EPTCS 80

Proceedings of the

7th Workshop on

Model-Based Testing

Tallinn, Estonia, 25 March 2012

Edited by: Alexander K. Petrenko and Holger Schlingloff

Published: 27th February 2012
DOI: 10.4204/EPTCS.80
ISSN: 2075-2180
Open Publishing Association

Preface

This volume contains the proceedings of the Seventh Workshop on Model-Based Testing (MBT 2012), which was held on March 25, 2012 as a satellite workshop of the European Joint Conferences on Theory and Practice of Software (ETAPS 2012).

The first workshop on Model-Based Testing (MBT) in this series took place in 2004, in Barcelona. At that time MBT already had become a hot topic, but the MBT 2004 workshop was the first event devoted exclusively to this topic. Since that time the area has generated enormous scientific interest, and today there are several specialized workshops and more broad conferences on software and hardware design and quality assurance covering model based testing. For example, in 2011 ETSI has started the MBT-UC (model-based testing user conference) series that considers problems of the application and effective use of MBT in business. Still, the MBT series of workshops offers a unique opportunity to share new technological and foundational ideas, and to bring together researchers and users of model-based testing to discuss the state of the theory, applications, tools, and industrialization.

Model-based testing has become one of the most powerful system analysis tools, where the range of possible applications is still growing. Currently, we see the following main directions of MBT development.

- Integration of model-based testing techniques with various other analysis techniques; in particular, integration with static analysis and other verification methods.
- Use of new notations and new kinds of models (like models of business processes in BPMN or BPL) along with the elaboration of approaches based on usual programming languages and specialized libraries (like CodeContracts for .NET).
- Integration of MBT tools into continuous development environments and application lifecycle management toolchains.

The contributions in this volume reflect these current trends. To quote just a few contributions, Danel Ahman and Marko Kääramees show how to use I/O automata for online test generation. Bernhard Aichernig and Elisabeth Joebstl report on the combination of reachability and refinement checking. Maximiliano Cristia and Claudia Frydman apply SMT solvers to the test generation problem. Dirk Richter and Christian Berg investigate code coverage metrics for ISO-C. Teemu Kanstrn and Olli-Pekka Puolitaival use domain-specific languages for model-based testing.

In 2012 the “industrial paper” category was added to the program, and two industrial papers were accepted by the program committee. Dimitry Polivaev use mind maps in rule-based test generation. Yevgeny Gerlits and Alexey Khoroshilov report on a case study with a safety-critical realtime controller.

We would like to thank the program committee members and all reviewers for their work in evaluating the submissions. We also thank the ETAPS 2012 organizers for their significant assistance in the preparation of the workshop.

Alexander K. Petrenko and Holger Schlingloff, February 2012.

Program committee

- Bernhard Aichernig (Graz University of Technology, Austria)
- Jonathan Bowen (University of Westminster, UK)
- Mirko Conrad (The MathWorks GmbH, Germany)
- John Derrick (University of Sheffield, UK)
- Bernd Finkbeiner (Universitt des Saarlandes, Germany)
- Patrice Godefroid (Microsoft Research, USA)
- Wolfgang Grieskamp (Google, USA)
- Ziyad Hanna (Jasper Design Automation, USA)
- Philipp Helle (EADS, Germany)
- Antti Huima (Conformiq Software Ltd., Finland)
- Mika Katara (Tampere University of Technology, Finland)
- Alexander S. Kossatchev (ISP RAS, Russia)
- Andres Kull (Elvior, Estonia)
- Mounier Laurent (VERIMAG, France)
- Bruno Legeard (Smartesting, France)
- Bruno Marre (CEA LIST, France)
- Alexander K. Petrenko (Institute for System Programming Russian Academy of Sciences, Russia)
- Alexandre Petrenko (Computer Research Institute of Montreal, Canada)
- Fabien Peureux (University of Franche-Comt, France)
- Holger Schlingloff (Fraungofer FIRST, Germany)
- Nikolai Tillmann (Microsoft Research, USA)
- Jan Tretmans (Embedded Systems Institute, The Netherlands)
- Nina Yevtushenko (Tomsk State University, Russia)

