\delta(q_0, u), v[d'/d])$, i.e., $\mathcal{SM}$ does not treat d in the same way as a fresh (previously unused) value d'. This happens, e.g., if $\lambda(\delta(q_0, u), v[d'/d])$ contains the data value d implying that d must have been remembered before seeing the subsequent input $v[d'/d]$, and that d should be stored in a state variable.

4 Experiments

We have implemented and applied our approach to infer models of two implemented standard protocols: the Session Initiation Protocol (SIP) and the Transmission Control Protocol (TCP). Due to space restrictions we were not able to include the TCP case study in this paper, but the methodology for inferring TCP is similar to SIP. In this section, we first describe our experimental setup, thereafter its application to the protocol. In order to have access to a large number of standard communication protocols, for evaluation of inference techniques, we use the protocol simulator ns-2[1], which provides implementations of many

[1] http:/www.isi.edu/nsnam/ns/

protocols, to serve as SMM Under Test (SUT). Messages are represented as C++ structures, saving us the trouble of parsing messages represented as bitstrings. As *Learner*, we use the LearnLib tool [30], developed at the Technical University Dortmund, which has an efficient implementation of the L^* algorithm that can construct both finite automata and Mealy machines. LearnLib provides several different realizations of equivalence queries, including random test suites of user-controlled size.

SIP. SIP is an application layer protocol for creating and managing multimedia communication sessions. Although a lot of documentation is available, such as the `RFC 3261`, no proper reference model, as a state machine, is available. We aimed to infer the behavior of the SIP Server entity when setting up connections with a SIP Client. We represent input messages from the SIP Client to the SIP Server as *Method*(*From*, *To*, *Contact*, *CallId*, *CSeq*, *Via*), where

- *Method* defines the type of request, either INVITE, PRACK, or ACK,
- *From* and *To* are addresses of the originator and receiver of the request,
- *CallId* is a unique session identifier,
- *CSeq* is a sequence number that orders transactions in a session,
- *Contact* is the address where the Client wants to receive input messages, and
- *Via* indicates the transport path that is used for the transaction.

We represent output messages from the SIP Server to the SIP Client as *StatusCode*(*From*, *To*, *CallId*, *CSeq*, *Contact*, *Via*), where *StatusCode* is a three digit status code that indicates the outcome of a previous request from the Client, and the other parameters are as for a input message.

Abstraction Mapping. We have constructed an abstraction mapping for the SIP server, which maps each parameter to an abstract value. The parameters *From*, *To*, and *Contact* must be pre-configured in a session with ns-2, so they are set to constant values throughout the experiment. The *Via* parameter is a pair, consisting of a default address and a variable branch. The parameters *Via*, *CallId*, and *CSeq* are potentially interesting parameters. A priori, they can be handled as parameters from a large domain, on which test for equality and potentially incrementation can be performed. Monitoring of membership queries, as described in Section 3.3 reveals that for each of these parameters, the ns-2 SIP implementation remembers the value which is received in the first *Invite* message (presumably, it is interpreted as parameters of the connection that is being established). The implementation also remembers the value received in the most recent input message when producing the corresponding reply, but thereafter forgets it. We therefore equip the abstraction with six state variables. The state variable *firstId* stores the *CallId* parameter of the first *Invite* message, and *lastId* stores the *CallId* parameter value of the most recently received message. The state variables *firstCSeq* and *lastCSeq* store the analogous values for the *CSeq* parameter, and the state variables *firstVia* and *lastVia* for the *Via* parameter.

The abstraction mapping for input symbols is shown in Table 2. Intuitively, the input parameter *CallId* is compared with the variable *firstId* (assigned at the

Table 2. Mapping table for input messages of SIP Server

par	FIRST	LAST	ANY
$CSeq$			$isInteger(CSeq)$
Via			$Via.Address = Default$ $\wedge isInteger(Via.Branch)$
$CallId$	$firstId = \bot \wedge mtype = Invite$ $\vee firstId \neq \bot \wedge CallId = firstId$	$otherwise$	

Table 3. Mapping table for output messages of SIP Server

par	FIRST	LAST	OTHER
$CSeq$	$CSeq = firstCSeq$	$CSeq = lastCSeq$	$Other$
Via	$Via = firstVia$	$Via = lastVia$	$Other$
$CallId$	$CallId = firstId$	$CallId = lastId$	$Other$

occurrence of the first *Invite* message) to check if it should be mapped to FIRST or LAST. For the input parameters *Via* and *Cseq*, we merged the abstract values FIRST and LAST into the single value ANY, since we found that these input parameters are not tested by ns-2: we could also have followed the methodology of Section 3.3 and kept these two values separate. In output messages, for which the mapping is shown in Table 3, these three parameters can take the value received in the first *Invite* message, or the value in the just received message, corresponding to the two abstract values FIRST and LAST.

The SIP Server does not always respond to each input message, and sometimes responds with more than one message. To stay within the Mealy machine formalism, we introduce the *nil* input symbol which denotes the absense of input, in order to allow sequences of outputs, and the *timeout* output symbol, denoting the absence of output. This could be made more systematic by using techniques in [1].

Results. The inference performed by LearnLib needed about one thousand membership queries and one equivalence query, and resulted in an abstract model with 10 locations and 70 transitions. For presentation purposes, we have pruned the model as follows: (1) removing transitions triggered by abstract symbols that have no corresponding concrete symbol: the Mapper will immediately reject these, and react with a distinguished error symbol, (2) removing transitions with empty input and output symbol, i.e., with labels *nil/timeout*, (3) removing locations which have become unreachable after the previous steps. In Figure 3, we show the resulting abstract model with 9 locations and 48 transitions. For readability, some transitions with same source location, output symbol and next location (but with different input symbols) are merged: the original input method types are listed, separated by a bar (|). Due to space limitations, we have suppressed the (abstract) parameter values. However, the *CallId* parameter of the input messages with abstract value FIRST, is depicted in the model with solid transition lines, the remaining transitions have a dashed line pattern. We suppressed all other parameters in the figure. A full abstract model, showing the abstract values of other output parameters can be found at `http://www.it.uu.se/research/group/testing/sip`, together with a description of the corresponding concrete model.

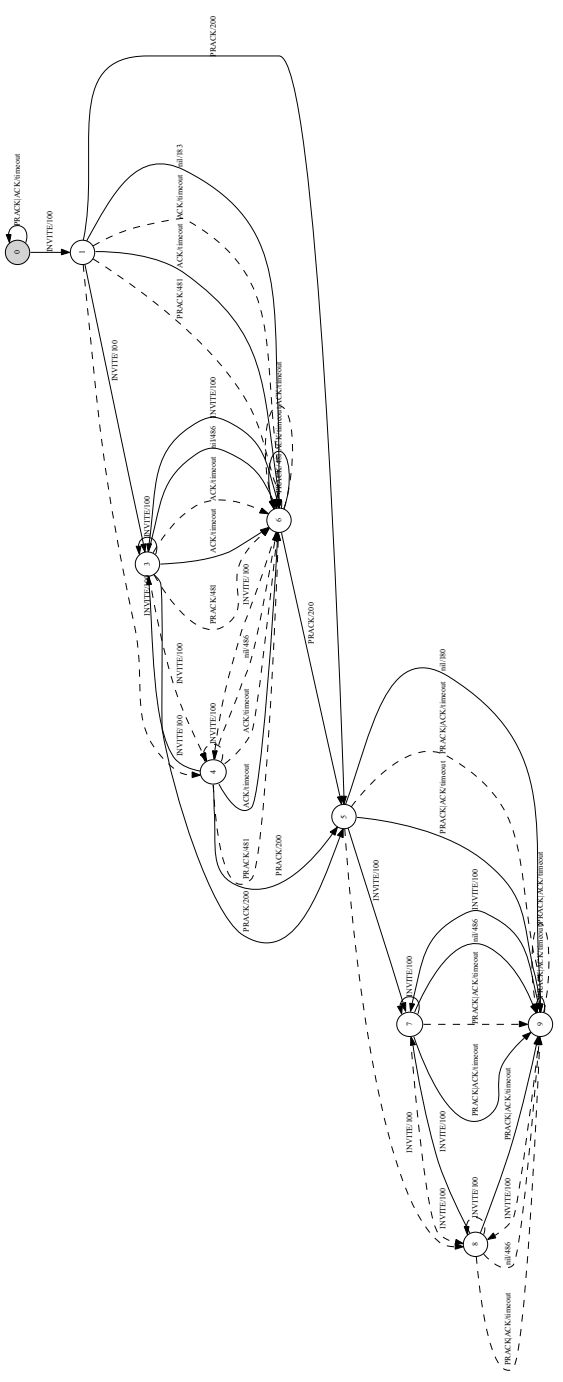


Fig. 3. Full SIP model

5 Conclusions and Future Work

We have presented an approach to infer models of entities in communication protocols, which also handles message parameters. The approach adapts abstraction, as used in formal verification, to the black-box inference setting. This necessitates to define an abstraction together with the local state needed to define it. This makes finding suitable abstractions more challenging, but we have presented techniques for systematically deriving abstractions under restrictions on what operations the component may perform on data. We have shown the feasibility of the approach towards inference of realistic communication protocols, by a feasibility studies on the SIP, as implemented in the protocol simulator ns-2. Our work shows how regular inference can infer the influence of data parameters on control flow, and how data parameters are produced. Thus, models generated using our extension are more useful for thorough model-based test generation, than are finite-state models where data aspects are suppressed. In future work, we plan to supply a library of different inference techniques specialized towards different data domains that are commonly used in communication protocols.

Acknowledgement. We are grateful to Falk Howar from TU Dortmund for his generous LearnLib support, to Falk Howar and Bernhard Steffen for fruitful discussions, and to Frits Vaandrager for many indispensable comments.

References

1. Aarts, F., Vaandrager, F.: Learning I/O automata. In: Gastin, P. (ed.) CONCUR 2010. LNCS, vol. 6269, pp. 71–85. Springer, Heidelberg (2010)
2. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: Proc. 29th ACM Symp. on Principles of Programming Languages, pp. 4–16 (2002)
3. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
4. Ball, T., Rajamani, S.: The SLAM project: Debugging system software via static analysis. In: Proc. 29th ACM POPL, pp. 1–3 (2002)
5. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines with parameters. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 107–121. Springer, Heidelberg (2006)
6. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines using domains with equality tests. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 317–331. Springer, Heidelberg (2008)
7. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
8. Brun, Y., Ernst, M.: Finding latent code errors via machine learning over program executions. In: ICSE 2004, pp. 480–490 (May 2004)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM 50(5), 752–794 (2003)
10. Cobleigh, J., Giannakopoulou, D., Pasareanu, C.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)

11. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
12. Grinchtein, O.: Learning of Timed Systems. PhD thesis, Dept. of IT, Uppsala University, Sweden (2008)
13. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 379–396. Springer, Heidelberg (2004)
14. Grinchtein, O., Jonsson, B., Leucker, M.: Inference of timed transition systems. Electr. Notes Theor. Comput. Sci. 138(3), 87–99 (2005)
15. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of event-recording automata using timed decision trees. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 435–449. Springer, Heidelberg (2006)
16. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 357–370. Springer, Heidelberg (2002)
17. Groz, R., Li, K., Petrenko, A., Shahbaz, M.: Modular system verification by inference, testing and reachability analysis. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 216–233. Springer, Heidelberg (2008)
18. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002)
19. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc.29th ACM Symp. on Principles of Programming Languages, pp. 58–70 (2002)
20. Huima, A.: Implementing conformiq qtronic. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 1–12. Springer, Heidelberg (2007)
21. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 315–327. Springer, Heidelberg (2003)
22. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
23. Li, K., Groz, R., Shahbaz, M.: Integration testing of distributed components based on learning parameterized I/O models. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 436–450. Springer, Heidelberg (2006)
24. Loiseaux, C., Graf, S., Sifakis, J., Boujjani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design 6(1), 11–44 (1995)
25. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proc. ICS 2008, pp. 501–510 (2008)
26. Mariani, L., Pezzé, M.: Dynamic detection of COTS components incompatibility. IEEE Software 24(5), 76–85 (2007)
27. Niese, O.: An integrated approach to testing complex systems. Technical report, Dortmund University, Doctoral thesis (2003)
28. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: FORTE/PSTV 1999, Beijing, China, pp. 225–240. Kluwer, Dordrecht (1999)
29. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. IEEE Trans. on Software Engineering 30(1), 29–42 (2004)
30. Raffelt, H., Steffen, B., Berg, T.: Learnlib: a library for automata learning and experimentation. In: FMICS 2005, New York, NY, USA, pp. 62–71 (2005)

31. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Information and Computation 103, 299–347 (1993)
32. Shahbaz, M., Li, K., Groz, R.: Learning and integration of parameterized components through testing. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 319–334. Springer, Heidelberg (2007)
33. Shu, G., Lee, D.: Testing security properties of protocol implementations - a machine learning based approach. In: Proc. ICDCS 2007. IEEE, Los Alamitos (2007)

Practical End-to-End Performance Testing Tool for High Speed 3G-Based Networks

Hiroyuki Shinbo[1], Atsushi Tagami[1], Shigehiro Ano[1], Toru Hasegawa[1], and Kenji Suzuki[2]

[1] KDDI R&D Laboratories Inc., 2-1-15 Ohara, Fujimino-shi, Saitama 356-8502, Japan
{shinbo,tagami,ano,hasegawa}@kddilabs.jp
[2] The University of Electro-Communications, 1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585, Japan
suzuki@cs.uec.ac.jp

Abstract. High speed IP communication is a killer application for 3rd generation (3G) mobile systems. Thus 3G network operators should perform extensive tests to check whether expected end-to-end performances are provided to customers under various environments. An important objective of such tests is to check whether network nodes fulfill requirements to durations of processing packets because a long duration of such processing causes performance degradation. This requires testers (persons who do tests) to precisely know how long a packet is hold by various network nodes. Without any tool's help, this task is time-consuming and error prone. Thus we propose a multi-point packet header analysis tool which extracts and records packet headers with synchronized timestamps at multiple observation points. Such recorded packet headers enable testers to calculate such holding durations. The notable feature of this tool is that it is implemented on off-the shelf hardware platforms, i.e., lap-top personal computers. The key challenges of the implementation are precise clock synchronization without any special hardware and a sophisticated header extraction algorithm without any drop.

Keywords: End-to-end Performance Tests, Clock Synchronization Protocol, Packet Header Analysis.

1 Introduction

Providing high performances in the 3rd generation (3G) mobile systems which have begun being deployed is important for 3G mobile system operators. The 1xEV-DO (1x Evolution Data Only) system based on the specification of cdma2000 High Rate Packet Data [1] is an example of 3G mobile system and it uses a special data link protocol [1,2]. The 3G mobile system operators (in the rest of paper, we call them just 3G operators) extensively perform interoperability tests [3], where end-to-end performances between a mobile terminal and a server are measured with various environments. In order to detect latent causes of performance degradations before a commercial service begins, testers

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 205–220, 2010.

(persons who do tests) of the 3G operators usually do the two tasks. First, if performance degradation is observed at the tests, they identify which software at which network node is the cause of degradation. Second, testers check whether network nodes process a packet within the predefined duration or not. This testing is especially important for network nodes on the backbone because they should process it in a small duration, e.g., hundreds of micro-seconds, to provide high end-to-end performances.

Since network nodes' implementations are black-boxes to the 3G operators, the testers should capture packets at incoming/outgoing links of each network node with timestamps and then calculate how long each packet is hold at the network node by comparing the timestamps. However, this style of tests is very time-consuming because such calculations should be manually performed for all packets at all network nodes.

In order to automate such testers' tasks, we propose a multi-point packet header analysis tool which consists of IP packet header capture devices and the manager. An IP packet header capture device captures IP packets from a tapped link, extracts only IP packet headers and records them to its disk with timestamps. The manager does calculations which the testers manually do by controlling all the IP packet header capture devices.

The main goal is that it is implemented on an off-the-shelf hardware platform. Actually we use a lap-top PC (Personal Computer) with a crystal oscillator, network interface cards and a RAM (Random Access Memory) disk instead of using expensive commercial packet tester such as IXIA [4]. This style of implementation contributes not only to reducing costs of testing, but also to increasing chances of using such tools in various environments.

However, implementing it as user-space software on (lap-top) PCs is not trivial. There are two key challenges for the implementation. The first and the most difficult challenge is clock synchronization with scores of micro-second level precision. Clocks, of which values are used as timestamps, of all PCs should be synchronized to calculate how long an IP packet is hold by a network node. Since a maximum holding duration of some network node is less than several hundred milliseconds, i.e., about 400 micro-seconds, the maximum error among the clocks should be less than scores of micro-seconds. This precision is not easy to achieve without special hardware.

The second challenge is recording all IP packet headers to a RAM disk without any drop. Since an IP packet sent by either a server or a mobile terminal is encapsulated by PPP (Point-to-Point Protocol) and segmented by RLP (Radio Link Protocol) [1], the boundaries of captured packets do not always correspond to IP packet boundaries. Although the straight-forward way is to capture all packets in order to bridge the gap, it is difficult for PCs to record a number of 30 or 50 byte long packets segmented by RLP to its RAM disk without any drop.

In this paper, we have developed a multi-point packet header analysis tool which is useful for end-to-end performance tests over 3G mobile systems. The contributions of are twofold: a simple and precise clock synchronization protocol

and a real-time IP packet header extraction algorithm. First, taking advantage of the fact that clock synchronization need to be maintained only for a test duration which would be less than scores of minutes, we adopt a post-processing approach where timestamps are synchronized after all IP packet headers are records. This achieves simple, but precise clock synchronization with the maximum error of scores of micro-seconds. Second, we design a real-time IP header packet extraction algorithm specialized for 3G mobile systems by reading as small number of bytes from captured packets as possible. This algorithm enables to record all IP packet headers to a RAM disk without any drop. These sophisticated features make this tool so useful that it was used for commercial 3G packet services, such as 1xEV-DO and BCMCS (Broadcast-Multicast Service) system [5,6]. In the tests with this tool, we detected more than 10 software bugs of network nodes, which were not detected by their vendors.

The rest of this paper is organized as follows: Section 2 describes the overview of 1xEV-DO system. Section 3 describes the overview of the multi-point packet header analysis tool. Section 4 and 5 describe the clock synchronization protocol and the IP packet header extraction algorithm, respectively. Section 6 describes how the tool is used at the actual tests. Section 7 describes related work.

2 Overview of 1xEV-DO System

Fig.1 shows network nodes and protocols of the 1xEV-DO system. The 1xEV-DO system provides a high speed IP communication between a mobile access terminal (called *AT* in Fig.1) and a server. We define that *Forward-link* is the direction from a server to an AT, and *Reverse-link* is the direction from an AT to a server. Each network node takes the following role:

- A server is located at the Internet.
- An HA (Home Agent) provides handovers between PDSNs according to the Mobile IP (MIP).
- A PDSN (Packet Data Serving Node) is used for authenticating/accounting ATs. It provides an endpoint of a PPP session between a PDSN and an AT.
- A PCF (Packet Control Function) / ANC (Access Network Controller) segments PPP frames to RLP (Radio Link Protocol) packets and reassembles RLP packets. RLP provides a reliable packet transfer using packet retransmissions.
- An ANTS (Access Network Transceiver System) transmits radio wave.
- An AT (Access Terminal) is a mobile access terminal.

In the 1xEV-DO system, PPP is used to encapsulate IP packets sent by either an AT or a server. We call these IP packets as end-to-end IP packets. PPP frames are segmented to 30 or 50 byte RLP packets between an AT and a PCF/ANC.