Additional reviewers

- Maxim Gromov
- Antti Jääskeläinen
- Elisabeth Jöbstl
- Markus Rabe
- Matti Vuori

Table of Contents

Model-Based Security Testing

Ina Schieferdecker
Fraunhofer FOKUS
Berlin, Germany
Freie Universitaet Berlin
Berlin, Germany
ina.schieferdecker@fokus.fraunhofer.de

Juergen Grossmann
Fraunhofer FOKUS
Berlin, Germany
juergen.grossmann@fokus.fraunhofer.de

Martin Schneider
Fraunhofer FOKUS
Berlin, Germany
martin.schneider@fokus.fraunhofer.de

Security testing aims at validating software system requirements related to security properties like confidentiality, integrity, authentication, authorization, availability, and non-repudiation. Although security testing techniques are available for many years, there has been little approaches that allow for specification of test cases at a higher level of abstraction, for enabling guidance on test identification and specification as well as for automated test generation.

Model-based security testing (MBST) is a relatively new field and especially dedicated to the systematic and efficient specification and documentation of security test objectives, security test cases and test suites, as well as to their automated or semi-automated generation. In particular, the combination of security modelling and test generation approaches is still a challenge in research and of high interest for industrial applications. MBST includes e.g. security functional testing, model-based fuzzing, risk- and threat-oriented testing, and the usage of security test patterns. This paper provides a survey on MBST techniques and the related models as well as samples of new methods and tools that are under development in the European ITEA2-project DIAMONDS.

1 Introduction

The times of rather static communication in strictly controlled, closed networks for limited purposes are over, while the adoption of the Internet and other communication technologies in almost all domestic, economic and social sectors with new approaches for rather dynamic and open networked environments overwhelmingly progresses. Social networks, networked communities, cloud computing, Web X.0, mashups or business process design are just some of the trends that reflect the tendency towards permanent connections and permanent data collection. Today's networked systems face the challenge of various security threats, which is usually met by various protection systems against attacks from the direct system users. In an interconnected world, software with vulnerabilities presents a threat not only to individuals but also to companies and public organizations, and last but not latest to national and international cooperation. Compared with functional hazards, which tend to be straightforward and accidental, security threats are often intentional and more persistent. Here are some facts that highlight the special nature of security threats:

- Attacks are frequently carried out by well organized groups with a commercial background (spamming, extortion, industrial espionage)
- Multi-stage attacks skilfully combine vulnerabilities on system level and organizational level

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 1–12, doi:10.4204/EPTCS.80.1

- information security risk analysis does often not hold for the complete life time of a product (context of product usage may change, new vulnerabilities are detected)

Practical security involves a sufficient understanding of risk in order to properly address it, manage it and, with care, to eliminate it. National and international standardization committees provide significant efforts on security evaluation and assessments. It includes classical concepts from security evaluation using common criteria (CCRA), but also European activities from ETSI addressing risk and threat analysis (TVRA).

The main aim of information security methods and techniques is the reduction or elimination of unwanted incidents that may harm the technical infrastructure or, even worth, its environment. The field of information security has been growing and reaches far further than simply using cryptography. The purpose of today's security standards is to specify countermeasures that can protect a system against certain forms of exploitation. Countermeasures depend on an understanding primarily on finding and assessing the vulnerabilities that exist within a system. Testing can be seen as an action to proactively detect such vulnerabilities. The Software Engineering Institute, USA, highlighted in 2009: "The security of a software-intensive system is directly related to the quality of its software." About 90 percent of all software security incidents are caused by attackers who exploit known vulnerabilities. Moreover most known vulnerabilities originate from software faults or design flaws. However, not every fault or design flaw constitutes vulnerability. Still, systematic testing increases the likelihood of identifying faults and vulnerabilities during the design, development or setup time of systems and enables purposeful fixes.

2 Model-based security testing and security testing

Software testing is an experimental approach of validating and verifying that a software system meets its functional and extra-functional requirements and works as expected. In this article, testing refers to active, dynamic testing, where the behavior of a system under test (SUT) is checked by applying intrusive tests that stimulate the system and observe and evaluate the system reactions. This is done by applying specification-based and/or code-based test cases that are – based on test hypotheses – directed to find faults in the SUT and by providing test suites, i.e. specifically selected collections of test cases, which provide an argument for the absence of faults.

Software security testing is a special kind of testing with the aim in validating and verifying that a software system meets its security requirements. Two principal approaches can be used: functional security testing and security vulnerability testing [5]. While security functional testing is used to check the functionality, efficiency and availability of the designed and developed security functionalities and/or security systems (e.g. firewalls, authentication and authorization subsystems, access control), security vulnerability (or penetration) testing directly addresses the identification and discovery of yet unknown system vulnerabilities that are introduced by security design flaws or by software defects. Security vulnerability testing uses the simulation of attacks and other kinds of penetration attempts.

The systematic identification and reduction of security-critical software vulnerabilities and of defects will increase the overall dependability of software-based systems and helps providing adequate security levels for open systems and environments. Unfortunately security testing, especially security vulnerability testing, lacks systematic approaches, which enable the efficient and goal-oriented identification, selection and execution of test cases. Risk-based testing [4] is a methodology that makes software risks the guiding factor to solve decision problems in the design, selection and prioritization of test cases.

2.1 Related work

The basic idea of model-based testing (MBT) is that instead of creating test cases manually, selected algorithms are generating them automatically from a (set of) model(s) of the system under test or of its environment. While test automation replaces manual test execution by automated test scripts, model-based testing replaces manual test designs by automated test generation. Although there are a number of research papers addressing model-based security (see e.g. [2, 9] and model-based testing (see e.g. [1]), there is until today little work on model-based security testing (MBST). Relevant publications in the field of MBST are [3, 10, 11, 16, 21, 23, 12, 19].

Kaksonen et al. [12] from the PROTOS project (1999-2001) discuss and implement an MBST approach using syntax testing as the starting point, and implement the models using Augmented Backus-Naur Form (ABNF). The PROTOS approach to model-based testing reads in context-free grammars (defined by BNF, ASN.1 (Abstract Syntax Notation One),or XML (Extended Markup Language)) for critical protocol interfaces and generates the tests by systematically walking through the protocol behaviour. These two approaches are the only ones commercially available [19].

Jurjens and Wimmel [11, 23] address the problem of generating test sequences from abstract system specifications in order to detect possible vulnerabilities in security-critical systems. Both papers assume that the system specification, from which tests are generated, is formally defined in the language Focus – a mathematical framework for the specification, refinement, and verification of distributed, reactive systems. The paper [11] focuses on testing of firewalls whereas [23] focuses on transaction systems. In [10], Jurjens extends [8] by considering system specification written in the language UMLsec – a security profile for the Unified Modelling Language. In [3], Blackburn et. al. summarizes the results of applying a model-based approach to automate functional security testing. The approach involves developing models of security requirements as the basis for automatic test vector and test driver generation. In particular, security requirements are written in the so-called SCRtool, are transformed into test specifications which in turn are transformed into test vectors and test drivers. The approach is targeted towards Java applications and database servers.

Mouelhi et al. [16] propose a model-driven approach to specifying, deploying and testing access-control policies in Java applications. The approach has four main steps. The first step is to build a platform-independent access-control model for the application. In the second step, the model is transformed into so-called platform specific policy decisions points (PDPs). In step three, the PDP is integrated into the functional code of the application by aspect oriented programming techniques. Finally, in step four, the resulting integrated application is tested against tests that are generated from the platform independent access-control model. Another approach covering specification, deployment, testing and monitoring of security policies has been proposed in the Politess project (Grenoble INP, IT, Smartesting) [DFGMR06, MBC08, LR07]. In [21], Wang et. al. presents a threat driven approach to MBST. In this approach, UML sequence diagrams to specify threat a model, i.e., event sequences that should not occur during the system execution. The threat model is then used as a basis for code instrumentation. Finally, the instrumented code is recompiled and executed using randomly generated test cases. If an execution trace matches a trace described by the threat model, security violations are reported and actions should be taken to mitigate the threat in the system.