Fig.2 shows how end-to-end IP packets are segmented and reassembled by network nodes and how capsule-headers are used on the Reverse-link of the 1xEV-DO system. The packets are done so on the Forward-link. Encapsulation, segmentation and reassembly are performed as follows:

1. An application on an AT makes an end-to-end IP packet. The AT adds a PPP header to it and segments it to RLP packets.
2. The AT adds the RLP header to each segmented packet (S1 to S3), and transmits them to an ANTS at a radio link.
3. After each segmented packet is received, the ANTS adds a UDP/IP header and transmits it to a PCF/ANC.
4. The PCF/ANC removes the UDP/IP header and the RLP header. Then, the GRE/IP header is added to the segmented packets, and transmits the packets to the PDSN.
5. The segmented packets are reassembled at a PDSN. The PDSN also checks whether the IP packet is correctly reassembled or not by a CRC of the PPP header.
6. The PDSN adds a MIP header to the IP Packet and transmits it to a HA.
7. The HA removes the MIP Header and transmits the IP packet to a server.

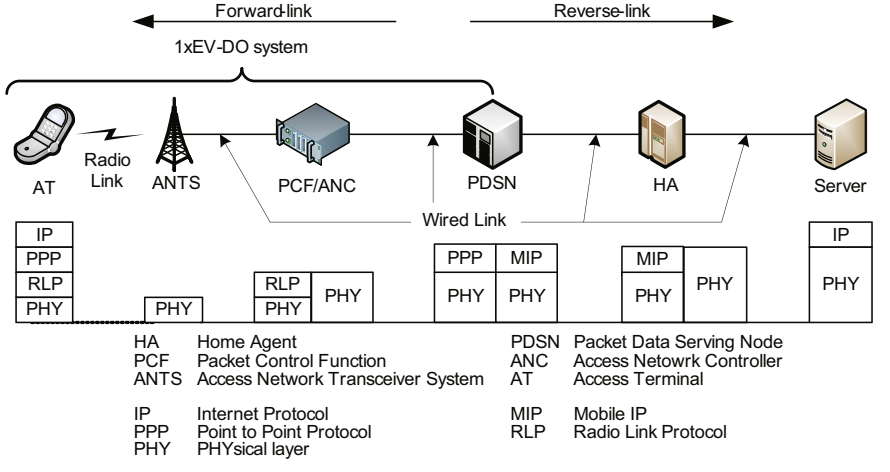


Fig. 1. Network Nodes and Protocols in 1xEV-DO system

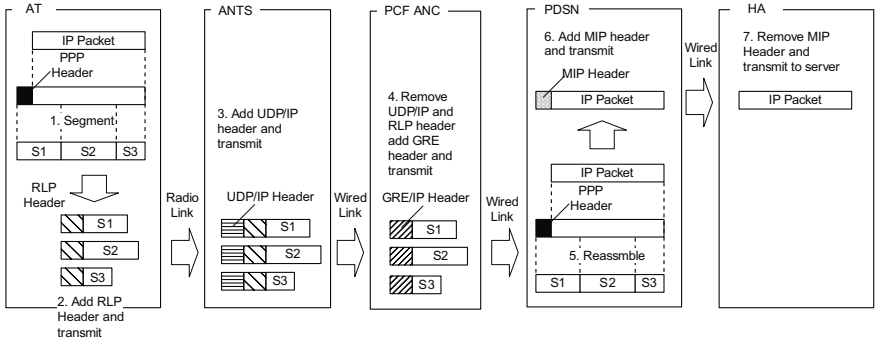


Fig. 2. Overview of segmenting and reassembling and used headers

3 Multi-point Packet Header Analysis Tool

3.1 Overview

The structure of multi-point packet header analysis tool is illustrated in Fig.3. It consists of *IP packet header capture devices* and the *manager*. In the rest of paper, we call the multi-point packet header analysis tool as the *analysis tool*, and an IP packet header capture device as a *capture device*. We implement them as the user-space software running a PC with a crystal oscillator, network interface cards and RAM disk.

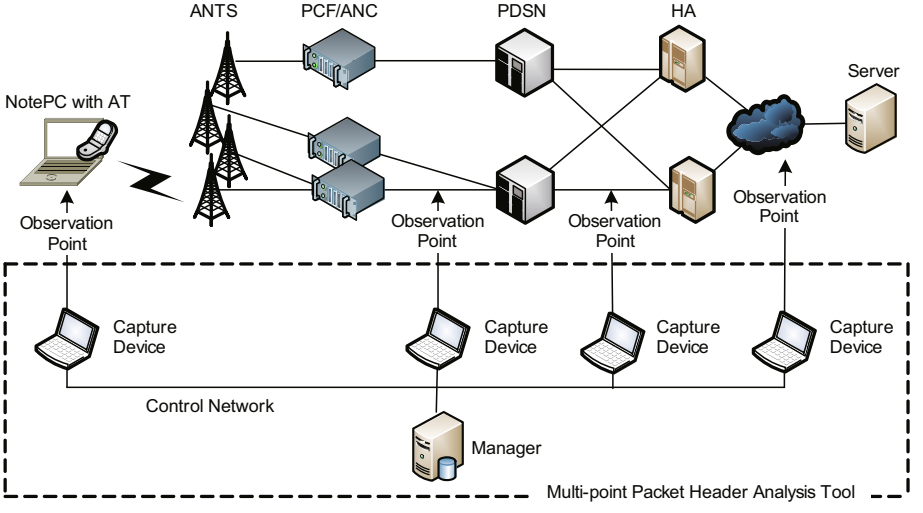


Fig. 3. Multi-point Packet Header Analysis Tool

Each capture device is a lap-top PC with two network interface cards. One network interface card is used to tap a link and the other is used to communicate with the manager. Capture devices are set to tap links between all pairs of network nodes including a server and an AT. We call tapped links as observation points. A capture device extracts an end-to-end IP packet header from a captured packet and records it into its RAM disk with a timestamp when the packet is captured. On the contrary, the manager controls capture devices and analyzes all captured IP packet headers collected from them.

3.2 How Analysis Tool Is Used?

How this tool is used to analyze a communication flow between the AT and the server is as follows:

Preparation for Clock Synchronization
Before the AT and the server start communicating, the manager broadcasts a packet for synchronizing the capture devices' clocks to that of one the capture

devices. All capture devices are connected via a broadcast medium such as Ethernet.

Packet Header Extraction
A tester makes all capture devices start capturing packets from tapped links. When a packet is captured, each capture device extracts an end-to-end IP packet header and then records it to its local RAM disk with its timestamp. The timestamp is read from the clock of PC which is a crystal oscillator. How the real-time IP packet header extraction algorithm handles protocol headers of captured packets will be described in Section 5.

Packet Holding Duration Analysis
After the test communication finishes, the manger collects all headers from all the capture devices. The timestamps set by individual capture devices are synchronized to the clock of one of the capture device. In other words, all timestamps are re-written so as to be synchronized to such device's clock. How clocks are synchronized will be described in Section 4. Then by analyzing timestamps for the same end-to-end IP packet at difference observation points, a tester calculates how long each end-to-end packet is hold by each network node.

4 Clock Synchronization Protocol

4.1 Problem Statement

The required precision should be around scores of micro-seconds. This is because the PDSN and the HA, i.e., backbone network nodes, handle a number of communication flows between ATs and servers. Usually, these network nodes should process each end-to-end IP packet within less than hundreds of micro-seconds. In some system, the average duration of processing a packet is about 400 micro-seconds as described in Section 6. Thus the goal of the maximum error is within scores of micro-seconds.

In order to precisely explain our clock synchronization protocol, we define several terms. A clock is a hardware register of a PC and its value is created from its crystal oscillator. A clock value is how many the crystal oscillator ticks and corresponds to the elapsed time. We use a time value and a timestamp interchangeably as a clock value. When a clock value is set to a captured packet, we call it is a timestamp. A clock frequency is how many it ticks in one second. Clock synchronization or synchronizing clocks means that difference between clock values of different capture devices are less than some threshold. For example, if the difference is always less than 100 micro-seconds, clocks are said to be 100 micro-seconds precise or the maximum error is within 100 micro-seconds.

4.2 Hurdles to Prevent Clock Synchronization

The goal of clock synchronization protocol is to synchronize all clocks of capture devices to a selected capture device, which we call it the *master capture device*,

within scores of micro-second error. The protocol consists of the two procedures. First, the manager broadcast a packet informing all capture devices of clock values being set to 0 (initialization). During the test, each capture device reads its clock value and sets it as a timestamp to a captured IP packet header. Second, after the test, timestamps at different capture devices are compensated so that the clocks are synchronized.

There are two factors which prevent the clocks from being synchronized.

Long-term and Short-term Errors of Crystal Oscillators
A crystal oscillator is not either accurate or stable. Each oscillator's frequency is different from the ideal frequency. The difference is called the *frequency error* and that in most PCs is accurate to one part in 10^4 to 10^6. Besides, its frequency changes due to environmental factors such as variations in temperature and supply voltage. Due to frequency errors between two devices, the clock values are gradually drifting as shown in Fig.4. The x-axis and y-axis show the ideal time and the time observed at the ideal clock or the actual clock. The ideal clock means a clock without any frequency error. The dotted line shows how the ideal clock advances, and the angle is 45 degree. The observed time values of the actual clock are plotted and they are interpolated to the line using the least squares method. The angle difference between the two lines corresponds to the average frequency error. Since this angle or the difference is stable for a long duration, we call it a long-term error. On the contrary, the plots are not always on the interpolated line. A short term error is a difference between the observed time value of the actual clock and the dotted line.

Non-deterministic Delay of Initialization Packet Transfer
At the initialization, the manger broadcast a packet for setting all clock values to 0. However, the packet does not always reach capture devices at precisely the same time. In other words, delay of sending the packet from the manager to each capture is not deterministic due to several non-deterministic delays: the

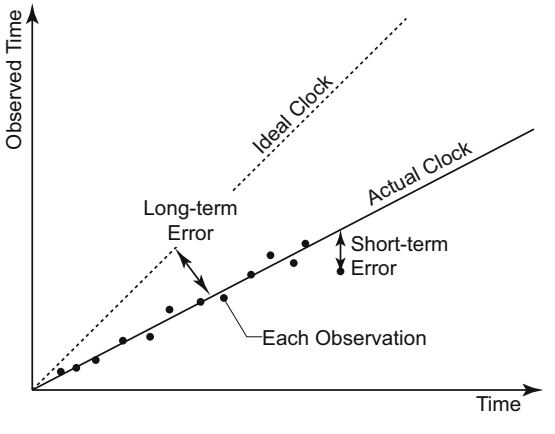


Fig. 4. How Ideal and Actual Clocks Advance

time spent by the manager, the delay incurred waiting for access to a physical interface card, the time needed for a packet from the manager to a capture device, and the time required for the capture device's network interface card to receive and notify the user space software of its arrival. Due to such non-deterministic delays, the ideal time of capture device is different from that of the master capture device.

4.3 Design Principles

We set out the two design principles to compensate for the two factors.

Compensating for Clock's Long-term Error
We assume that the duration of each test is less than 20 minutes. We choose 20 minutes because it is recommended that TCP performance tests should continue more than 15 minutes. Due to such a short duration, we can assume that a hardware clock is stable for the duration and that we can ignore the short term error. By assuming that all clocks are stable, we can easily synchronize the clocks by compensating for the angle difference from the capture device to the master one.

Compensating for Non-deterministic Delay with Broadcast Communication
We compensate for non-deterministic delays using the broadcast media. Capture devices, which are usually located at an in-house test-bed, are directly connected each other via a broadcast medium so that a packet sent for synchronizing clocks are received by all the capture devices at almost the same time.

4.4 Preliminary Measurements of Long-term and Short-term Errors

How the design principles work well depends on how long-term and short-term errors of actual crystal oscillators are, how stable they are and how the packet delays of the actual broadcast medium are. Thus we have measured them in the following experimental conditions:

- A packet tester, e.g., *IXIA 400* [4], is used to broadcast a 40 byte long test packet every second for 20 minutes. The clock's precision of the packet tester is 1 PPM (Part Par Million).
- A shared Ethernet hub is used as a broadcast medium.
- Two PCs, which run Linux (kernel 2.4.18), are set at the broadcast medium and capture the test packets using *tcpdump* software [7]. Tcpdump runs in the user space and sets a timestamp to every captured packet.
- This 20 minutes experiment was performed 10 times and thus 20 measurement results for a clock were totally obtained.

Clock's Long-term and Short-term Errors
Two typical measurement results are shown in Fig.5. The x-axis is the clock value (the time value) at the packet tester and the y-axis is the error from the clock

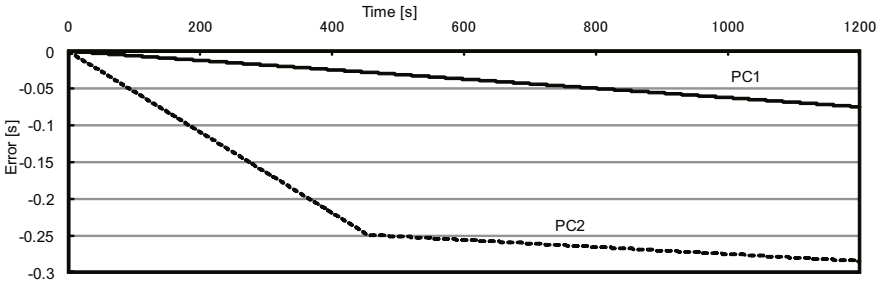


Fig. 5. Timestamps at the Two PCs

value at the PC (*PC1* or *PC2*) from that of the packet tester. We interpolate all the errors from the clock values to the lines with the least squares method. Among the 20 measurement results, 19 measurements show that the clock was stable for the 20 minutes. This result is shown by the line with caption *PC1* in Fig.5. All the errors are almost on the interpolated line. On the contrary, at one measurement, the clock was not stable, i.e., the clock frequency changes during the 20 minutes. This result is shown by the line with caption *PC2* in Fig.5. The angle of interpolated line of *PC2* changes when 445 seconds elapses at the packet tester. Since the error of *PC2* increases to about 300 micro-seconds after 20 minutes elapse, it is difficult to make the error at any time less than 100 micro-seconds only by compensating for the angles of the interpolated lines.

Due to this, we adopt an approach that results of a test during when such clock unstableness is detected are thrown away and the test is retried.

Besides, we calculate all the short-time errors, i.e., the distances from all the measured clock values to the interpolated lines. The maximum short-time error is about 20 micro-seconds. Since the long-duration error would become almost 0 by compensating for the angle differences, this 20 micro-seconds level error is not a problem to synchronize clocks within the maximum error of scores of micro-seconds.

Non-deterministic Delay

We cannot correctly measure how long the difference between the delays from the packet tester to each PC is. However, the above maximum 20 micro-seconds error includes both the short-term clock error and the above difference. Thus it is estimated that the maximum difference is less than 20 micro-seconds. It means that the differences between start times at different PCs are less than 20 micro-seconds. As the result, the error of clocks would be less than twice as 20 micro-seconds, i.e., 40 micro-seconds during 20 minutes.

4.5 Protocol Details and Performance

This subsection describes the details of the clock synchronization protocol. It is assumed that there are n capture devices numbered from 0 to $n-1$ and the master capture device is 0.

1. The manager broadcasts m broadcast packets $P_s = \{P_{s1}, P_{s2}, \ldots, P_{sm}\}$ to all capture devices at equal intervals via the broadcast medium. Each capture device k records the clock values when it received P_s as $t_s(k) = \{t_{s1}(k), t_{s2}(k), \ldots, t_{sm}(k)\}$.
2. When the test starts, the capture device k reads its clock value and records it as a timestamp of captured IP packet header. The timestamp of the i th IP packet header is denoted as $t_k(i)$.
3. After the test finished, the manager broadcasts m broadcast packet $P_e = \{P_{e1}, P_{e2}, \ldots, P_{em}\}$ to all capture devices at equal intervals via the broadcast medium. Each capture device k records the clock values when it received P_e as $t_e(k) = \{t_{e1}(k), t_{e2}(k), \ldots, t_{em}(k)\}$.
4. The manager calculates angles of the clock values before and after the test, i.e., $t_s(k)$ and $t_e(k)$, using the least squares method. If the difference between the calculated angles is larger than the predefined threshold th such that $Angle(t_s(k)) - Angle(t_e(k)) \geq th$, the frequency error of capture device i is larger than the predefined threshold. The clock of capture device k cannot be synchronized to the master capture device 0. The clock synchronization protocol stops. (As the result, the test result would be thrown away and this test would be re-tried again.)
5. The manager rewrites timestamp $t_k(i)$ to $\overline{t_k}(i)$ according to Equation(1), and $t_{s1}(0)$, $t_{em}(0)$ are the clock values of the first and the last packets received by the master capture device. (All timestamps of capture device k are synchronized to those of the master capture device 0.) This rewrite is performed for all the capture devices except for the master capture device 0.

$$\overline{t_k}(i) = a_k \cdot t_k(i) + b_k \tag{1}$$

where

$$a_k = \frac{t_{s1}(0) - t_{en}(0)}{t_{s1}(k) - t_{en}(k)}, b_k = \frac{t_{s1}(k) \cdot (t_{en}(0) - t_{s1}(0))}{t_{s1}(k) - t_{en}(k)}$$

We applied the clock synchronization protocol to the 10 experiments described in Section 4.4. *PC1* and *PC2* are regarded as a capture device and the master capture device, respectively. For the 10 experiments, the maximum error after clock synchronization is about 16.8 micro-seconds for 9 experiments. At one experiment, the clock synchronization protocol stops. (This corresponds to the case that the clock frequency changes after 445 seconds elapse in Fig.5.) This result shows that our clock synchronization protocol can synchronize clocks within an error of scores of micro-seconds. This definitely fulfills the tool's requirement, i.e., the maximum effort within scores of micro-seconds.

5 End-to-End IP Packet Header Extracting Algorithm

5.1 Design Principles

It is not trivial to capture and record all packets in the 1xEV-DO system without any drop. We should be careful to handle packets at a link between a PCF/ANC

and a PDSN. The PDSN receives packets with GRE/IP headers (we call these packets as GRE/IP packets) to which an end-to-end IP packet encapsulated according to PPP by an AT is segmented. Each size of segmented packet is either 30 or 50 bytes long as shown in Fig.2 of Section 2. It is difficult for an off-the-shelf PC with tcpdump software to record a number of small GRE/IP packets into its RAM disk at the link rate of 100Mbps without any drop. Since many flows between ATs and servers are aggregated at this link, the total traffic is about 100Mbps. Thus we decide to record only end-to-end IP packet headers instead of all packets at the link between a PCF/ANC and a PDSN. Please note that a timestamp at the link between a PCF/ANC and a PDSN is the time when the last GRE/IP packet is captured.

5.2 End-to-End IP Packet Header Extraction Algorithm

In order to record only end-to-end IP packet headers, we design a real-time header extraction algorithm. The algorithm is not trivial because boundaries between PPP frames which encapsulate end-to-end IP packets do not always correspond to the beginning/end of GRE/IP packets. Besides a PPP header does not have a PPP payload length and a size of a PPP frame is not the same as that of an encapsulated end-to-end IP packet due to PPP escape sequences.

The algorithm finds boundaries of PPP frames as small number of accesses to captured GRE/IP packets as possible. It makes use of the fact that an end-to-end IP packet header follows just after a PPP header and that the PPP payload is longer than the end-to-end IP packet by the size of the PPP escaped sequences. For example, PPP escapes 0x7E to two bytes 0x7D-5E because 0x7E is used as the Flag Sequence. Thus the algorithm skips reading bytes from the captured GRE/IP packets while the sum of skipped GRE/IP packet sizes is less than the end-to-end IP packet size, and then reads bytes from the next GRE/IP packet on byte-to-byte basis to find a flag sequence of the next PPP frame.

The details of the algorithm are as follows, as illustrated in Fig.6. The algorithm uses variable *REMAINING_BYTES* to skip reading the bytes described above.

1. The algorithm captures a GRE/IP packet.
2. Bytes of the captured GRE/IP packet are read on byte-to-byte basis until finding Flag Sequence (0x7E) of a PPP frame. After finding Flag Sequence, it goes to 3. Otherwise, it goes to 1.
3. "Protocol" field in the PPP header is checked. If "Protocol" field is IP packet, it goes to 4. Otherwise, since this PPP frame is not IP packet, it goes to 2 for finding the next PPP frame.
4. It obtains an end-to-end IP packet length by reading an IP packet length field of an end-to-end IP packet header just after the PPP header. If the current GRE/IP packet does not include the IP packet length field, the next GRE/IP packet is read on byte-to-byte basis until finding the IP packet length field. Then, the end-to-end IP packet length is set to variable *REMAINING_BYTES*.

5. When the next GRE/IP packet is captured, it checks the payload size of the GRE/IP packet. If the payload size is less than *REMAINING_BYTES*, the end of current PPP frame does not exist in this GRE/IP packet. In this case, *REMAINING_BYTES* is decreased by the payload size of this GRE/IP packet, and it repeats this procedure. Otherwise, since there is the end of the current PPP frame in this GRE/IP packet, it goes to 6.
6. Bytes of the captured GRE/IP packet are read on byte-to-byte basis until finding Flag Sequence as the end of the current PPP frame. Then, it puts the end-to-end IP packet header information with this GRE/IP packet timestamp to a RAM disk, and it goes to 2.

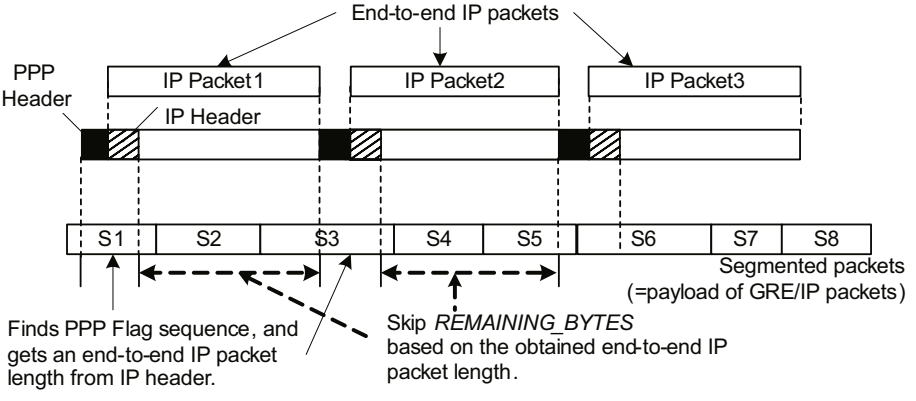


Fig. 6. End-to-end IP Packet Header Extraction Algorithm

We have measured performances of our algorithm recording only IP packet headers and the tcpdump software recording packets. The traffic is collected from the real 1xEV-DO system. *Tcpreplay* [8] is used to replay the collected traffic with various rates. The performance tests are performed using PCs with GbE (Giga bit Ethernet) network interface cards. The number of transmitted packets is 7,119,000 (1,228Mbytes), which includes 1,547,000 end-to-end IP packets. Table 1 shows the numbers of end-to-end IP packets correctly recorded by tcpdump,

Table 1. How many Packets/Headers are recorded?

Traffic Packet per sec	80Mbps 63kpps	100Mbps 79kpps	120Mbps 96kpps
tcpdump	1546963 (99.9%)	1539242 (99.50%)	1541213 (99.62%)
Our algorithm	1547000 (100%)	1547000 (100%)	1547000 (100%)

Note: The percentage means the ratio of how many packets (tcpdump) or headers (our algorithm) are recorded.

and the numbers of end-to-end IP packet headers recorded by our algorithm. The performance of our algorithm is better than that of tcpdump. Our algorithm did not drop any end-to-end IP header even for 120 Mbps traffic. On the contrary, the tcpdump software dropped some end-to-end IP packets.

6 Actual Tests with the Developed Tool

We applied the analysis tool to tests of the BCMCS system which provides multicast packet delivery to ATs using forward-links of the 1xEV-DO system. The network nodes and protocol stacks are those of Fig. 1 except for that MIP and HAs are not used. Important requirements to the BCMCS system are that the number of lost packets should be as small as possible and that every packet should be processed by the PDSN within hundreds of micro-seconds. The main objections of these tests are to check whether the PCF/ANC and the PDSN satisfies the above requirements under the condition that radio-link's quality is good and that no congestion occurs at the backbone network.

We conducted about 50 test scenarios in this environment. At each scenario, a server sends multicast packets to ATs at various sending rates. In order to check how long each packet is process at network nodes, we set capture devices at the link between the server and the PDSN (*observation point 1*), the link between the PDSN and the PCF/ANC (*observation point 2*) and the AT (*observation point 3*).

During the tests, we found more than 10 software bugs of the PDSN and the PCF/ANC. The bugs are classified into two bugs. Fig.7 (a) and (b) show typical examples of these two. Please note that several packets are received by the AT at the same time in Fig.7 (a) and (b) because such packets are encoded using some FEC (Forward Error Collection) method and these packets are decoded at the same time by the AT.

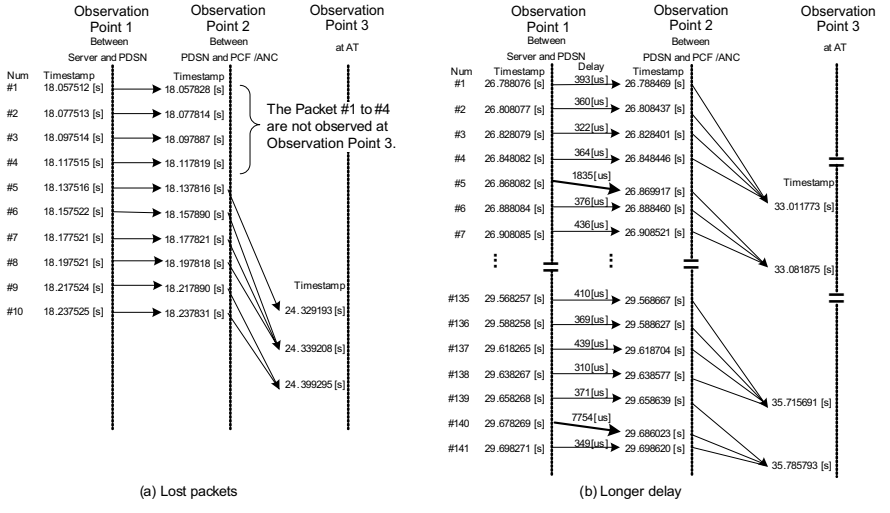


Fig. 7. Packet sequences with typical bug cases

Lost packets at the PCF/ANC
At some scenario, we observed packet losses at the AT even if there is no congestion in the network. Fig.7 (a) shows the outputs of the analysis tool, i.e., the timestamps of the multicast packets (these are end-to-end IP multicast packets) at three observation points. We easily knew that the multicast packets were lost between the observation points 2 and 3. It means that they were lost either at the PCF/ANC or the ANTS. Then we identified that they were lost at the PCF/ANC by analyzing the communication logs of the PCF/ANC. Finally, after talking with its vendor, we found that the PCF/ANC had a software bug such that multicast packets are lost with a specific condition.

Longer Duration of Processing Packets at PDSN
At another scenario, the test seemed to be successful because all the multicast packets were correctly received by the AT. However, we observed two multicast packets (Packet #5 and #140 in Fig.7 (b)) went through longer delay than one millisecond between the observation points 1 and 2 although most packets did about 400 micro-second delays. We identified that the longer delay occurred at the PDSN and asked its vendor to identify the reason and fix the problem. Finally, we knew that this was caused by the software bug of how the PDSN handles packets at its memory.

Although the vendors intensively tested the software for the PCF/ANC and the PDSN, they missed these bugs. It means that the analysis tool was useful to detect such bugs. Especially, in the case of Fig.7 (b), we would miss the software bug unless we used the analysis tool.

7 Related Work

As far as we know, there is no testing tool which captures packets or packet headers with synchronized timestamps at multiple observation points. Besides, there would be no study for extracting IP headers from packets which are transferred on 3G mobile systems.

On the contrary, clock synchronization is studied for distributed systems. The most straight-forward way is to use the Global Positioning System [9]. Many products are commercialized using the pps (plus per second) signal of GPS and synchronize their clocks with an absolute precision of better than 10us to the absolute time, i.e. UTC (Universal Time, Coordinated). However, the GPS requires a clear sky view, which is usually unavailable in our test-bed environments. There is another product, which synchronizes clocks with a precision of better than 1 nano-second using common timing signal [10] among nodes only which are located within a few meters. This requirement is not fulfilled in our test-bed environment and this product requires a special and expensive hardware.

Over the years, many clock synchronization protocols, which would be implemented based on the software, have been designed for distributed system [11,12,13,14,15,16]. NTP [11] and SNTP [12] are most prominent clock synchronization protocols used in Internet; however, their several millisecond

level precision is not enough for our testing tool. IEEE1588 [13] is a clock synchronization protocol over local area networks. Although this precision is one micro-second level on some types of local area networks, it is vulnerable to random delay on local area network switches which would be used as the broadcast medium of our testing tool. Some clock synchronization protocol implementations [14,15] achieve several micro-second precision. However, these optimize firmware of MAC (Media Access Control) protocol and require some hardware support.

Our clock synchronization scheme is similar to the clock synchronization protocol of [16] in that the both use the broadcast medium for compensating non-deterministic packet delays. However, taking advantage of the fact that clock synchronization needs to be maintained only for a testing duration, which would be just less than scores of minutes, we adopt a post-processing approach where the timestamps are synchronized after all IP packet headers are captured with timestamps. This makes our scheme far simple and efficient from the clock synchronization protocol of [16].

8 Conclusion

This paper has proposed a multi-point packet header analysis tool for end-to-end performance tests for 3rd generation (3G) mobile systems. This tool extracts end-to-end IP packet headers with synchronized timestamps at multiple observation points. The synchronized timestamps at multiple observation points enable testers to identify a reason of performance degradation and to check whether each network node processes a packet within a predefined threshold. We implemented this tool on off-the-shelf hardware platforms, i.e., lap-top personal computers, which enable this tool to be used widely for various purposes. The notable features of this tool are scores of micro-second precision in clock synchronization without using any special hardware and a sophisticated end-to-end IP packet header extraction algorithm without any drop. Due to these features, this tool is so practical and useful that it was used for testing a few commercial 3G mobile systems. Especially, scores of micro-second precision in synchronized clocks was useful to detect more than 10 bugs of some network nodes which were not detected by their vendors. This was helpful to launch the commercial 3G services on time. Besides, although this tool is developed for testing 3G mobile systems, the main functions of multi-point packet capture and scores of micro-second level clock synchronization are so general that this tool is used for testing various IP-based communication systems.

References

1. 3GPP2: cdma 2000 High Rate Packet Data Air Interface Specification. 3GPP2 Spec. of C.S.0024-0 v4.0 (2002)
2. 3GPP2: Wireless IP Network Standard. 3GPP2 Spec. of X.S0011-001-C v1.0 (2003)

3. ETS 300 406: Methods for Testing and Specification (MTS). Protocol and profile conformance testing specifications Standardization methodology (1995)
4. IXIA, http://www.ixiacom.com/
5. 3GPP2: cdma2000 High Rate Broadcast-Multicast Packet Data Air Interface Specification. 3GPP2 Spec. of C.S.0053-0 v2.0 (2005)
6. Kyungtae, K., Jinsung, C., Heonshik, S.: Dynamic packet scheduling for cdma2000 1xEV-DO broadcast and multicast services. In: Proc. of Wireless Communications and Networking Conference 2005 (WCNC 2005), vol. 4, pp. 2393–2399 (2005)
7. tcpdump, http://www.tcpdump.org/
8. tcpreplay, http://tcpreplay.synfin.net/
9. Kaplan, E.D. (ed.): Understanding GPS: Principles and Applications. Artech House, Norwood (1996)
10. National Instruments Corporation PXI System, http://www.ni.com/pxi/
11. Mills, D.L.: Internet Time Synchronization: Network Time Protocol. IEEE Trans. Communications 39(10), 1482–1493 (1991)
12. Mills, D.L.: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI. RFC 2030, IETF (1996)
13. IEEE1588: IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE Standard 1588 (2002)
14. Liao, C., Martonosi, M., Clark, W.: Experience with an adaptive globally-synchronizing clock algorithm. In: Proc. of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1999), pp. 106–114 (1999)
15. Maroti, M., Kusy, B., Simon, G., Ledeczi, A.: The Flooding Time Synchronization Protocol. In: Proc. of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys 2004), pp. 39–49 (2004)
16. Elson, J., Girod, L., Estrin, D.: Fine-Grained Network Time Synchronization using Reference Broadcasts. In: Proc. of OSDI, Operating systems design and implementation 2002, pp. 147–163 (2002)

A Learning-Based Approach to Unit Testing of Numerical Software

Karl Meinke and Fei Niu

Royal Institute of Technology, Stockholm
karlm@csc.kth.se, niu@csc.kth.se

Abstract. We present an application of learning-based testing to the problem of automated test case generation (ATCG) for numerical software. Our approach uses n-dimensional polynomial models as an algorithmically learned abstraction of the SUT which supports n-wise testing. Test cases are iteratively generated by applying a satisfiability algorithm to first-order program specifications over real closed fields and iteratively refined piecewise polynomial models.

We benchmark the performance of our iterative ATCG algorithm against iterative random testing, and empirically analyse its performance in finding injected errors in numerical codes. Our results show that for software with small errors, or long mean time to failure, learning-based testing is increasingly more efficient than iterative random testing.

1 Introduction

For black-box specification-based testing, (see e.g. [11]) an important scientific goal is *automated test case generation* (ATCG) from a formal requirements specification, by means of an efficient and practical algorithm. A general approach common to several tools is to apply a satisfiability algorithm to a formal specification and/or a code model in order to generate counterexamples (test cases) to correctness. For an SUT that involves floating point computations one important problem is therefore to find an an expressive formal requirements language suitable for modeling floating point requirements together with an efficient satisfiability algorithm for generating floating point counterexamples. One possibility is to consider the first-order language and theory of real closed fields for which satisfiability algorithms have been known since [14].

To achieve high coverage levels it is important to go beyond individual test case generation towards *iterative testing techniques* that can iteratively generate a large number of high quality test cases in a reasonable time. In [10] we identified one such iterative heuristic for ATCG that we term *learning-based testing* (LBT). Our earlier work concerned black-box unit testing of numerical programs based on:

(i) simple learning algorithms for 1-dimensional problems,
(ii) simple requirements specifications which are quantifier free first-order formulas, and
(iii) an elementary satisfiability algorithm based on algebraic root solving for cubic 1-dimensional polynomials.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 221–235, 2010.

In this paper we present a systematic and more powerful extension of LBT that is suitable for black-box testing of complex numerical software units, including high-dimensional problems by means of n-wise (e.g. pairwise) testing. We apply n-dimensional polynomial learned models that can support n-wise testing and thus tackle the dimensionality problem associated with software units. We generalise our previous learning algorithm to support n-dimensional piecewise polynomial models using non-gridded data. This step also tackles the dimensionality problem on the SUT level. Finally, we use the Hoon-Collins cylindric algebraic decomposition (CAD) algorithm for satisfiability testing (see e.g. [1]). This is a powerful satisfiability algorithm for first order formulas over the language of real closed fields. Thus we greatly expand the complexity of the requirements specifications that our tool can deal with.

It is natural to question the achieved quality of the test cases generated by any new TCG method. In the absence of a theoretical model of efficiency, we have benchmarked the quality of our learning-based ATCG empirically, by comparing its performance with that of an iterative random test case generator. This was the only other iterative ATCG method for floating point computations to which we had any access. Since iterative random testing (IRT) is closely related to measures of mean time to failure (MTF), our benchmarking results have a natural and intuitive interpretation.

To carry out this performance comparison systematically, it was necessary to avoid the experimental bias of focussing on just a small number of specific SUTs and specific requirements specifications. To generate the largest possible amount of comparative data we automated the synthesis of a large number of numerical SUTs, their black-box specifications and their mutations. In this way we compared the performance of learning-based and iterative random testing over tens of thousands of case studies. Our average case results over this data set demonstrate that when mutated errors have small effects (or equivalently when the MTF of the mutated SUT is long) then learning based testing is increasingly superior to random testing.

The structure of this paper is as follows. In Section 1.1 we review some related approaches to unit testing of numerical software and ATCG, found in the literature. In Section 2, we discuss the general principles of learning-based testing. In Section 3, we present our learning-based testing algorithm for numerical programs. In Section 4, we describe our evaluation technique, and present the results of evaluation. In Section 5 we draw some conclusions from our work.

1.1 Related Work

There are two fields of scientific research that are closely related to our own. On the one hand there is a small and older body of research on TCG for scientific computations. On the other hand, there is a more recent growing body of research on ATCG methods obtained by combining satisfiability algorithms with computational learning. Finally, there is an extensive mathematical theory of polynomial approximation, see for example [12].

Unit Testing for Scientific Software. It is widely recognised that the problem of testing scientific software has received relatively little attention in the literature on testing. The existing literature seems focussed either on manual techniques for TCG, or ATCG for individual test cases. Besides iterative random testing, we are not aware of any other approach to iterative ATCG for numerical programs.

The intrinsic unreliability of many existing libraries of scientific code for the earth sciences has been empirically studied at length in [6] and [7], which cite static error rates in commercial code of between 0.6% and 20% depending upon the type of program error.

The correctness of scientific library codes for metrology is considered in [4]. Here the authors consider black-box unit testing of well specified computations such as arithmetic mean and deviation, straight-line and polynomial regression. They apply 1-wise testing where all fixed parameter values and one variable parameter value are chosen from sets of reference data points that are manually constructed to span a given input profile. Reference values are then algorithmically computed using numerical codes which the authors posit as providing reliable benchmarks. The actual outputs produced by the SUT are compared with these reference values using a composite performance measure. This measure is more sophisticated than an individual error bound as it can account for the degree of difficulty of the chosen reference data set. The approach seems more oriented towards measuring overall code quality while our own approach focusses on finding individual coding errors quickly. Nevertheless, [4] successfully highlights problems of numerical stability occurring in a variety of well known scientific libraries and packages including NAG, IMSL, Microsoft Excel, LabVIEW and Matlab, thus confirming the real problem of correctness even among simple and widely used scientific codes.

The method of manufactured solutions (MMS) presented in [13] provides a theoretically well-founded and rigorous approach for generating a finite set of test cases with which to test a numerical solver for a PDE (usually non-linear). The test cases are constructed by analysis of the underlying mathematical model as a non-linear system operator $L[u(x,\ y,\ z,\ t)]$ to produce a finite set of analytical solutions for different input parameter values. Like our own approach (but unlike [4]) MMS does not make use of any benchmark or reference codes for producing test cases. This method is shown to be effective at discovering order-of-accuracy errors in fault injected codes in [8]. The main weakness of MMS is its restriction to a specific class of numerical programs (PDE solvers). By contrast, the generality of our requirements language (first-order logic) and our modeling technique (piecewise polynomials) means that by general results such as the Weierstrass Approximation Theorem, we can perform ATCG for a much larger class of programs and requirements.

ATCG using Computational Learning. Within the field of verification and testing for reactive systems, the existence of efficient satisfiability algorithms for automata and temporal logic (known as model checkers) has made it feasible to apply computational learning to ATCG using similar principles to our own. This

approach is known as *counterexample guided abstraction refinement* (CEGAR). Some important contributions include [3], [2] and [5]. The basic principles of CEGAR are similar to those we outline in Section 2, though our own research emphasizes *incremental learning algorithms* and related *convergence measures* on models both to guide model construction and to yield abstract black-box coverage measures. While existing research on CEGAR emphasizes reactive systems, temporal logic and discrete data types, our work presented here considers procedural systems, first order logic and continuous (floating point) data types. However, the success of CEGAR research strongly suggests to generalize this approach to other models of computation, and our work can be seen as such a generalization.