2.2 Models in model-based security testing

In order to test security properties information from different sources are needed and need to be systematically related to each other to support tracing and proper usage of the information. In addition to

the functional system specification and to system architecture information, information on known or potential vulnerabilities, potential attacks and their occurrence probabilities can give guidance on what to test and how to test. In addition, probabilities and estimations on the severity of potential attacks can be summed up to form a risk analysis that points at the threats that are to be considered. Such a risk analysis provides guidance for test ordering and test prioritization and supports test management by indicating the need for test recommended test resources.

Hence, model-based security testing needs to be based on different types of models in order to cover the different perspectives used in securing a system. In the following we provide three different model categories that each represent a perspective on its own and may serve as input models for test generation.

2.2.1 Architectural and functional models

Architectural and functional models of the SUT are concerned with system requirements regarding the general behaviour and setup of a software-based system. The main perspective of these models are the structure and properties of the system under test. The models exist on different level of abstraction and in different granularity. Often they show additions that allow to focus on specific system properties like robustness properties (e.g. failure states) or performance properties (e.g. durations or throughputs). Regarding security testing, we are principally interested in locating critical system functionality with respect to the overall software architecture and in identifying security-critical interfaces, which might be an entry point for an adversary. Related to security-critical interfaces, interaction models or protocol models (involving data models or behavioural models) are of high interest for security testing. In addition functional security measures (such as authentication or access control means) can be specified within functional models and be tested by use of functional testing approaches.

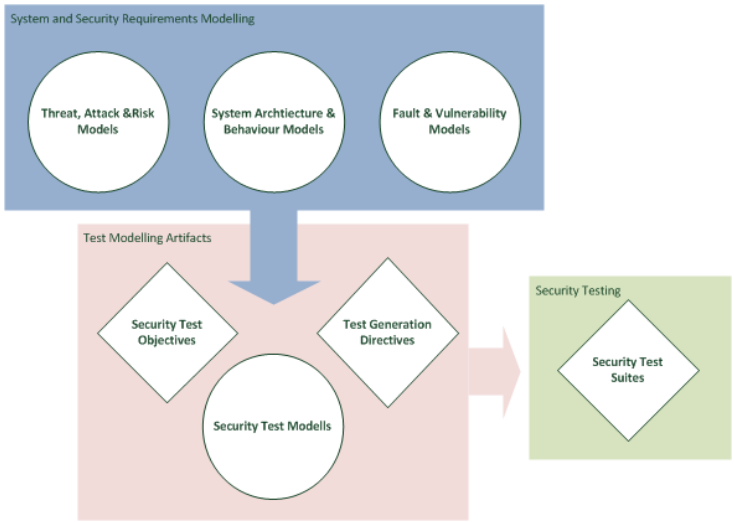


Figure 1: Modelling Artifacts in MBST

2.2.2 Threat, fault and risk models

While architectural and functional models typically describe the expected system configuration and behaviour, risk modelling techniques like the CORAS risk modelling approach [13, 7] focus on what can

go wrong. CORAS provides means to mode risks, threats or faults and enables the identification of multiple risk factors, describe their relationships and relate them to occurrence probabilities and potential impacts.

Besides CORAS, further approaches for fault and attack modelling exist. Well-known are the fault tree analysis (FTA [20]) and the cause-consequence analysis (CCA [17]). FTA considers high-level faults and decomposes them top-down to basic events, which can be identified and tested for in the system.

A variant of fault trees are so called attack trees [14]. Attack trees are directly related to security risks. They start from a high-level attack scenario and decompose them to concrete basic interactions with the system.

ETA (event tree analysis [18]) works bottom-up. It starts with the identification of unwanted system events and analyses the consequences in case of an occurring unwanted event.

A CCA (cause-consequence analysis [6]) – a combination of the FTA and ETA concepts – can be used as well. That analysis starts with a thread. The causes (top-town) and the consequences (bottom-up) are analyzed simultaneously.

2.2.3 Weakness and vulnerabilities models

While threat, fault and risk models concentrate on causes and consequences of system failures, weaknesses or vulnerabilities, a weakness or vulnerability model describes the weakness or vulnerability by itself. The information needed to develop such models are normally given by databases like the National Vulnerability Database (NVD) or the Common Vulnerabilities and Exposures (CVE) database. These databases collect known vulnerabilities and provide the information to developers, testers and security experts, so that they can systematically check their products for known vulnerabilities. One of the challenges yet not sufficiently solved is how these vulnerabilities can be integrated in system models, so that they can be used for test generation.

One possible solution is based on the idea of mutation testing [22]. Typically, mutation testing is used to qualify test suites by running tests against a mutation of the system under test. The quality of the test suite is stated with respect to the number of mutants being detected by the test suite. For security testing, models of the system under test are mutated in a way that the mutants represent weaknesses or known vulnerabilities. These weakness or vulnerability models can then be used for test generation by various MBT approaches. The generated tests are used to check whether the system under test is weak or vulnerable with respect to the weaknesses and vulnerabilities in the model.

2.3 Activities in model-based security testing

Security testing like any other testing follows a series of activities and uses artifacts that aim to systematically plan, design, specify, realize, and execute tests, and to evaluate the test results and, if needed, to readjust the planning etc.

MBST shows slight differences in these activities, but follows the main sequence of activities. A main task is to provide system properties under consideration concrete test cases (data and behaviour) that represent the stimuli to the system under test and the evaluation of the system reactions by test oracles. The following discusses how to generate security tests with a model-based approach.

- Identify security test objectives and methods: The test objectives define the overall goals of testing and relate the goals to testing methods that allow to accomplish the objectives. For model-based testing the modelling techniques and test generation strategies need to be planned. Especially

threat, fault, and risk models are to be considered to guide or strengthen the test identification with respect to the identified risks, threats, faults, and their consequences.

- Design a functional test model: The test model reflects either the expected functional scenarios of the SUT (system perspective) or the scenarios of the SUT usage (system perspective). Standard modelling languages such as UML can be used to formalize the points of control and observation of the SUT, the dynamic behaviour when interacting with the system, the entities associated with the test in various test configurations, and the test data applied to the system. The test models need to be precise and complete enough to allow automated derivation of executable tests from these models. However, security testing focuses either on testing the correctness of security functions or on testing the robustness against a dedicated misuse of the system. Thus, *functional test models* used for security testing describe not only the typical environment or usage of a system, but also adversary environments or atypical usages like attacks and hacking attempts.
- Determine test generation criteria: Usually, there is an infinite number of possible tests that can be generated from a model, so that test designers choose test generation or selection criteria to limit the number of (generated) tests to a finite number by e.g. selecting highest-priority tests, or to ensure specific coverage of system structures, behaviours, or alike. For security testing, approaches based on structural model coverage, i.e. determining the coverage of model elements by generated tests, is not sufficient. Hence, fuzz test approaches are used that follow other kind of coverage criteria and lead to different, but also larger number of generated test data or test behaviour. However, approved test generation criteria like the coverage of security functional requirements or the weighting of security functional requirements with risk values are applicable as well.
- Generate the tests: The test generation is in MBT typically a fully automated process to derive the test cases from a given test model as determined by the chosen test generation criteria. This is also true for MBST. The generated test cases are sequences of high-level events or actions to or from the SUT, with input parameters and expected output parameters and return values for each event or action. If needed, the generated tests are further refined to a more concrete level or adapted to the SUT to support their automated execution.
- Assess the test results: During test result evaluation and test assessment, the quality of the SUT can be rated with respect to the test results as well as the quality of tests can be rated with respect to their fault and vulnerability revealing capabilities. While for functional MBT, measurements and metrics for system and test quality exist this is still a research challenge for MBST. Furthermore, the test results need to be analyzed if changes to the system requirements, the system design, the risk analysis or to the test process itself are needed. In these cases, required iterations are to be started.