2 Learning-Based Testing

The paradigm for ATCG that we term *learning-based testing* is based on three components:

(1) a (black-box) *system under test* (SUT) S,

(2) a *formal requirements specification Req* for S, and

(3) a *learned model* M of S.

Now (1) and (2) are common to all specification-based testing, and it is really (3) that is distinctive. Learning-based approaches are *heuristic iterative methods* to search for and automatically generate a sequence of test cases until either an SUT error is found or a decision is made to terminate testing. The heuristic approach is based on *learning a black-box system using tests as queries*.

A learning-based testing algorithm should iterate the following five steps:

(Step 1) Suppose that n test case inputs $i_1, \ldots, i_n$ have been executed on S yielding the system outputs $o_1, \ldots, o_n$. The n input/output pairs $(i_1, o_1), \ldots, (i_n, o_n)$ can be synthesized into a learned model M_n of S using an *incremental learning algorithm*. Importantly, this synthesis step involves a process of *generalization* from the data, which usually represents an incomplete description of S. Generalization gives the possibility to predict as yet unseen errors within S during Step 2.

(Step 2) The system requirements *Req* are satisfiability checked against the learned model M_n derived in Step 1. This process searches for counterexamples to the requirements.

(Step 3) If several counterexamples are found in Step 2 then the most suitable candidate is chosen as the next test case input i_{n+1}. One can for example attempt to identify the most credible candidate using theories of *model convergence* (see Section 3.3) or *approximate learning*.

(Step 4) The test case i_{n+1} is executed on S, and if S terminates then the output o_{n+1} is obtained. If S fails this test case (i.e. the pair (i_{n+1}, o_{n+1}) does not satisfy *Req*) then i_{n+1} was a *true negative* and we proceed to Step 5. Otherwise

S passes the test case i_{n+1} so the model M_n was inaccurate, and i_{n+1} was a *false negative*. In this latter case, the effort of executing S on i_{n+1} is not wasted. We return to Step 1 and apply the learning algorithm once again to $n+1$ pairs $(i_1,\ o_1),\ \ldots,\ (i_{n+1},\ o_{n+1})$ to synthesize a refined model M_{n+1} of S.

(Step 5) We terminate with a true negative test case $(i_{n+1},\ o_{n+1})$ for S.

Thus an LBT algorithm iterates Steps $1 \ldots 4$ until an SUT error is found (Step 5) or execution is terminated. Possible criteria for termination include a bound on the maximum testing time, or a bound on the maximum number of test cases to be executed. From the point of view of abstract black-box coverage, an interesting termination criterion is to choose a minimum value for the convergence degree d_n of the model M_n as measured by the difference $|M_n| - |M_{n-1}|$, according to some norm $|.|$ on models. We can then terminate when M_n achieves this convergence degree.

This iterative approach to TCG yields a sequence of increasingly accurate models M_0, M_1, M_2, ..., of S. (We can take M_0 to be a minimal or even empty model.) So, with increasing values of n, it becomes more and more likely that satisfiability checking in Step 2 will produce a true negative if one exists. Notice if Step 2 does not produce any counterexamples at all then to proceed with the iteration, we must construct the next test case i_{n+1} by some other method, e.g. randomly.

3 Algorithm Description

In this section we show how the general framework of learning-based testing, described in Section 2, can be instantiated by: (i) piecewise polynomial models M_i, (ii) incremental learning algorithms, and (iii) an implementation of the CAD algorithm in Mathematica$^{\text{TM}}$, used as a satisfiability checker. The resulting *learning-based ATCG* can be used to automatically unit test numerical programs against their requirements specifications expressed as first order formulas over the language of real closed fields.

3.1 Piecewise Polynomial Models

The learned models M_i that we use consist of a set of overlapping local models, where each local model is an n-dimensional and d-degree polynomial defined over an n-dimensional sphere of radius r over the input/output space. Since $(d+1)^n$ points suffice to uniquely determine an n-dimensional degree d polynomial, each local model has $(d+1)^n$ such points. One such point $c \in \mathbf{R}^n$ is distinguished as the *centre point*. The *radius* r of a local model is the maximum of the Euclidean distances from the centre point to each other point in that model. Figure 1 illustrates this for the simple case $n = 1$ and two overlapping 1-dimensional cubic polynomial models of degree 3. With n-dimensional polynomials we can automate n-wise testing, e.g. for $n = 2$ we obtain pairwise testing. Concretely, a model M_i is represented as an array of *LocalModel* objects.

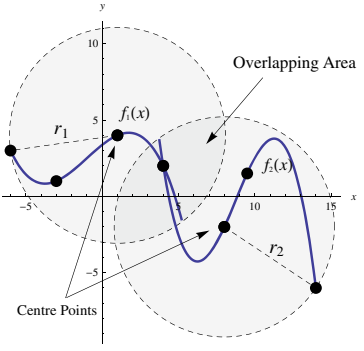


Fig. 1. Two cubic local models $f_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i,\ i = 1, 2$

We use a *non-gridded approach* to selecting the centre point for each local model. This means that the centre point c can take any value $x \in \mathbf{R}^n$ and is not constrained to lie on a vertex of an n-dimensional grid of any specific mesh size. Non-gridded modeling helps avoid an exponential blowup in the number of grid vertex points as the model dimension n increases. It also allows us to choose test case values with complete freedom. So test cases can cluster densely within areas of suspected errors, and sparsely within areas of likely correct behavior. This is one reason why LBT exceeds the performance of iterative random testing since the latter uniformly samples the search space of test cases.

3.2 A Learning-Based Testing Algorithm

We now give a concrete instantiation of the abstract learning-based ATCG, outlined in Section 2, for numerical programs. This combines incremental learning methods for the piecewise polynomial models described in Section 3.1 together with an implementation of the CAD algorithm for satisfiability checking such polynomial models against requirements specifications. As is usual for procedural programs, a requirements specification for a numerical program S under test is expressed as a *Hoare triple* (see e.g. [9])

$$\{pre\}S\{post\},$$

where the *precondition pre* and *postcondition post* are first order formulas over the language of real closed fields. Thus *pre* and *post* describe constraints on the input and output floating point variables of S at the start and end of computation. A true negative (or failed) test case is any test case which satisfies the triple $\{pre\}S\{\neg post\}$ under the usual partial correctness interpretation of Hoare triples. Our algorithm automatically searches for at least one true negative test case where S terminates. (We handle non-termination, should it arise, with a simple time-out.) This approach is easily extended to multiple negatives if these are required.

Before satisfiability checking can be applied, it is necessary to have at least one local polynomial model. Therefore the ATCG procedure given in Algorithm 1

Algorithm 1. Learning-BasedTesting

Input:
1. $S(t : TestCase)$ - the SUT expressed as a function
2. $pre, post$ - pre and postcondition for the SUT
3. $d : int$ - the maximum approximation degree of *poly*
4. $max : int$ - the maximum number of tests
5. $C : int$ - the maximum number of model checks for each test

Output: `ErrorFound` or `TimeOut` if no error was found

```
   // Initialisation phase
 1 T, M ← ∅ // Initialise set T of test cases and array M of models
 2 n ← input dimension of S
 3 support_size ← pow(d + 1, n)
 4 T ← set of support_size randomly generated test cases
 5 foreach t in T do
 6     t.output ← S(t) // run S on t
 7     if t violates post then
 8         return ErrorFound + t.toString()
 9 foreach TestCase t in T do
10     M.add(LearnModel(T, t, support_size))
   // Learning-based phase
11 count ← 0
12 while count < max do
13     t ← null
14     for i ← 0 to min(C, length(M)) do
15         m ← M[i]
16         t ← FindInstance (m, pre ∧ ¬post)
17         if m.converg < ϵ then
               // Delete converged local model
18             M.delete(m)
19         if t is NOT null then
20             break
21     if t is null then
22         t ← FindInstance (M[random()], pre)
23     t.output ← S(t) // run S on t
24     if t violates post then
25         return ErrorFound + t.toString()
26     M.add(LearnModel(T, t, support_size)) // cf. Algorithm 2
27     T.add(t)
28     M ← UpdateModels(M, t, support_size, C) // cf. Algorithm 3
29     count ← count + 1
30 return TimeOut
```

below is divided into an *initialisation* phase and a *learning-based* phase. During the initialisation phase (lines 1-10), the minimum number $(d+1)^n$ of test cases necessary to build one local model, is randomly generated (line 4). Each such test case t is executed on the SUT S and the output is stored (line 6). During the iterative learning-based phase (lines 11-30), on each iteration we try to generate a new test case either through: (i) performing satisfiability checking on the learned model (line 16) or, (ii) random test case generation (line 22). Whenever a new test case is generated and executed, the result is added to the model (line 26). Then the local models nearest the test case are updated and refined using this test case (line 28).

Note that Algorithm 1 makes two API calls to the Mathematica kernel[1]. Line 16 of Algorithm 1 makes a call to the Mathematica kernel in order to satisfiability check the formula $pre \wedge \neg post$ against the local model m, for each of the C best converged local models. The kernel function *FindInstance*(...) is the Mathematica implementation of the Hoon-Collins CAD algorithm. If a satisfying variable assignment to $pre \wedge \neg post$ over m exists then this kernel call returns such a variable assignment. In the case that no counterexample is found among the C best converged local models, then line 22 of Algorithm 1 makes a call *FindInstance(M[random()], pre)* to the same kernel function to find a satisfying variable assignment over a randomly chosen local model for precondition *pre*.

As shown in Algorithm 1, an array M of *LocalModel* objects is maintained. We use $M[i]$ for $0 \leq i < length(M)$ to denote the i-th element of M and $M[i:j]$ for $0 \leq i \leq j < length(M)$ to refer to the subinterval of M between i and $j-1$, inclusive.

3.3 Learning Local Models

The two subprocedures *LearnModel* and *UpdateModels* called in Algorithm 1 are detailed in Algorithms 2 and 3. These implement a simple incremental learning method along the general lines described in Step 1 of Section 2. Algorithm *LearnModel* infers one new local model using the newly executed test case t on the SUT. We use a simple linear parameter estimation method in line 6. The use of more powerful non-linear estimation methods here is an important topic of future research. Algorithm *UpdateModels* updates all the other existing local models using t and sorts the C best converged models.

In Algorithm 2 (line 6), model learning is performed using a linear parameter estimation method. The Mathematica kernel function *LinearSolve*(c, b) is used to find a vector of $(d+1)^n$ coefficients satisfying the matrix equation $c \cdot x = b$, for the approximating d degree polynomial function in n variables. Note, if the call to *LinearSolve* fails, then temporarily we have no local model around *m.centrePoint*, but this can be corrected later by calls to Algorithm 3. Also note in Algorithm 2 that without any previous model history, we are not yet able to compute the convergence value of a newly constructed local model (as we do in Algorithm 3). Hence convergence is initialized to 0.0 (line 7). This forces the new local model to have the minimum possible convergence value, so it has

[1] We use Mathematica^{TM} version 7.0 running on a Linux platform.

Algorithm 2. LearnModel

Input:
1. T - the set of all executed test cases
2. $t : TestCase$ - a newly executed test case
3. $support\_size : int$ - the number of test cases needed to build one local polynomial model

Output: a new local model with centre t

1 $m \leftarrow new\ LocalModel()$
2 $m.centrePoint \leftarrow t$
3 $m.localPoints \leftarrow$ from T pick $support\_size$ test cases nearest to t
4 $m.radius \leftarrow$ maximum Euclidean distance between t and each data point of $m.localPoints$
5 Form the matrix equation $c \cdot x = b$ from $m.localPoints$
6 $m.poly \leftarrow \texttt{LinearSolve}(c, b)$
7 $m.converg \leftarrow 0.0$
8 **return** m

Algorithm 3. UpdateModels

Input:
1. M - array of all local models obtained so far
2. $t : TestCase$ - a newly executed test case
3. $support\_size : int$ - cf. Algorithm 2
4. $C : int$ - cf. Algorithm 1

1 **foreach** *LocalModel* m *in* M **do**
2 **if** t *inside* m **then**
3 $T \leftarrow m.localPoints.add(t)$
4 $\hat{t} \leftarrow m.centrePoint$
5 $\hat{m} \leftarrow \texttt{LearnModel}(T, \hat{t}, support\_size)$
6 randomly pick N test cases $t_1, t_2, \ldots t_N$
7 $\hat{m}.converg \leftarrow \frac{\sum_{i=1,\ldots,N} |m.poly(t_i) - \hat{m}.poly(t_i)|}{N}$
8 $m \leftarrow \hat{m}$
9 **if** $C < length(M)$ **then**
10 Partially sort M to ensure $M[0 : C]$ are linearly ordered by convergence
11 Randomly permute $M[0 : C]$
12 **else**
13 Randomly permute M
14 **return** M

the possibility to be satisfiability checked during the learning-based phase even though its convergence value is undefined.

In Algorithm 3, when updating a local model, a *Monte-Carlo method* (lines 6, 7) is used to efficiently approximate the convergence value obtained using the integral norm L_1 on bounded polynomials. Each time local models are updated, the first C best converged local models are randomly swapped to the first C

positions of the array M, provided C is greater than or equal to the length of M (line 10, 11). This, together with line 14 in Algorithm 1, implements the prioritisation Step 3 of the abstract LBT algorithm of Section 2. In practice, the value of C is empirically determined. A higher value of C can probably take better advantage of model checking while it slows down the speed of learning-based testing if model checking is time consuming.

4 Experimental Evaluation

4.1 Construction of SUTs, Specifications and Mutations

We wished to benchmark the performance of LBT against another iterative ATCG method for floating point computations. The simplest and most obvious candidate for comparison was iterative random testing, which is fairly easy to implement when requirements specifications are simple. This comparison has the advantage that iterative random testing can be viewed as a Monte Carlo method to estimate the mean time to failure (MTF) of an SUT over an equiprobable distribution of input values. Our experiments confirmed that this estimated MTF value was inversely proportional to the size of injected mutation errors, as one would expect.

In order to obtain the largest possible data set of performance results, we used random generators to construct the numerical SUTs, their first-order logic specifications and their mutations. These also allowed us to perform experiments in a controlled way, in order to more accurately assess factors that influence the performance of our ATCG. The random numerical program generator (RPG) and specification generator (SG) were used to generate hundreds of SUTs with their specifications and introduce thousands of mutations. Iterative random testing was also repeated hundreds or thousands of times on each SUT, until the estimated MTF value appeared well converged.

Random Numerical Program Generation (RPG). To randomly generate a numerical program as an SUT, the RPG divides a global n-dimensional input space into subspaces at random. Within each of these randomly chosen subspaces, the RPG generates an n-dimensional polynomial surface of random shape (i.e. coefficients) and degree. We allowed for much higher degrees in such SUT models than in the learned models of Section 3.1 in order to ensure that the learning problem would be non-trivial. When a random SUT was generated, we then mutated it randomly to inject faults by changing the shapes or positions of the polynomial surfaces in one or more subspaces. Figure 2 gives an example of a randomly generated SUT for one dimensional input and six randomly chosen subspaces. To mutate the SUT, we simply regenerate different curves over some of the same subspaces, and this is also shown in Figure 2. The ratio of the total size of the mutated subspaces to the total size of all subspaces represents the *percentage error size*. Controlling this percentage error size experimentally, was the key to understanding the relative performance of LBT and iterative random testing. For example in Figure 2, the percentage error size is 50%.

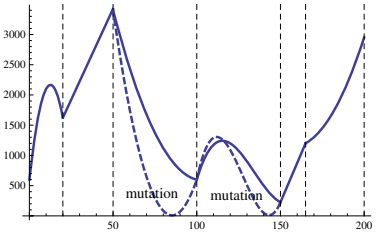


Fig. 2. A randomly generated SUT: 6 subspaces of which 2 are mutated

Random Specification Generation (SG). Since for evaluation purposes a large number of SUTs were automatically randomly generated, it was necessary to automatically generate their requirements specifications too. At the same time it was necessary to ensure that the requirement generated for each SUT was semantically correct in its unmutated state. It is well known (see [1]) that the logical complexity of requirements formulas has an effect on the efficiency of satisfiability checking, and hence on LBT as a whole. To explore the effects of this complexity, we studied the relative performance of LBT and IRT against two different logical types of requirements specifications.

Let S be a randomly generated SUT with k subspaces and 1-dimensional input $S = f_1(x),\ f_2(x),\ \ldots,\ f_k(x)$. Then for both types of requirements specifications, the same precondition was used. We call this an *interval bound precondition* on the input variable x of the form

$$pre(S) \equiv c_1 \leq x \leq c_{k+1}.$$

Here the input interval $[c_1,\ c_{k+1}]$ for an SUT is divided into k subintervals $[c_i,\ c_{i+1}]$ for $1 \leq i \leq k$ by the boundary values $c_1, c_2, \ldots, c_k$, and $f_i(x)$ describes the behaviour of S over the i-th subinterval $[c_i,\ c_{i+1}]$.

On the other hand two different types of postconditions could be generated: we call these *equational* and *inequational postconditions*.

For the same SUT S, its *equational postcondition* is a formula of the form:

$$eq\_post(S) \equiv \bigwedge_{i=1,\ldots,k} (\ c_i \leq x < c_{i+1}\ \Rightarrow \|f_i(x) - m_i(x)\| < \epsilon\ )$$

where $m_i(x)$ describes the mutation of $f_i(x)$ (if any) over the i-th subinterval. Intuitively, $eq\_post(S)$ asserts that the function of the mutated SUT is equal to the function of the original SUT S, to within an absolute tolerance ϵ.

The *inequational postcondition* for S is a formula of the form:

$$ineq\_post(S) \equiv \bigwedge_{i=1,\ldots,k} (\ lower\_bound < f_i(x)\ )$$

Intuitively, $ineq\_post(S)$ asserts that all values of each function $f_i(x)$ lie above a constant lower bound. The value of *lower_bound* is randomly chosen so that this postcondition is semantically correct for the unmutated (correct) program S.

These preconditions and two types of postcondition generalise in an obvious way to n-dimensional input for $n \geq 2$.

4.2 Results and Analysis

Having set up random generators for numerical programs and their specifications, we proceeded to generate a large number of SUT/specification pairs, and mutate these SUTs to achieve different percentage error sizes within the range 10% to 0.01%. We then measured the minimum number of test cases using LBT and IRT needed to find the first true negative in each mutated SUT. This measurement was chosen since for IRT it provides an estimate of the mean time to failure (MTF) of the mutated SUT under an equiprobable input distribution. (We can view IRT as a Monte Carlo algorithm to estimate MTF.) To deal with the stochastic behavior of IRT, this value was averaged out over many runs until a well converged mean value had emerged. We then compared the ratio of these two measurements, and averaged out this figure over many different SUTs and many different mutations all of the same percentage error size. The results of our experiments are given in Figure 3 which illustrates the relative performance of LBT and IRT as the percentage error size is reduced. The x-axis expresses the percentage error size (c.f. Section 4.1) on a logarithmic scale. The y-axis gives the ratio IRT/LBT of the average number of IRT test cases divided by the average number of LBT test cases. Note that Figure 3 shows two curves, one for testing SUTs against equational specifications and one for testing SUTs against inequational specifications. Also note that above an error size of 10%, both curves converge rapidly to $y = 1.0$, which explains our chosen range of error sizes.

The two distinct curves in Figure 3 clearly indicate that relative performance of LBT is influenced by the logical complexity of specifications, as we expected. However, the shapes of both curves are similar. Both curves show that as the percentage error size decreases (or equivalently the MTF of the mutated SUT increases) the efficiency of LBT over IRT increases. Since the x-axis is logarithmic, this improvement in relative performance seems approximately exponential in the percentage size of errors.

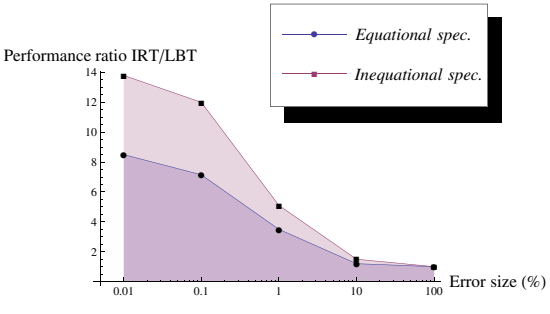


Fig. 3. Relative performance of IRT and LBT

4.3 A Concrete Case Study: Bubblesort

The statistical results of Section 4.2 may appear somewhat abstract, since the thousands of randomly generated case studies do not correspond to specific well known algorithms. Therefore, we complement this statistical analysis with a concrete case study.

```
1  class BubbleSort {
2      public void sort(double[] a) {
3          for (int i = a.length; --i >= 0; ) {
4              boolean flipped = false;
5              for (int j = 0; j < i; j++ ) {
6                  // Mutated from "if (a[j] > a[j+1]) {"
7                  if (a[j] - N > a[j+1]) {
8                      double T = a[j];
9                      a[j] = a[j+1];
10                     a[j+1] = T;
11                     flipped = true;
12                 }
13             }
14             if (!flipped) {
15                 return;
16             }
17         }
18     }
19 }
```

Fig. 4. Bubblesort algorithm with mutation

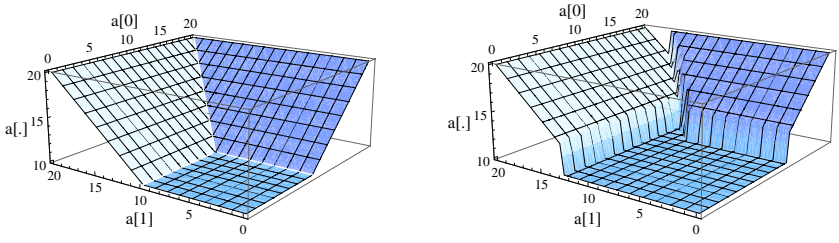
Figure 4 presents the familiar Bubblesort algorithm for an array of floating point numbers. This algorithm represents a typical high dimensional problem, since the input (and hence output) array size is usually rather large. In line 7 we introduce a mutation into the code via a parameter N. This particular mutation was mainly chosen to evaluate the quality of test cases, since it allows us to control the percentage error size of the mutation. Despite the high dimension of this SUT computation, pairwise LBT testing can find the mutation error fairly quickly. Figure 5 illustrates[2] why this is so. Taking a 2-dimensional polynomial model on any output array value, Figure 5(a) shows that the SUT itself can be modeled quite easily. Furthermore Figure 5(b) shows that the mutated SUT can also be modeled with just a little more effort, since large regions of this model are again essentially simple. A suitable requirement specification for this code is just to assert that the output array is linearly ordered:

$$\{ \bigwedge_{i=0}^{a.length-1} MIN < a[i] < MAX \}\ BubbleSort\ \{ \bigwedge_{i=0}^{a.length-2} a[i] \leq a[i+1] \}$$

where MIN and MAX are simply the lower and upper bounds of input values.

As in Section 4.2, we can measure the minimum number of test cases required by LBT and IRT to find the first true negative in the mutated SUT, against the

[2] Note that the grid structure in Figures 5(a) and 5(b) is an artifact of the Mathematica graphics package, the models themselves are non-gridded.



(a) Model segment of original SUT (b) Model segment of mutated SUT

Fig. 5. Modeling the Bubblesort algorithm with and without mutation

above requirement specification. Our results show that on average (since IRT has a stochastic performance) LBT is 10 times faster than IRT at uncovering the mutation error.

5 Conclusion

We have presented a systematic and powerful extension of the learning-based testing (LBT) approach to iterative ATCG introduced in [10]. We have compared the performance of LBT against the results of iterative random testing over a large number of case studies. Our results clearly demonstrate that LBT, while never worse than iterative random testing, can be significantly faster at discovering errors. Future research will also consider non-linear models and learning algorithms for floating point data types. More generally, we also need to consider the problem of learned models, learning algorithms and satisfiability checkers for other data types besides floating point, in order to increase the range of applicability of our testing methods.

Acknowledgement

We gratefully acknowledge financial support for this research from the Chinese Scholarship Council (CSC), the Swedish Research Council (VR) and the European Union under project HATS FP7-231620.

References

1. Caviness, B.F., Johnson, J.R.: Quantifier Elimination and Cylindrical Algebraic Decomposition. Springer, Heidelberg (1998)
2. Chauhan, P., Clarke, E.M., Kukula, J.H., Sapra, S., Veith, H., Wang, D.: Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, Springer, Heidelberg (2002)
3. Clarke, E., Gupta, A., Kukula, J., Strichman, O.: Sat-based abstraction refinement using ilp and machine learning. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, Springer, Heidelberg (2002)

4. Cox, M.G., Harris, P.M., Johnson, E.G., Kenward, P.D., Parkin, G.I.: Testing the numerical correctness of software. Technical Report CMSC 34/04, National Physical Laboratory, Teddington (January 2004)
5. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. Logic Journal of the IGPL 14(5), 729–744 (2006)
6. Hatton, L., Roberts, A.: How accurate is scientific software? ACM Transactions on Software Engineering 20(10), 786–797 (1994)
7. Hatton, L.: The chimera of software quality. Computer 40(8), 104, 102–103 (2007)
8. Knupp, P., Salari, K.: Verification of Computer Codes in Computational Science and Engineering. CRC Press, Boca Raton (2002)
9. Loeckx, J., Sieber, K.: The foundations of program verification, 2nd edn. John Wiley & Sons, Inc., New York (1987)
10. Meinke, K.: Automated black-box testing of functional correctness using function approximation. In: ISSTA 2004: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, pp. 143–153. ACM, New York (2004)
11. Poston, R.M.: Automating Specification-Based Software Testing. IEEE Computer Society Press, Los Alamitos (1997)
12. Reimer, M.: Multivariate Polynomial Approximation. Birkhäuser, Basel (October 2003)
13. Roache, P.J.: Building pde codes to be verifiable and validatable. Computing in Science and Engineering, 30–38 (September/October 2004)
14. Tarski, A.: Decision Method for Elementary Algebra and Geometry. Univ. of California Press, Berkeley (1951)

From Scenarios to Test Implementations via Promela

Andreas Ulrich[1], El-Hachemi Alikacem[2], Hesham H. Hallal[3], and Sergiy Boroday[2]

[1] Siemens AG, Corporate Technology, Munich, Germany
andreas.ulrich@siemens.com
[2] CRIM, Montreal, Canada
{sergiy.boroday,el-hachemi.alikacem}@crim.ca
[3] Lebanese International University, Beirut, Lebanon
hesham.hallal@crim.ca

Abstract. We report on a tool for generating executable concurrent tests from scenarios specified as message sequence charts. The proposed approach features three steps: 1) Deriving a MSC test implementation from a MSC scenario, 2) Mapping the test implementation into a Promela model, 3) Generating executable test scripts in Java. The generation of an intermediate Promela model allows for model-checking to inspect the test implementation for properties like soundness, fault detection power as well as for consistency checking between different test scenarios. Moreover decoupling the executable test scripts from the scenario specification makes it possible to use different backend code generators to support other scripting languages when needed.

Keywords: Scenario-based testing, distributed testing, test consistency, Promela, Message Sequence Charts, UML2 sequence diagrams, tool implementation.

1 Introduction

A recent survey on model-based testing (MBT) approaches [1] analyzed about 400 papers to find evidence about the use of MBT in industrial projects. It could identify 85 papers describing UML-based and non-UML approaches of some degree of uniqueness. However only 11 papers out of them report about an industrial application or experimental case studies that go beyond a proof of concept. It turns out that most MBT approaches of industrial strength target the domains of safety-critical or embedded systems; wider use in general software engineering is still deficient. The paper concludes that "it's […] risky to choose an MBT approach without having a clear view about its complexity, cost, effort, and skill required to create the necessary models".

The trend towards complex systems with an increasing degree of connectivity, configurability, heterogeneity, and distributedness requires improved processes and methods especially for the system integration phase. Testing is still the preferred validation method used in this context. Because of the nature of complex systems, an MBT approach seems to be a good candidate for its ability to abstract away certain system aspects that distract test engineers from the specification of proper tests.

To support system integration testing, a scenario-based testing approach is developed and re-fined to the needs of the domain of embedded systems as they are of interest, for example, at Siemens. Scenario-based testing has been broadly applied in

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 236–249, 2010.

the telecommunication domain already. It is based on the specification of interaction scenarios between components to be integrated and is typically described in terms of message sequence charts (MSCs) or UML2 sequence diagrams [2], [3]. Additional specification features however need to be added to a scenario specification such as the expression of real-time constraints to make it applicable to the considered class of systems. The UML2 profiles SysML and MARTE provide some useful language features that are an initial input to develop the new test specification approach.

This paper presents an overview about the test tool *ScenTest* that currently supports the untimed specification of scenarios for testing distributed systems and the generation of concurrent testers in Java. As a boon the tool supports the verification of test implementations as well as the consistency check between different test scenarios via the intermediately generated Promela model. The possibility to verify test implementations turned out to be particularly helpful when developing and fine-tuning the test generation algorithms outlined in [4]. The consistency checks are of great help for tool users if they incrementally build up a test suite from a number of test cases that all together should not contradict each other.

The paper is organized as follows. Section 2 introduces the scenario-based testing approach and puts it into the context of other MBT approaches. Section 3 describes the different steps of the test generation process in some detail. Afterwards, Section 4 explains the verification feature of the tool before Section 5 concludes the paper.

2 Scenario-Based Testing

In software testing practice, MBT approaches evolved as the latest innovation step as depicted in Fig. 1 [5]. To apply these approaches successfully in today's industrial software development projects, one has to put efforts to automate the execution of tests in the first place. Today, a full range of test automation solutions of varying abstraction levels exists depending on the actual application domain and the project history. However, with an improved applicability of model-driven approaches due to the provision of better tools and supporting development processes, a high impact of MBT approaches on software testing can be expected now and in the near future.

There are various modeling technologies that can be used for MBT. The orientation towards an industrial context imposes however some extra requirements and constraints. First of all, it is not wise to assume that a formal (complete and consistent) model of the software system always exists. What is reasonable to assume instead is an informal model that describes the system (or parts of it) mostly in natural language. Therefore, domain understanding and modeling play the major role for a successful application of MBT in a typical industrial context. Semi-formal models, such as UML, are becoming increasingly common in industrial projects contributing to the reduction in MBT modeling efforts. However, the development of a domain-specific language that is adequate for modeling the requirements of a given application domain remains a crucial factor that decides about the success of MBT.

Second, MBT can only be effective in industrial projects when the total efforts for applying an MBT approach are affordable. This means that the testing approach should produce readily executable tests, even if the software system is modeled only

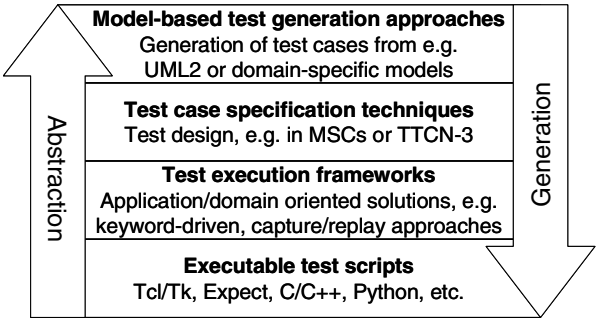


Fig. 1. Testing approaches according to their level of abstraction they offer

partially. The model should also have some level of resilience to modifications (changing requirements, product evolution). Another important aspect is that the level of abstraction in the model must be adequate for describing the intended test purposes concisely. This implies that the different abstraction levels must be bridged in order to obtain executable tests (cf. Fig. 1).

Driven by these experiences, a scenario-based testing approach is developed that differs from other MBT approaches as follows. The basic assumption is that the (sub-) system that is the subject of the integration test only needs to be specified partially in terms of test scenarios that can be observed at the system's boundary and possibly at internal interfaces if they are exposed to a tester.

The test scenario (see next section) is a restricted MSC that represents the system under test (SUT) only as a single lifeline, but considers all of the SUT's interfaces to be covered by concurrent tester components. It turned out in practice that this representation of a test scenario is flexible enough to cover a wide range of system integration tests where the SUT consists of one or many components. Since the scenario-based testing approach is basically a black-box test, a single SUT lifeline is therefore sufficient to specify the interactions of the SUT with its environment (tester) as long as its interfaces are clearly distinguishable.

Because of its capability to work with partial system specifications the entrance level to apply this method is lower than other MBT approaches that require more complete system specifications such as Conformiq Qtronic or Microsoft SpecExplorer to name just a few. Although the latter tools provide an even higher abstraction and are more powerful because test cases are automatically generated, the overall efforts to specify system specifications is still a limiting factor to apply MBT approaches in the industrial practice. Therefore we believe that scenario-based testing with appropriate tool support offers a valuable contribution to many real-world projects.

A scenario-based testing approach is not new. The closest contender of our *ScenTest* tool is Motorola's *ptk* tool [6]. Since it is an in-house tool we need to rely on published data to assess its capabilities. First of all, it clearly addresses the needs in the telecommunication domain and strictly sticks to the full MSC semantics. Outside of this domain however, a strict MSC notation is seldom used. Instead a general ubiquity of UML can be observed. With the UML2 version it became also sufficiently strong enough to be of use also for the purpose of (test) code generation. Therefore we decided to embrace rather a UML2 approach because of its higher acceptance and

also easier tool support. Another distinguishing factor of both approaches might be the capability to check the consistency between test scenarios (see Section 4) that allows to incrementally build up a more and more complete system specification (in terms of test scenarios).

3 Test Tool *ScenTest*

3.1 Tool Overview

In this section we describe an approach, employed by our prototype tool, to map test scenarios, created by the test designer from more general use case scenarios, into test scripts. In order to develop the test scenarios, the test designer uses a UML2 editor. Currently, the prototype tool works with Sparx Enterprise Architect, which is one of more affordable commercial UML editors with XMI export support.

Each test scenario is a sequence diagram or MSC with one designated SUT lifeline (instance). All the other lifelines represent test components. The test scenario can depict communications between the SUT and test components, but not among test components. All the necessary communication between test components are automatically implemented by the tool; the test designer is, therefore, relieved from doing this manually. At the moment the prototype tool supports parallel blocks with a limited support of alternative blocks and loops.

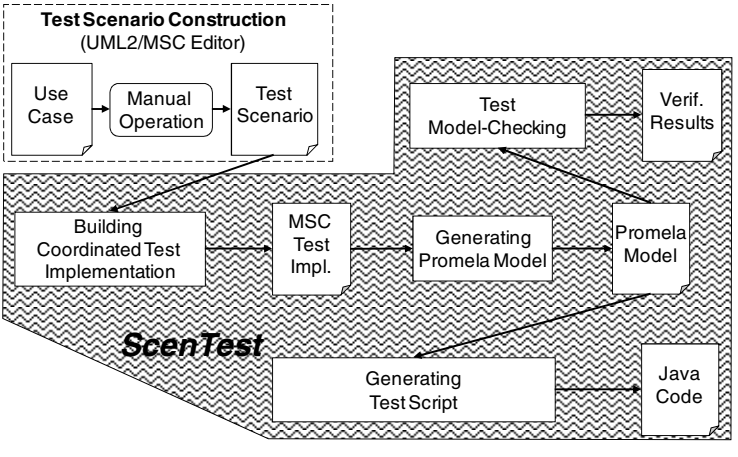


Fig. 2. *ScenTest* – Test implementation and verification framework

The tool generates test scripts in the platform independent language Promela [7] developed for modeling and automated analysis of distributed systems. While currently the test scripts are mapped further into Java, support of other languages is also foreseen. One of the benefits of using Promela is the possibility of simulation and formal verification of tests, which can be performed by the test tool developers to debug the tool itself and by the test designers who wish to simulate test scripts or check their properties. Model-checking of test script properties and test scenario consistency [8] is discussed in Section 4.

The tool chain is illustrated in Fig 2. The main advantage of the underlying approach over ones that rely on state machines or labeled transition systems, is that the test designer is not required to provide a complete formal specification of the SUT; it suffices to have a partial model in the form of an MSC scenario.

3.2 Mapping Test Scenario MSC to Test Implementation MSC

Correct implementation of a test scenario requires coordinating messages and delays [4] to ensure test soundness and increase fault detection. While the need for coordinating messages was identified in early research in scenario-based testing already [9], it has been only demonstrated recently that in addition delays have to be introduced to assure soundness [4]. For instance, consider a vending machine example (Fig. 3). The vending machine has two interfaces, one for a service man only and another for consumers. Thus, a tester for the vending machine consists of two test components emulating the behaviors of the serviceman and a consumer. If the *turn on* message is followed by the *coin* message with no delay in a test implementation, the outcome would depend on the race between these two messages, and the tester may produce the fail verdict even though the SUT behavior is correct. Thus, sound testing requires delays.

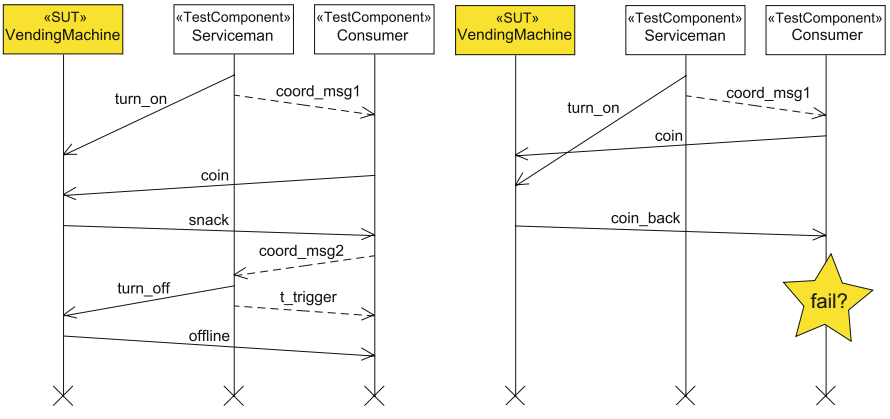


Fig. 3. Sound test implementation (left) and a possible execution of an unsound test implementation (right)

As a consequence, following the approach in [4], coordinating messages and delays are inserted. The duration of the delay is determined according to message latency. However, while approaches suggested in [4] and [9] address all the races [10] among coordinating messages by allowing them to occur concurrently, one can avoid additional concurrency by storing coordinating messages arriving from different test components in different buffers [11]. Thus, since the tool uses different buffers for different channels the degree of concurrency in the original test scenario is preserved in the test implementation. We also add additional coordinating messages, such as *t_trigger* in Fig. 3, which allow one to detect missing SUT messages using a local timeout. The duration of this timeout needs to suffice for a message to arrive at the

SUT, trigger a response message from it, and this latter message to arrive back at the test component that waits for this message.

Simple alternative blocks, where each section starts with messages sent from the SUT to the same test component, are supported by a straightforward extension of the algorithm suggested in [8]. Meanwhile, test scenarios containing alternative blocks of a general nature are treated as ill-formed scenarios at the moment. They could be still processed in future by adding additional coordinating messages, which allow different test components to inform each other of the alternative taken by the SUT.

3.3 Mapping Test Implementation MSC to Promela

Next, we map an MSC representing a test implementation into an intermediate Promela model. Promela is a modeling language for the Spin model checker [7] that is not intended for execution but for simulation and analysis of concurrent systems. Model-checking in our context helps detect errors prior to producing executable test scripts in Java. The advantages of a model checker over usual simulation are obvious since we deal with concurrent processes: model-checking allows the verification of all the possible execution paths. Since test scenarios describe finite behavior, the potential state space explosion can be usually controlled and lies in the limits of Spin.