2.4 Testing Approaches in DIAMONDS

The project DIAMONDS (Development and Industrial Application of Multi-Domain Security Testing Technologies) develops under the direction of Fraunhofer FOKUS, Berlin, efficient and automated security test methods for security-critical, networked systems in various industrial domains such as industrial automation, banking and telecommunications. DIAMONDS develops methods to design objective, transparent, repeatable, and automated security tests that focus on system specifications and related risks. The project goals include the definition of security fault and vulnerability modelling techniques, the definition of a security test pattern catalogue, the development of MBST techniques, and the definition of

a MBST methodology. DIAMONDS examines vulnerabilities of networked systems in the considered domains in order to derive common principles, methods and means that enable effective security testing of industrial importance. In reflection of the case studies results, the DIAMONDS security testing methodology will be evaluated and optimized. The project results are made available to interested parties and also through contributions to the standardization at ETSI and to other standardization bodies.

A special focus is given in DIAMONDS to (1) risk-based MBST and to (2) model-based fuzz testing.

2.4.1 Risk-based security testing

Risk-based testing can be generally introduced with two different goals in mind. On the one hand side risk-based testing approaches can help to optimize the overall test process. The results of the risk analysis, i.e. the results of threat and vulnerability analysis, are used to guide the test identification and may complement requirements engineering results with systematic information concerning threats and vulnerabilities of a system. On the other hand side, attack simulation is to find deviations of the SUT to its specification that leads to vulnerabilities because invalid inputs are not rejected but processed by the SUT instead. Such deviations may lead to undefined states of the SUT and can be exploited by an attacker, for example to successfully perform a denial-of-service.

A comprehensive risk assessment additionally introduces the notion of risk values, that is the estimation of probabilities and consequences for certain threat scenarios. These risk values can be additionally used to weight threat scenarios and thus help identifying which threat scenarios are more relevant and thus identifying the threat scenarios that are the ones that need to be treated and tested more carefully.

Furthermore, risk-based testing approaches can help to optimize the risk analysis and the risk assessment itself. Risk analysis and risk assessment, similar to other development activities in early project phases, are mainly based on assumptions on the system itself. Testing is one of the most relevant means to do real experiments with a system and thus enables to gain empirical evidence on the existence of vulnerabilities, the applicability and consequences of threat scenarios and the quality of countermeasures. Thus, risk-based testing results can be used as a form of evidence for the assumptions that have been made during the risk evaluation and risk assessment.

In particular, risk-based testing can help in

- providing evidence on the functional correctness of countermeasures,
- providing evidence on the absence of known vulnerabilities, and
- discovering unknown vulnerabilities,
- optimizing risk analysis by identifying new risk factors and reassessing the risk values.

The CORAS language [7] integrates different tree-based approaches for risk modelling. It is a graph-based modelling approach that emphasizes the modelling of threat scenarios and provides formalisms to annotate the threat scenarios with probability values and formalisms to reason with these annotations.

Figure 2 shows a simple CORAS risk model that depicts a threat scenario for an unauthorized database access. The CORAS language allows to relate threat scenarios to adversaries (so called threats, e.g. "'hacker"' or "'script kiddy"' in Figure 2) and to potential vulnerabilities. A vulnerability is denoted by the unlocked padlock (see e.g. "'SQL injection"'). Last but not least the threat scenario is related to unwanted incidents, e.g. "'Database entries are modified by unauthorized people"'. This modification harms the asset "'database contents"' (see "'brown moneybag"' in Figure 2) and can therefore negatively influence the asset "'revenue"' (see "'white moneybag"'). The treatment "'use of prepared statements"' instead of strings containing SQL queries is depicted with the green pliers in that figure. Except the