The Promela model addresses several concerns omitted in the test implementation MSC, namely a mechanism producing the final verdict, the architecture of the test deployment, and the detection of unexpected SUT outputs. The Promela model does not address concerns related to non-deterministic channel delays and resolution of stalling test executions in case of an incorrect SUT since these issues require a notion of time.

The main idea behind the suggested Promela model generation procedure is to represent each test component and in addition each section of a parallel block by separate concurrent Promela processes and to represent alternatives using Promela's selection construct if..fi. In alternative blocks the coordinating messages are inserted similarly to the ones in parallel blocks as explained in [4].

The resulting test architecture, represented in Promela, is defined as follows. Each test component communicates with the SUT through a pair of FIFO channels representing the interfaces of the SUT. Test components are also pair-wise interconnected by a couple of unidirectional channels. Additionally, a master test component *MTC* is defined, which starts the test components and issues a final verdict:

- Pass, when all the test components notify the MTC by sending *pass* messages, and
- Fail, when at the least one test component sends a *fail* message to the MTC.

A test component can have subcomponents introduced to handle parallel blocks, which execute concurrently within the test component and which share the same channels with the test component in the communications with the SUT and other test components.

We conclude the section with a sample Promela model for a test component from the test implementation in Fig. 3 that fits to the test architecture shown in Fig. 4. In the presented code, the notation "ServiceConsumer ? coordMsg2" means the reception of message coordMsg2 by process Consumer from process Service over the channel ServiceConsumer. Similarly, the notation "ConsumerService ! coordMsg1" means

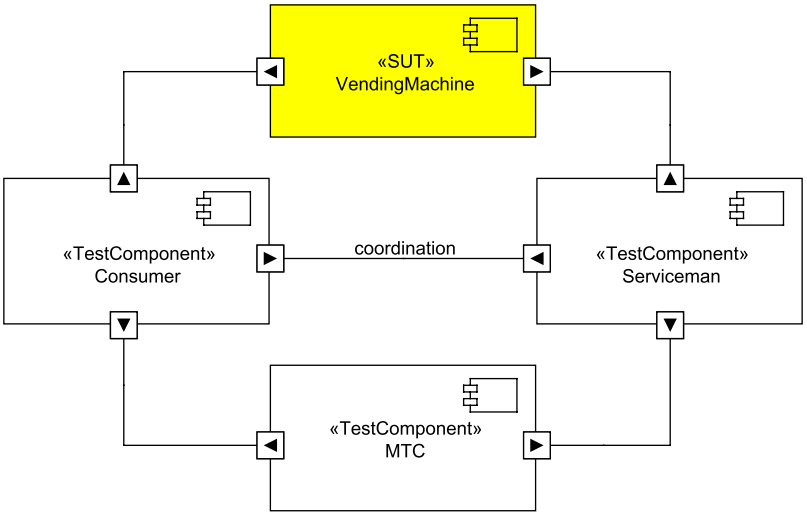


Fig. 4. Test architecture of the example system

that coordMsg1 is sent by Consumer to Service over the channel ConsumerService. In order to allow quick detection of unexpected messages we use the case selection construct of Promela if..fi along with the else clause. The use of the else clause to catch unexpected messages is only allowed in polling mode, i.e., checking the head of the input queue without actual consumption. Choice of the else clause due to the emptiness of the buffer is prevented by the nempty guard. The timeout statement after the reception of the first coordinating message is used to model delay.

```
proctype Consumer () {
  ConsumerMTC ! init_confirm;
  ServiceConsumer ? coordMsg2;
  timeout;
  ConsumerSUT ! coin;
  nempty(SUTConsumer);
  if
  :: SUTConsumer ?[snack] -> {SUTConsumer ? snack;}
  :: else -> ConsumerMTC ! fail
  fi;
  ConsumerService ! coordMsg1;
  ServiceConsumer ? timerTrigger1;
  nempty(SUTConsumer);
  if
  :: SUTConsumer ?[offline] -> {SUTConsumer ? offline;}
  :: else -> ConsumerMTC ! fail
  fi;
  ConsumerMTC ! pass;
}
```

3.4 Mapping Promela to Java

Apparently, no translator from Promela to Java is available, save few research prototypes, such as HiSpin, a Promela simulator [12], and SpinJ [13], a reimplementation of Spin in Java. Thus, we develop a translator for a subset of Promela corresponding to the language elements that are currently used to represent test implementation MSCs. For each of these constructs, we define a corresponding Java code that preserves the semantics of Promela test models. Table 1 summarizes the Promela language subset and its corresponding Java counterparts. Based on this mapping, the translator generates a multi-threaded Java code, in which each Promela *proctype* is mapped into a Java thread.

Table 1. Promela-to-Java mapping

Promela Code	**Java Code**
Proctype	Java thread
Mtype	a new class *MTypeSymbols*, in which the *mtype*'s elements are defined as static fields.
buffered channels	Java queue
data types: *byte*, *int* and *bool*	corresponding Java types
local variable	variable in the corresponding thread
operators +, -, ! (not), &&	similar operators in Java
assignment	similar construct in Java
send, receive, polling and *nempty*	invocation of corresponding methods
run	creation of a new thread
if..fi, *do..od*, *goto*, *skip*	corresponding Java patterns

To map Promela channels, we implement a dedicated Java class PromelaChannel, which represents a blocked queue data structure that contains thread-safe methods corresponding to the Promela communication operations send, receive, polling, nempty. Therefore, for each channel present in the Promela model, a PromelaChannel Java object is created. The translation of a Promela communication operation is in fact the generation of its corresponding Java method invocation on the object corresponding to the channel involved in the communication operation.

Along with that, we implement timed and non-timed Java methods to support Promela communication operations. The timed methods must be completed within a given duration, otherwise an exception is raised. Timed methods are used only in the communication of the test components with the SUT. Since the communication between test components is assumed to be reliable, the non-timed versions are employed in communications between test components. Handling the exception of a timed Java communication method consists of sending a *fail* message to the MTC indicating that an expected message from the SUT was not delivered. The Promela communication operations are translated to either timed or non-timed methods depending on whether the communication is between a test component and the SUT or between test components.

For most of the Promela constructs, the translation to the corresponding Java code is straightforward. However, in few cases specific Promela patterns, rather than individual operations are matched and translated to the corresponding Java code. This approach simplifies the translation of some Promela constructs that have no direct counterparts in Java, e.g., goto, while preserving atomicity of Promela operations.

The Promela's construct *mtype,* which is used to declare symbolic message names and constants, is mapped into a Java class *MtypeSymbols*, in which a static field is defined for each element of *mtype*.

For Promela's selection construct *if..fi*, which allows one to describe several alternative execution sequences, called options, we define a Java pattern, which is based on Java's *switch* construct that implements a similar behavior. The Java pattern for the selection construct of receive statements consists in finding an executable option (i.e., choosing an execution path) among the ones listed within the construct. In the case when no option is executable the thread either executes the *else* clause of the selection construct or, in absence of the *else* clause, re-attempts to select an executable option upon receiving a notification from the corresponding input buffer. Once an executable option is found, it is executed. To detect missing SUT output messages, the described process terminates after a certain timeout by sending a *fail* message to the MTC thread. The repetition construct *do..od*, which in Promela resembles the selection construct, is mapped in Java in the same manner. Unlike the selection construct however, options of the repetition construct can be executed several times until the *break* command is encountered.

Fig. 5 depicts the Java code generation workflow. In this workflow, the input is the generated Promela model that gets parsed to obtain the abstract syntax tree of the model. We use the ANTLR [14] tool to implement the parser for the supported subset of Promela. From an internal representation the Java code generator module produces Java code according to the mapping rules presented above.

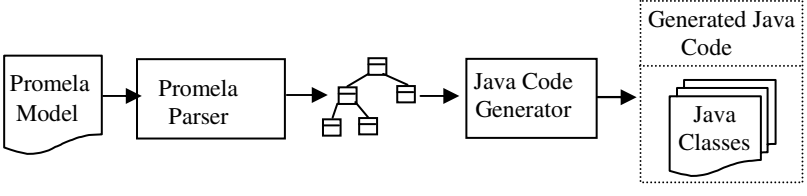


Fig. 5. Java code generation workflow

3.5 Interfacing Tester with the SUT

In the automatic test execution setup, the generated Java tester is executed against the SUT. Therefore, test adaptors, which are system specific, need to be implemented to interface the SUT with the test components. Test scenarios and implementations use abstract names for messages, which typically convey the meaning or purpose of concrete messages without revealing full details. We implement a keyword driven approach that maps abstract message names to concrete messages ready to be sent out to the SUT. The adaptor implementation accepts an abstract message from the Java

queue of a Promela channel implementation and calls the corresponding procedure to pass the input to the SUT using the appropriate technology, e.g. (remote) method invocation or network operations. Another adaptor attempts to identify a received SUT output message and maps it to an abstract message to be forwarded to the waiting test component.

4 Validation of Tests

As a direct side effect of generating an intermediate test implementation in Promela, the model checker Spin can be used to evaluate the validity of tests. In particular, we discuss in this section two test validation methods. The first method aims at verifying soundness of a test implementation by checking its model in Spin against predefined properties, mainly reachability properties to check if, for example, a pass verdict is reachable on all possible paths in the model. The second method, a model based comparison technique, aims at detecting inconsistency between different test scenarios by comparing and analyzing the corresponding SUT models.

4.1 Test Case Verification

The Promela model of a test implementation can be simulated and formally verified in Spin. In particular, model checking a test implementation model together with an SUT model can help ensure soundness of the test implementation, while model checking with a mutant SUT helps check fault detection power. In the case when an independent Promela model of the SUT is unavailable, the *ScenTest* tool is able to a build a partial model from the test scenario's SUT lifeline.

As discussed in Section 3, the tool implements the test generation approach elaborated in [4], which uses delays to guarantee the soundness of test implementations. To illustrate the latter point, we consider an example of the vending machine that deploys the Promela timeout construct to model delays. The Promela code is given in Section 3.3, while the test implementation MSC can be found in Fig. 3. We combine this test

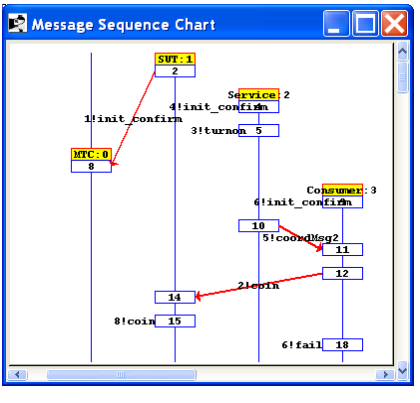


Fig. 6. Model checking demonstrates unsoundness of a test implementation without timer

scenario with a SUT model of the vending machine. Being turned on, the vending machine generates a snack when a coin is inserted. However, when the machine is not turned on, it simply returns back the coin. Model-checking with Spin against the property "eventually pass on all executions" confirms that the pass verdict is always produced. However, after removal of the delay, model-checking in Spin against the same property produces execution paths that do not result in the pass verdict. In one of such paths, shown in a Spin counterexample MSC in Fig. 6 where the two test components send turn on and coin messages without a delay between them, the latter message can overtake the former, resulting in the return of the coin and, since the return of the coin is not foreseen by the test scenario, the test subsequently fails.

4.2 Test Scenario Inconsistency Check

1. In this section, we discuss how the *ScenTest* tool implements the model-based approach to check a set of test scenarios for mutual inconsistencies. The approach is based on comparing the Promela SUT models using exhaustive simulation to detect differences between them. This approach follows from the fact that in our framework a test scenario is treated as a closed system that includes the test specification (the tester component lifelines) and the SUT (the SUT lifeline). Consequently, the inputs of the SUT are the outputs of the test components and the outputs of the SUT are the inputs of the test components. This duality allows us to state that two test scenarios are inconsistent if their SUT models are inconsistent [8], i.e., the two models contain at least one common trace that leads to states, which have different outputs enabled. In the tool, the comparison approach is implemented following the workflow outlined in Fig. 7. The SUT Model Extractor module builds and combines the models of the SUT lifelines of two test scenarios into a joint Promela model, where one model plays the role of a sender and the other one of a receiver.

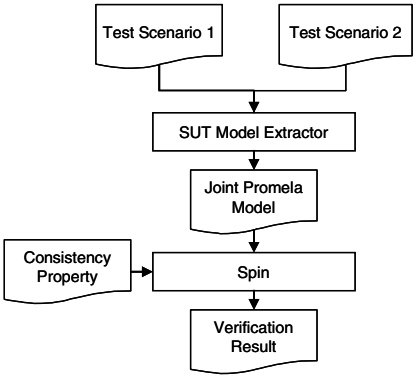


Fig. 7. Workflow for inconsistency checking

The joint model reaches a deadlock whenever the two SUT models that correspond to different test scenarios diverge in their behavior in an improper state. Up to the state where the deadlock occurs, the two models share a common set of traces. The divergence could be either due to different SUT input messages, which does not

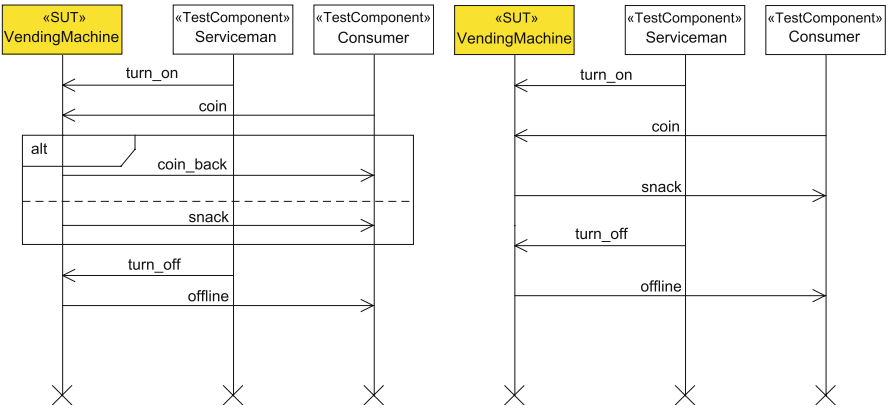


Fig. 8. Two inconsistent test scenarios

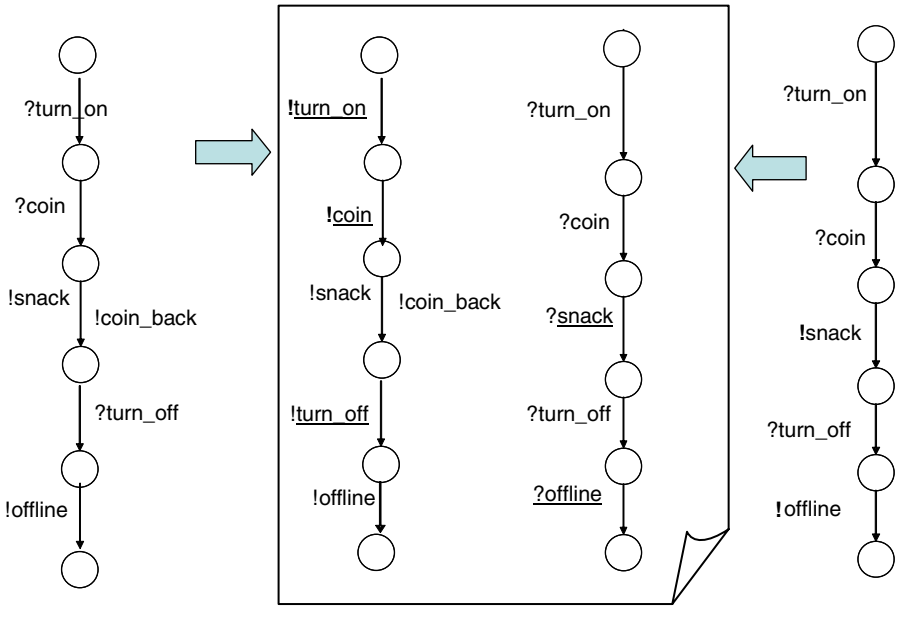


Fig. 9. Detecting a scenario inconsistency in Spin

constitute inconsistency, or due to different SUT output messages. The latter case indicates that some common trace of the two models can be continued with different outputs (Fig. 8), which constitutes an inconsistency. Thus, in order to detect inconsistency we model-check the joint model against a property, which states that each deadlock in the simulation of the joint model is due to divergent receive messages. The violation of the property indicates the inconsistency.

Let us consider the example in Fig. 8 with two different test scenarios; the set of outputs of the vending machine after receiving a *coin* is different in the scenarios. In

the first scenario, the machine either issues *snack* or returns *coin-back* while in the second one only *snack* can be issued. Fig. 9 shows the first step in building the joint Promela model from the two SUT lifelines extracted from the test scenarios, where one process is transformed into a receiver process and other into a sender process. The SUT outputs turn into inputs and vice versa in order to compose two processes into a joint model (underlined message names in Fig. 9). The verification in Spin reveals that the two test scenarios are inconsistent because of the additional SUT output *coin_back*. Our tool is not able to decide, which of the two scenarios is deficient; it is up to the user to decide whether he should remove a redundant SUT output from the first scenario or add an alternative to the second scenario.

5 Conclusions and Outlook

We have presented a scenario-based testing approach and associated tool support. It supports the specification of test scenarios for system integration tests, the generation of executable test scripts, and the test validation and consistency check of test scenarios. Because we rely on Promela as an intermediate representation of the test implementation, it is quite easy to change the generated output language of the test scripts, which is currently Java. Different backend code generators for, for example, TTCN-3 or scripting languages such as Python are possible. The tool is currently completed to be used in first industrial case studies at Siemens.

Although it supports already many MSC/UML2 features to specify test scenarios, the formal specification approach requires further improvements by providing better language facilities to specify data flows within the control flow structure of typical sequence diagrams, e.g. local variables and parameterization of messages and whole test scenarios, variable assignments and local operations on data. Textual extensions to describe such data flow features need to be worked out and integrated into the tool.

Last but not least, support for real-time testing is needed to cover a wider applicability of the tool. For this purpose, an extended test scenario specification approach based on an appropriate representation of timed behavior needs to be worked out. Notwithstanding, we plan to evaluate the efficiency of our MBT approach in the context of industrial projects in the near future.

References

1. Neto, A.D., Subramanyan, R., Vieira, M., Travassos, G.H., Shull, F.: Improving Evidence about Software Technologies – A Look at Model-Based Testing. IEEE Software 24, 10–13 (2008)
2. Haugen, O.: Comparing UML 2.0 Interactions and MSC-2000. In: Amyot, D., Williams, A.W. (eds.) SAM 2004. LNCS, vol. 3319, pp. 69–84. Springer, Heidelberg (2005)
3. OMG UML Specification, http://www.omg.org/spec/UML/2.1.2/
4. Boroday, S., Petrenko, A., Ulrich, A.: Implementing MSC Tests with Quiescence Observation. In: Núñez, M. (ed.) TESTCOM/FATES 2009. LNCS, vol. 5826, pp. 235–238. Springer, Heidelberg (2009)
5. Ulrich, A.: Introducing Model-Based Testing Techniques in Industrial Projects. In: GI-Edition. LNI, Proc. Bd., vol. 106, pp. 29–34 (2007)

6. Baker, P., Bristow, P., Jervis, C., King, D., Mitchell, W.: Automatic Generation of Conformance Tests from Message Sequence Charts. In: Sherratt, E. (ed.) SAM 2002. LNCS, vol. 2599, pp. 170–198. Springer, Heidelberg (2003)
7. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)
8. Boroday, S., Petrenko, A., Ulrich, A.: Test Suite Consistency Verification. In: 6th IEEE East-West Design & Test Symposium (EWDTS 2008), pp. 235–238 (2008)
9. Grabowski, J., Koch, B., Schmitt, M., Hogrefe, D.: SDL and MSC Based Test Generation for Distributed Test Architectures. In: SDL Forum 1999, pp. 389–404 (1999)
10. Mitchell, B.: Lazy Buffer Semantics for Partial Order Scenarios. In: Automated Software Engineering, vol. 14, pp. 419–441. Springer, Heidelberg (2007)
11. Holzmann, G., Peled, D., Redberg, M.: Design Tools for Requirement Engineering. Bell Labs Technical. J. 2, 86–95 (1997)
12. Papesch, M.: Generating Implementations from Formal Specifications: A Translator from Promela to Java. Student thesis no 1826. University of Stuttgart (2002)
13. de Jonge, M.: The SpinJ Model Checker. Master's Thesis. University of Twente
14. ANTLR Parser Generator, `http://www.antlr.org/`

Vidock: A Tool for Impact Analysis of Aspect Weaving on Test Cases

Romain Delamare[1], Freddy Munoz[1], Benoit Baudry[1], and Yves Le Traon[2]

[1] IRISA / INRIA Rennes
{rdelamar,fmunoz,bbaudry}@irisa.fr
[2] LASSY, University of Luxembourg
yves.letraon@uni.lu

Abstract. The addition of a cross-cutting concern in a program, through aspect weaving, has an impact on its existing behaviors. If test cases exist for the program, it is necessary to identify the subset of test cases that trigger the behavior impacted by the aspect. This subset serve to check that interactions between aspects and the base program do not introduce some unexpected behavior. Vidock performs a static analysis when aspects are compiled with a program to select the test cases impacted by the aspects. It leverages the pointcut descriptor to locate the set of methods impacted by aspects and then selects the test cases that can reach an impacted method. This static analysis has to perform over-approximations when the actual point where the aspect is executed can be computed only at runtime and when test cases call polymorphic objects. We measure the occurrence of these assumptions in 4986 projects containing 498 aspects to show they have a limited impact. Then, we run experiments with Vidock on 5 cases studies and analyze the impacts that different types of aspects can have on test cases.

1 Introduction

Aspect-oriented Programming (AOP) [1] encapsulates crosscutting behaviors into single units called *aspects*. The intent of this programming paradigm is to improve the readability, modularity and maintainability of the code. An aspect is composed of an *advice*, which realizes the cross-cutting behavior, and a *pointcut descriptor* (PCD), which describes the points in the program where the advice is woven (called joinpoints).

The cross-cutting concern either keeps the base program's behavior untouched by adding a new behavior, or might modify the existing behavior. An aspect can replace a routine execution when needed, or access the value of a protected data structure at runtime. Thus, the introduction of aspects can modify the control or the data flow of the program in which it is inserted.

When an aspect is woven in a program it is necessary to ensure that it interacts correctly with the program. If the program has already been tested, this implies identifying the test cases that are impacted by the aspect weaving to run them to check the integration of the aspect in the program. A test case is considered impacted if it covers a joinpoint matched by the aspect.

A. Petrenko, A. Simão, and J.C. Maldonado (Eds.): ICTSS 2010, LNCS 6435, pp. 250–265, 2010.

We propose an analysis which does not require executing any test case. The Vidock tool is based on a static analysis performed in two steps. First, the analysis leverages the pointcut descriptor that identifies all the points in the program that are impacted by the aspect weaving. Then, the test cases and the base program are analyzed to determine the points reached by the test cases. If a test case reaches a point that is impacted by an aspect, it is identified as an impacted test case. The advantage of such a static analysis is that the analysis is less time consuming than solutions relying on dynamic analysis.

Because our analysis is static, it has to perform over-approximations on impacted points in two cases. First, a pointcut can declare a dynamic expression. In that case, we statically over-approximate the set of joinpoints that can be matched by the aspect. Second, we have to over-approximate the coverage of test cases in case of a method call on a polymorphic object, since it is not possible to statically know the type of the object. The benefit of these over-approximations is that our impact analysis identifies all the test cases that are impacted. We chose to build Vidock for AspectJ aspects and JUnit test cases.

In section 2 we recall the main constructs in AOP. Section 3, presents the details of the analysis and section 4, discusses the hypotheses for this work: the choice of an impact analysis for programs that compile without the aspects and the over-approximation for dynamic pointcuts and in the presence of polymorphism in the base code. This discussion is based on empirical measurements on 46 open source Java projects that use aspects. Section 5 describes Vidock that implements this automatic analysis as a plug-in for the Eclipse platform. Section 6 presents experiments that apply Vidock on various programs. In section 7 we discuss related work.

2 Illustrating Example

Aspect-Oriented Programming (AOP) is a programming paradigm that separates cross-cutting concerns. AOP encapsulates concerns that crosscut across several modules into *aspects*. In our work, we used AspectJ, a popular and mature implementation of AOP for Java. In this section we present the core concepts of AspectJ through a running example.

2.1 Auction System

We illustrate AOP through examples extracted from an online auction system developed using Java and AspectJ. A user can sell an item by creating an auction with an end date and a minimum price. Other users can place bids on this auction until it closes. The user who placed the highest bid (if any) wins the auction. The system insures that the seller will be paid: a bidder must credit his account before he can bid, and the system checks that the total of all his bid is not greater than the amount available on his account. When the auction closes, money is immediately transferred from the buyer's account to the seller's account.

Two aspects have been added to the base program. The first aspect, Reserve, gives a seller the option to set a secret reserve price. If the highest bid does not

```
1  public privileged aspect Reserve {
2    private int Auction.reservePrice = 0;
3    pointcut closeOpen(Auction a):
4      execution(void Open.close(Auction));
5    void around(Auction auction): closeOpen(auction) {
6      if(auction.reservePrice>0) { ... }
7      else proceed(auction);
8    }
9  }
```

Listing 1.1. The Reserve aspect

```
1  public aspect AltBid {
2    pointcut getAmount(): call(int Bid.getAmount())
3      && cflow(execution(* *.close(Auction)))
4    int around(): getAmount() {
5      ...
6      int res = secondBid.getAmount()*11/10;
7      return res;
8    }
9  }
```

Listing 1.2. The AltBid aspect

match the reserve price then the item is not sold. The second aspect, AltBid, changes the way the final price is computed. In the base program the price an item is sold is the amount of the highest bid. With this aspect, the price is the amount of the second highest bid plus 10%. Listings 1.1, 1.2 respectively show an excerpt of the Reserve and AltBid aspects.

2.2 Aspect Oriented Programming: The Case of AspectJ

AOP encapsulates crosscutting behaviors into single units called *aspects*. An aspect itself is composed of several units realizing the crosscutting behavior, these units are called *advices*. Aspects also provide pointing elements that designate well defined points in the program execution or structure where the crosscutting behavior is executed. The pointers are called *pointcut descriptors* (PCD) and the execution points *joinpoints*.

In AspectJ, a PCD is defined as a combination of names and keywords. Names are used to match specific places in the base program. For instance, the name `void Open.close(Auction)` in listing 1.1 (line 4) matches the method `close` which returns type `void`, receives a parameter of type `Auction` and is declared in the class `Open`. Names can contain wild-cards that enlarge the number of matches. The wild-cards `*` in the name `* *.close(Auction)` of listing 1.2 (line 3) are used to specify that the method can be declared in any class (`* *.close`) and have any return type (`* *.close`). Keywords define when the places matched by names will be intercepted. For instances, in the PCD `execution(* *.close(Auction))` the keyword `execution` indicates that the execution of the matched places (method `close(Auction)`) will be captured. The combination of names and keywords is referred as *expression*. PCDs can be composed of multiple expression joint by the logic operators `&&`

(conjunction) and || (disjunction). For instances the PCD in listing 1.2 (lines 2-3) is the conjunction of two expressions.

PCDs can also point joinpoints that occur dynamically during the execution of the program. For instance, the PCD of listing 1.2 (lines 2-3) is composed of two expressions. The first (line 2) intercepts the calls to `getAmount()`. The second (line 3) constrains the interception to the calls occurring inside the control flow of the execution of `close(Auction)`. To know whether a call to `getAmount()` occurs during the execution of `close(Auction)`, the program must be running. Therefore, there is no mean to statically know the exact occurrence of these joinpoints. We refer to these joinpoint as *dynamic joinpoints* and to a PCD pointing these points a *dynamic-PCD*.

AspectJ extends the Java syntax to allow developers to implement advices as natural as possible. Therefore, advices can be seen as routines that are executed at some point. Typically AspectJ advices are bounded to a PCD designating the points where they will be executed. The advice in listing 1.1 (lines 5-8) is bounded to the PCD `closeOpen`, therefore, it will be executed during the execution of `close(Auction)`. AspectJ provides three different kind of advices *Before*, *After* and *Around* indicating the moment when they are executed. Before and After advices are executed respectively just before / after reaching the joinpoints designated by the PCD. Around advices are a special type of advice, they are executed instead of the designated joinpoints. Besides, by invoking the special operation `proceed` they can execute the captured joinpoint. For instance, the advice in listing 1.1 (lines 5-8) executes the captured joinpoint only given a special condition (line 7), otherwise it replaces its execution (line 6).

3 Selection of the Impacted Test Cases

To determine which test cases of the base program are impacted by the aspects, we statically analyze the Java base program, the aspects and the test cases. We analyze the PCD to select the set of methods impacted by an aspect. Then we can statically analyze the test cases to determine if they are impacted.

Figure 1 presents an overview of the analysis process. Initially, Vidock gathers information from the AJDT and the Spoon [2] tools. Then, driven by each test case description, it constructs a static call graph (SCG) – one for each test case (a,b). Besides Vidock selects all the methods having at least an aspect weaved, namely impacted methods (c). Finally, Vidock calculates the impacted test cases by intercepting the impacted methods with the SCG nodes (d).

3.1 Selecting Methods

The first part of our analysis deals with the selection of the methods impacted by the aspect weaving. The method selection is divided in two stages, (1) we gather a set of joinpoints pointed by a PCD (using AJDT) and then (2) we calculate the set of methods related to those joinpoints (using Vidock). However, the selection of the methods *impacted* by aspects is not straightforward. The first issue we

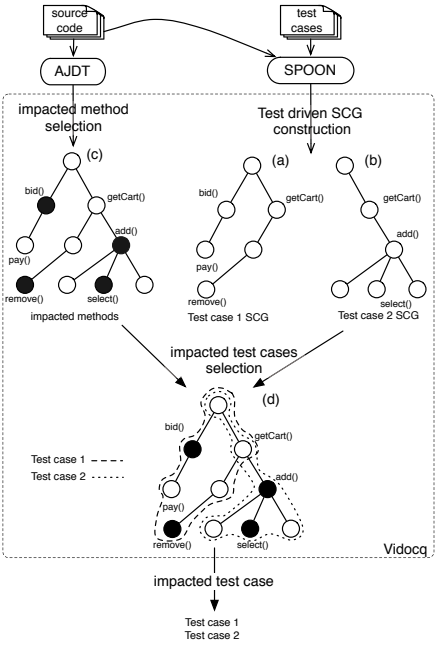


Fig. 1. Process overview

must deal with is the presence of dynamic-PCDs. Because dynamic-PCDs cannot be computed statically, it is not feasible to know the set of methods they point.

Expressions pointing dynamic joinpoints (with keyword such as `if`, `cflow` and `cflowbelow`) are used to constrain the amount of joinpoints pointed by static expressions. To deal with dynamic-PCDs statically, AJDT performs an over-approximation of the selected joinpoints. Such over-approximation consists in removing the dynamic expressions from the PCD. The resulting PCD points a set of joinpoints that contains those pointed by its dynamic version.

Formally, this over-approximation can be described as follows. Let P be a program, C be the set of all the PCDs declared in P and J be the set of all the joinpoints in a program P. Let $f_{pc} : C \rightarrow \mathcal{P}(J)$ be a function that returns the set of all the joinpoints pointed by a PCD. Let $f_{over} : C \rightarrow \mathcal{P}(J)$ be an over-approximation function. Given a PCD, it removes its dynamic part (if any) and gives the set of all the joinpoints it points.

$$\forall c \in C, f_{pc}(c) \subseteq f_{over}(c)$$

In the best case scenario, C contains no dynamic-PCDs and then $f_{over} = f_{pc}$.

Figure 2 illustrates this over-approximation. In the figure, the dynamic-PCD `getAmount()` (A) points the set of joinpoints $f_{pc}(A) = J_d$. However, it is unfeasible to compute J_d statically. The static version of `getAmount()` (B) points a larger set of joinpoints $f_{pc}(B) = J_s$ that contains those pointed by A. AJDT

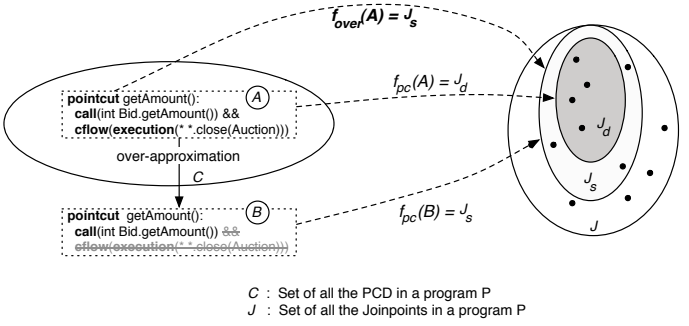


Fig. 2. Dynamic-PCD pointed joinpoints over-aproximation

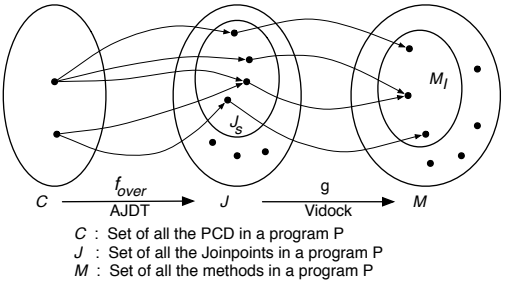


Fig. 3. Schematic view of the definition of an *impacted method*

performs this over-approximation directly on the dynamic-PCD. This results in a set of joinpoints $f_{over}(A) = J_s$, which contains J_d and is also computable.

The second issue we must deal with is the actual selection of the *impacted* methods. An impacted method is any method related to a joinpoint that is pointed by a PCD. It can be specified as follows.

Let M be the set of all the methods in a program P. Let $g : J \rightarrow M$ be a function that, for a given joinpoint, gives the method where it is located.

$$g(j) = \begin{cases} \text{the caller} & \text{if } j \text{ is a call joinpoint} \\ \text{the executed method} & \text{if } j \text{ is an execution joinpoint} \\ \text{the constructor} & \text{if } j \text{ is an initialization joinpoint} \\ \text{the accessor method} & \text{if } j \text{ is a field access joinpoint} \end{cases}$$

Then, given g, the set $M_I \subseteq M$ of impacted methods is defined by:

$$M_I = \{m \in M | \exists c \in C, \exists j \in f_{over}(c), g(j) = m\}$$

Figure 3 depicts how the set M_I of impacted methods is obtained from the combination of f_{over} and g. First f_{over} provides the set of joinpoints J_s from

```
1  public class TestAuction {
2        @org.junit.Test
3        public void testClose() {
4              Auction a = new Auction(..);
5              a.open();
6              a.bid(user1,30);
7              a.bid(user2,50);
8              a.close();
9              assertTrue(a.isClosed());
10             assertEquals(user2,a.getBuyer());
11       }
12 }
```

Listing 1.3. An example of JUnit test case for the `Auction` class

the PCD `getAmount()` (A), $J_s = f_{over}(A)$. This part is performed by AJDT. Then, the set $M_I = g(J_s)$ is determined by Vidock, in which g is implemented. In this case $g(j_1) =$ `Open.open()`, $g(j_2) = g(j_3) =$ `Bid.finalize` and $g(j_4) =$ `Bid.bid()`. These methods are impacted.

3.2 Impacted Test Cases

The second step of the impact analysis consists in detecting the test cases that are impacted by the introduction of aspects. We want to know if a test case can reach an impacted method or not, to know if its control or data flow can be modified. First, we need to know which methods are reachable by each test case. To do so we build a static call graph for each test case.

Static call graph. A static call graph is a triple (V, E_c, E_p) where:

- V is the set of vertices for methods, each method is represented by only one vertex.
- $E_c \subseteq V \times V$ is a set of edges that represent method calls. If $(m1, m2) \in E_c$, it means that the method represented by $m1$ explicitly calls the method represented by $m2$.
- $E_p \subseteq V \times V$ is a set of edges that represents potential method calls. If $(m1, m2) \in E_p$, then the method represented by $m1$ potentially calls the method represented by $m2$.
- $E_c \cap E_p = \emptyset$.

Explicit and Potential Calls. A method $m1$ explicitly calls the method $m2$ if $m2$ is explicitly invoked in the source code of $m1$. The method $m1$ potentially calls $m2$ if it calls a method overridden by $m2$. We call it a *potential* call because we do not know statically which method is actually executed. The `Auction` class uses the state pattern – as illustrated by Figure 5 –, so when `Auction.close` is called, the method `close` of the current `State` object is called, but the method that is actually executed could be any of the overriding methods of `Pending`, `Open`, `Closed` and `Cancelled` (the four possible states for an auction).

Listing 1.3 shows an example of a JUnit test case. The test case tests the `close` method of class `Auction`. First an auction is created and two users

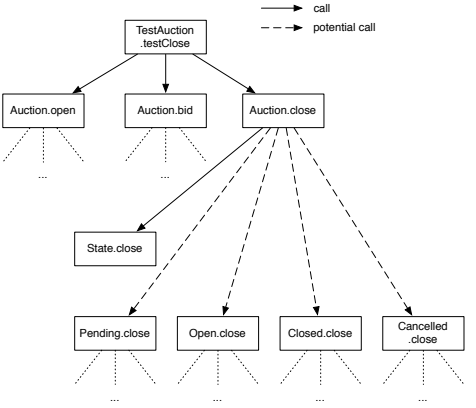


Fig. 4. The static call graph of the test case from Figure 1.3

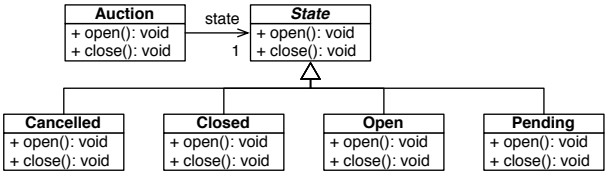


Fig. 5. Class diagram illustrating the implementation of the state design pattern

place a bid. Then the auction is closed and the assertions check if the auction is actually closed and if the buyer is the highest bidder. Figure 4 shows an excerpt of the static call graph of this test case, with the calls and potential calls.

Vidock uses Spoon, a tool for static analysis of Java programs, to build the static call graphs. It provides a visitor pattern to explore the abstract syntax trees (AST) of the java source files. This allows the extraction of the static call graph with only the call edges (no potential calls). To obtain the potential calls we must explore the ASTs of the whole system looking for methods overriding the method targeted by a call. Potential calls are also extracted using Spoon. An inheritance tree is built to obtain the subclasses of a class. Then, for each call, we check if any of the subclasses of the targeted method's class overrides the targeted method. For each overriding methods found, a potential call is created with the source of the explicit call as source and the overriding method as target.

Impacted test case. An impacted test case is a test case that can potentially execute an impacted method. Let t be a test case, S its static call graph, T_I the set of impacted test cases and M_I the set of impacted methods:

$$t \in T_I \Leftrightarrow \exists P = t, v_1, \ldots, v_n \text{ a path in } S, v_n \in M_I$$

```
1  boolean impacted(Vertex v) {
2    if onImpactedPath.contains(v) or impactedMethods.contains(v.method) then
3      return true
4    end
5    beingVisited.add(v)
6    for each v' in v.next do
7      if not beingVisited.contains(v') then
8        if impacted(v') then
9          onImpactedPath.add(v')
10         beingVisited.remove(v)
11         return true
12       end
13     end
14   end
15   visited.remove(v)
16   return false
17 }
```

Listing 1.4. The algorithm determining whether a test case is impacted or not

A test case is impacted if, from the root of its static call graph, it is possible to reach an impacted method. Once the static call graph of a test case has been built we can determine whether it is impacted or not by using the algorithm on Listing 1.4. This algorithm returns true if the argument vertex represents an impacted method or can reach such a vertex. If we call this algorithm with the vertex representing a test case it returns true if it is impacted.

This algorithm implements a depth-first search on a graph and thus has a time complexity in $\mathcal{O}(|V| + |E_c| + |E_p|)$, where V is the set of vertices of the graph, and $E_c \cup E_p$ the set of edges, with $E_c \cap E_p = \emptyset$.

The `impactedMethods` attribute contains all the impacted methods. On line 2, the algorithm checks if the current vertex represents an impacted methods, and returns true if it is the case.

The `v.next` is the set $\{v' | (v, v') \in E_c \cup E_p\}$. On line 6, the loop calls the algorithm on the vertices that can be reached with a call or potential call edge.

A vertex that can reach a vertex representing an impacted method is said to be on an impacted path. The attribute `onImpactedPath` contains such vertices and is used to optimize the algorithm. Several test cases will most likely execute the same methods, so if we know from a previous execution of the algorithm that a vertex is on an impacted path we can stop the algorithm and return true.

Computing the complete static call graphs of all test cases can be very time consuming. This is why we build the static call graph *on the fly*. Actually the set `v.next` is only built when needed, so if we quickly find an impacted method, we do not have to find all the potential calls of the static call graph. If the test case is not impacted we cannot avoid building the complete static call graph.

The notion of potential calls may lead to an over-approximation. When there are potential calls, several of them are not executed at all at runtime, but we cannot statically remove them. So if one of this *false* potential calls can reach an impacted method, then the test case can be selected as impacted without being actually impacted. Note that this over-approximation leads to *false* positive but not to *false* negative (an impacted test case not detected as impacted).

For instance, in the example of Figure 4, the method `Open.close` is impacted by the `Reserve` aspect, and this method is actually executed so the test case is actually impacted (and detected as such). But suppose that the aspect does not impact `Open.close` but `Pending.close`, then the test case would not be actually impacted but still would be detected.

4 Validation of Hypotheses

Our proposal is based on three hypotheses. First, we assume that the base program can be tested in isolation, before the aspect weaving. Thus we assume that the base program can compile without the aspects – which is only possible if the base program is not aware of the aspects. Second, the impact analysis is based on two over-approximations. (1) The set of impacted methods is over approximated in case of dynamic-PCDs, and (2) the static call graph for a test case is over approximated in presence of overridden methods. To evaluate the impact of these hypotheses on the proposed analysis, we have studied their occurrence over a set of 46 open source aspect-oriented projects. This study is detailed in a previously published paper [3].

We have selected the projects according to the following criteria: (1) Project implemented in Java/AspectJ, (2) project source code publicly available, (3) project compilable using the AspectJ compiler version 1.5, and (4) project size of at least 10 classes and 1 aspect. The 46 projects fulfilling these criteria were gathered from open source repositories. We started our search at sourceforge.net, the most popular open source repository in Internet. Out of 74 aspect-oriented projects, only 29 fulfilled our criteria. Then, we continued gathering projects by inspecting other repositories by using the Google™ code search engine. Out of 2000 files, equivalent to 31 projects, only 17 were fulfilling our selection criteria. Finally, we successfully gathered 46 open source aspect-oriented projects of size ranging from 10 to 913 classes and 1 to 64 aspects.

4.1 Hypothesis 1: Project Compilation

The first hypothesis we have formulated is that aspect-oriented projects can compile without aspects. To confirm whether this hypothesis occurs in real aspect-oriented projects we have studied its occurrence over the 46 open source projects.

To know the amount of projects compiling without aspects, we have disable the aspectJ capability of each project. After re-compiling each project without aspect support, we have obtained the following results: Out of 46 aspect-oriented projects, 30 are compilable in the absence of aspects. That is, the 65% of the projects compile without the crosscutting functionality added by aspects.

The results obtained in this experience endorse our hypothesis. The percentage of projects effectively compiling without aspects is more than half of the total amount of projects. Thus, most of the projects are analyzable using our tool. Moreover, an important portion of the compilable projects range form large and medium size (between 3000 and 80900 LOC).

4.2 Hypothesis 2: Low Method Overriding

The second hypothesis we formulated is that the method overriding in general have a low frequency. To determine the frequency of overriding, we have studied the method overriding practice on the 46 open source projects.

We have used the Metrics [1] eclipse-plugin to analyze the java sources in each project project. The results obtained from this analysis are the following: Out of 58184 methods scattered in 8052 classes (total of 46 projects), only 3295 are overridden by subclasses. On average there are 1265 methods per projects and only 72 of them are overridden. That is, only a 6% of the total amount of methods (an average of 5,6% per project, range from 0% to 14% per project) were overridden, leaving 94% of them with no further change by subclasses.

These results widely support our hypothesis. The 6% of overridden methods is negligibly small. The over-approximation we applied in the case of inheritance overriding methods is reasonable, given the small amount of overridden methods. A common practice when building test cases is to call directly a method in a subclass instead of the method in a superclass. This allows the tester to predict the expected behavior and build a precise oracle. Thus, the impact of over approximation in the test case static call graph should be small.

4.3 Hypothesis 3: Low Usage of Dynamic-PCDs

The third and final hypothesis we formulated for our analysis is that the dynamic-PCDs are rarely used. To determine if dynamic-PCDs are used, we have studied the taxonomy of the PCDs declared in the aspects of the 46 open source projects.

We have analyzed each project with a custom tool [2] inspecting the PCDs definition and specifically their taxonomy. We have obtained the following results: Out of 1145 PCDs declared on 498 aspects (total of 46 projects), only 206 contained dynamic expressions (`cflow`, `cflowbelow`, `if`). That is, only a 17% of the total amount of PCDs contained dynamic expressions. Moreover, the dynamic-PCDs were present only in 15 out of 46 projects.

These results confirm our hypothesis. The 17% of dynamic-PCDs reveals that they are rarely used in real life open-source projects. Moreover their occurrence only in the 33% of the projects advocates that an over-approximation for impact analysis is reasonable. In section 6.2 we observe the effects of these hypotheses on actual impact analyses.

5 Implementation

Vidock has been implemented as a plug-in for the Eclipse IDE and is available on the internet[3]. As seen previously, the tool relies on AJDT[4] to resolve the

[1] http://metrics.sourceforge.net/
[2] `http://contract4aj.gforge.inria.fr/analysis`
[3] `http://www.irisa.fr/triskell/Softwares/protos/vidock/`
[4] `http://www.eclipse.org/ajdt/`

Table 1. Results of our experiments: added aspects and their impact on the test cases (*: actually impacted test cases)

Example	TC	Aspect	Impacted TC	Precision	Recall
Introduction	7	Cloneable	0	NA	NA
		Comparable	0	NA	NA
		Hashable	0	NA	NA
Bean	7	BoundPoint	6 (85.7%)	100%	100%
Telecom	33	Timing	3 (9.1%)	100%	100%
		Billing	3 (9.1%)	100%	100%
Tetris	45	TestAspect	8 (17.8%)	100%	100%
		NewBlocks	7 (15.6%)	100%	100%
		Counters	2 (4.4%)	100%	100%
		Levels	2 (4.4%)	100%	100%
		All aspects	17 (37.8%)	100%	100%
Auction	306	Reserve	11 (3.6%) 7* (2.3%)	63.6%	100%
		AltBid	49 (16.0%) 39* (12.7%)	79.6%	100%
		Log	306 (100%)	100%	100%
		Reserve + AltBid	49 (16.0%) 39* (12.7%)	79.6%	100%

PCDs and obtain the matched joinpoints and on Spoon [2] to obtain an abstract syntax tree of the system that allows us to build the static call graphs.

After each compilation of an AspectJ project, the plug-in applies the process described in section 3, and for each test cases that is detected as impacted a warning is reported and appears as a regular Eclipse warning. At the compilation, AJDT resolves the PCDs so we can obtain the matched joinpoints, as explained on figure 1. Then we use spoon to detect the JUnit test cases within the classes of the project. If a test case is detected as impacted we use Spoon to report a warning.

6 Case Studies

To validate our approach we have experimented our analysis on several examples. Some of them are distributed with AspectJ (the *Introduction*, *Bean* and *Telecom* examples), one is a classic project freely available on the Internet (*Tetris*) and the last one has been developed by our group (*Auction*).

6.1 Description of the Examples

The *Introduction* example has a unique class `Point` that encapsulate the two-dimensions coordinates of a point (using either Cartesian or polar coordinate system). Three aspects are written for that class, *Cloneable*, *Comparable* and

Hashable. All these aspects declare that `Point` implements a new interface and adds inter-type definitions to add the implementation of new methods.

The *Bean* example also implements a `Point` class. The only aspect declares that `Point` implements `Serializable` and adds bound properties. Listeners can be registered with an observer design pattern and an advice is woven around each setter of `Point` to notify the listeners.

The *Telecom* example simulates telephonic communications and handles local and long distance calls as communications with more than two interlocutors. The `timing` aspect calculates the duration connection and the customer's cumulative connection time. The `billing` aspect relies on `timing` and calculates the cost of a connection.

The *Tetris* example is an implementation of the classic video game that has been developed by Gustav Evertsson[5]. The `TestAspect` aspect traces the images that are loaded and prints their name in the console. The `NewBlocks` aspect adds new kind of blocks to the game. The `Counters` aspect counts the deleted lines and prints them on the game layout. The `Levels` aspects adds a level system to the game: each time a certain number of lines has been deleted the level is increased and the blocks fall faster.

Finally the *Auction* example is an implementation of an online auction system where users can buy or sell items. The implementation uses the command and the state design pattern [4]. The command pattern is used by the server to process the instruction received by the clients. The state pattern is used to represent the state of an auction. There are 4 different states (pending, open, closed and cancelled). The system (without the test cases) has 1382 lines of code, 41 classes. The aspects that have been added are those presented in section 2.1.

6.2 Results

For each example, test cases have been written using statement coverage as a minimum criterion. These test cases allowed us to detect and fix several errors. For the auction system we validated the program. In the *Introduction* example of AspectJ we detected several bugs with the handling of polar coordinates.

Then the aspects have been added to the base program, alone if possible, with the depending aspects otherwise. After each weaving the impact analysis has been performed on the test cases. The results are summarized in Table 1 and are discussed below. In the following, we first discuss the extreme results – 0% or 100% of impacted test cases –, then we discuss the results of most cases.

For the *Introduction* example none of the test cases are impacted by the aspects. This result happens because the aspects do not add any advice so none of the test cases are impacted. Also, the methods introduced by the aspects are not executed by the test cases for the base class.

The `log` aspect of *Auction* impacts all the test cases but this is a particular case: the aspect is woven everywhere in the code, before each method execution.

[5] `http://www.guzzzt.com/coding/aspecttetris.shtml`

When the system has few classes, the aspects are more likely to impact a large part of the test cases. In the *Bean* example the `BoundPoint` impacts 6 out of 7 test cases. This aspect defines only one advice which is woven around methods that are called by many other methods of `Point`. This particular case, where a few impacted methods are called by almost every method in the system will more likely happen in a small system.

Apart from these borderline cases, less than 20% of the test cases are impacted, and in some cases less than 5%. These results show that, in our examples, our analysis can save execution time and help the evolution of the test cases. Execution time is saved by avoiding the execution of the non-impacted test cases. This analysis helps the evolution because the programmer can focus on the impacted test cases to determine whether they should be modified or not as the non-impacted test cases can be ignored.

To estimate the effects of the over-approximation, we have computed the precision and recall for each aspect. Each test case was manually checked to see if it was impacted. The recall is always 100%, which means that are no *false negative* (i.e., impacted test cases not selected). In most cases the precision is 100%, which means that there are no *false positive* (i.e., selected test cases that are not impacted). With the `Reserve` aspect, the precision is only 63.6%, but there are only four false positives. In the other two cases where there are false positives, the precision is good (79.6%). These results show that the over-approximation has little effect and no impacted test cases are missed.

When several aspects are added simultaneously, the number of impacted cases is not always the sum of the test cases impacted independently by each aspect. Table 1 shows the results when all the aspects of *Tetris* are added and when `Reserve` and `AltBid` are added in *Auction* – note that in *Telecom*, `Billing` relies on `Timing`, so when `Billing` is added both are added.

This shows that the number of impacted test cases does not necessarily grow with the number of aspects in the system. In *Tetris*, the number of test cases impacted by all the test cases (17) is a bit less than the sum of the test cases impacted by each aspect (19). In *Auction*, all the test cases impacted by `Reserve` are also impacted by `AltBid`, so the number of test cases impacted by the two aspects is equal to the number of test cases impacted by `AltBid`.

7 Related Work

Several work have proposed static analysis to evaluate the impact of changes in a program. Vokolos *et al.* [5] have introduced a tool for regression test selection on C programs. It selects test cases based on a textual comparison of the source files. Rothermel *et al.* [6] have presented an algorithm for regression testing that compares control flow graphs (CFGs) and thus is less dependent on the programming language. It requires a CFG for each version of the program and coverage informations for each test case. If a test case covers a part of the initial CFG that differs in the new CFG then the test case is selected. This algorithm has been adapted to Java programs by Harrold *et al.* [7], and then to AspectJ by

Xu *et al.* [8]. Zhang *et al.* [9] have developed a change impact analysis tool for AspectJ, Celadon. It detects atomic changes in the source code of the program by comparing the ASTs of the different versions. Then call graphs of the test cases are produced to detect the test cases affected by the change.

These static analysis are close to the analysis of Vidock but they differ because they evaluate the impact of a change whereas Vidock focuses the impact of the introduction of aspects. Vidock relies on the PCD to locate the impacted methods, whereas other techniques compare two models of the program (e.g., ASTs, CFGs) to detect which parts have changed. This results in complex graph comparison algorithms: according to Rothermel *et al.* [6], their algorithm is quadratic in the worst case. Also, the algorithm of Rothermel *et al.* requires the coverage information of the test cases on the previous version of the program. Vidock is more focused, but in the specific case of aspect weaving, proposes an efficient solution for change localization.

Ren *et al.* [10] have proposed a change impact analysis tool for Java programs. This work is different because the goal is to compute the subset of changes that affect a test cases, instead of computing the subset of test cases that are affected by all the changes. This approach uses a dynamic analysis, to detect at runtime the changes that are executed by a test case. The changes are detected statically.

8 Conclusion

In this paper we have presented Vidock, a tool for the impact analysis of aspect weaving on test cases. It performs a static analysis that identifies the subset of test cases that are impacted by the aspect weaving.

To ensure that we do not miss impacted test cases, the analysis over-approximates the set of impacted test cases in the case of dynamic poincut descriptors and polymorphism. Thus we can have *false positive* (test cases detected as impacted that are actually not impacted) but no *false negative* test cases (impacted test cases that are not detected).

To validate the ability of Vidock to detect impacted test cases and the effect of over approximation on the Vidock's results, we have performed two types of studies. First, we have experimented Vidock on 5 systems. The experiments showed that in most cases only a few test cases are impacted by the aspects. They also showed that the over-approximations have a minimal effect on the results. Second, we have studied 46 open-source AspectJ projects to evaluate the occurences of dynamic PCDs and method overriding. We observed that 65% of the projects can compile without aspects, and thus the *base program* can be tested in isolation; 6% of the methods are overridden and 17% of the PCDs are dynamic, so the effect of over-approximations should be minimal in general.

Vidock has been developed as a plug-in for the Eclipse IDE. It performs the impact analysis automatically after each compilation and reports a warning for each impacted test cases. This tool is available for download.

In future work we want to investigate aspects classifications such as the one proposed by Rinard *et al.* [11] or Munoz *et al.* [12], to assist the evolution of

impacted test cases. Munoz *et al.* propose a classification based on the impact of an aspect on the control flow or data flow. Using this information, we can know, for example, that test cases impacted by an aspect that does not modify the control and data flow must be executed and must pass. If these test cases fail, we can locate the fault in the aspect that has been introduced. Also, better understanding the impact of an aspect will probably help the evolution of the test cases. If an aspect is independent of the base code – i.e. neither the aspect nor the base program may write a field that the other may read or write – the oracle of the impacted test cases most likely will need to be augmented but the existing assertions should remain unchanged.

References

1. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley, Reading (2005)
2. Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA (2006), http://spoon.gforge.inria.fr
3. Munoz, F., Baudry, B., Delamare, R., Le Traon, Y.: Inquiring the usage of aspect-oriented programming: an empirical study. In: ICSM 2009: Proceedings of the 25th International Conference on Software Maintenance (2009)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
5. Vokolos, F.I., Frankl, P.G.: Pythia: a regression test selection tool based on textual differencing. In: 3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems, pp. 3–21. Chapman & Hall, Ltd., London (1997)
6. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. 6(2), 173–210 (1997)
7. Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A.: Regression test selection for java software. In: OOPSLA 2001: Proceedings of the 16th conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 312–326 (2001)
8. Xu, G., Rountev, A.: Regression test selection for AspectJ software. In: ICSE 2007: Proceedings of the 29th International Conference on Software Engineering, pp. 65–74 (2007)
9. Zhang, S., Gu, Z., Lin, Y., Zhao, J.: Celadon: a change impact analysis tool for aspect-oriented programs. In: ICSE Companion 2008: Companion of the 30th international conference on Software engineering (2008)
10. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of Java programs. In: OOPSLA 2004, pp. 432–448 (2004)
11. Rinard, M., Salcianu, A., Bugrara, S.: A classification system and analysis for aspect-oriented programs. SIGSOFT Softw. Eng. Notes 29(6), 147–158 (2004)
12. Munoz, F., Baudry, B., Barais, O.: Improving maintenance in aop through an interaction specification framework. In: ICSM 2008: 24th International conference on Software Maintentance, Beijing, China. IEEE Computer Society, Los Alamitos (2008)

Author Index

Aarts, Fides 188
Alikacem, El-Hachemi 236
Ano, Shigehiro 205
Arcuri, Andrea 63, 95

Baudry, Benoit 250
Bjørner, Nikolaj 47
Bogdanov, Kirill 126
Boroday, Sergiy 236
Briand, Lionel 63, 95

da Veiga Cabral, Rafael 16
de Almeida, Eduardo Cunha 174
de Halleux, Jonathan 142
Delamare, Romain 250
Derrick, John 126

Falcone, Yliès 30
Fernandez, Jean-Claude 30

Gladisch, Christoph D. 158
Gonzalez-Sanchez, Alberto 79
Gross, Hans-Gerhard 79

Hallal, Hesham H. 236
Harman, Mark 142
Hasegawa, Toru 205
Heitmeyer, Constance 15
Hemmati, Hadi 63

Iqbal, Muhammad Zohaib 95

Jéron, Thierry 30
Jonsson, Bengt 188

Lakhotia, Kiran 142
Le Traon, Yves 174, 250

Marchand, Hervé 30
Marynowski, João Eugenio 174
Meinke, Karl 221
Mounier, Laurent 30
Munoz, Freddy 250

Niu, Fei 221

Paris, Javier 126
Pérez Lamancha, Beatriz 111
Piel, Éric 79
Polo Usaola, Macario 111
Pozo, Aurora 16

Schieferdecker, Ina 1
Shinbo, Hiroyuki 205
Sunyé, Gerson 174
Suzuki, Kenji 205

Tagami, Atsushi 205
Tillmann, Nikolai 142

Uijen, Johan 188
Ulrich, Andreas 236

Valduriez, Patrick 174
Veanes, Margus 47
Vergilio, Silvia Regina 16

Walkinshaw, Neil 126