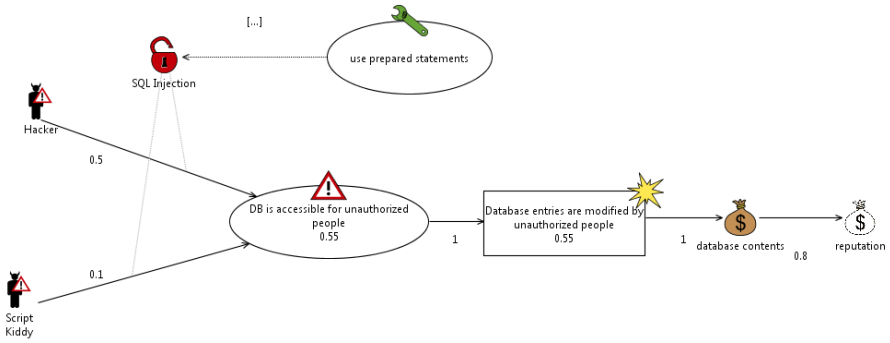


Figure 2: CORAS treatment diagram

threats, the vulnerability, the assets and the treatment, all elements have annotations that denote the probability or frequency of the transition or the incidence of for instance a threat scenario.

Such risk models can be used in different ways to support testing. The goal of *risk-based test identification* is to improve the test design such that high-risk areas of the SUT are covered and that at the same time test resources are optimally used by focusing on highest risks first.

In risk-based test identification, for individual risk factors test objectives are developed. We consider a test objective to be foremost an informal specification that defines which aspect of a certain system, functionality, or protocol etc. should be tested. Similar to requirements in requirements engineering, test objectives constitute test requirements that can be refined and decomposed during the test development. In the following we describe the relationship between test objectives and the elements used in the risk analysis. Quantifications of the related risks can be used to weight the test objectives.

Test objectives for

- an unwanted incident describe which test methods can be applied to initiate and detect an unwanted incident and to characterize its consequences.
- threat scenarios describe which test methods can be applied to initiate a threat scenario and to characterize its consequences.
- vulnerabilities describe which test methods can be applied to elicit a vulnerability.
- treatment scenarios describe which test methods can be applied to characterize the maturity and effectiveness of a treatment scenario.

Another way of using risk models in testing is *risk-based test selection*. It is used to find an optimal set of test cases along certain selection strategy. The selection strategy takes into account available test resources and optimizes the selected tests with respect to the chosen coverage criteria. In functional testing coverage is often described by the coverage of requirements or by the coverage of system or test model elements such as states, transitions, or decisions. In risk-based testing, we aim at the coverage of system risks. The criteria are designed by taking the risk values from the risk assessment to set priorities for the test generation or to order the test execution in a test run. The test selection can be either accomplished on existing test cases to select tests for test run or during test generation to enable the directed, goal-oriented generation of tests.

Last but not least, security testing supports *risk control*. Risk control deals with the revision of risk assessment results by correcting assumptions on probabilities, consequences or the maturity of treatments scenarios or deals with the completion of risk analysis result by integrating vulnerabilities and thus

potentially threats, threat scenarios and unwanted incidents. The test results can reflect and verify the assumptions that have been made during risk analysis. The test results can be used to adjust risk analysis results by introducing new or revised vulnerabilities or revised risk estimations on basis of the defects being found. Test results, test coverage information and a revised or affirmed risk assessment can provide solid arguments for the security level of a system as test results relate to the

- risks, which are addressed and covered by related test cases.
- treatment or threat scenarios, which are to be checked by the test cases.
- assets, whose related risks are checked by corresponding test cases.
- vulnerabilities, which have been identified and located related test cases.

2.4.2 Model-Based Fuzzing

While the origin of fuzzing is based on a complete randomized approach [15], block-based and model-based fuzzers use their knowledge about the message structure to systematically generate messages containing invalid data among valid data [19].

Systematic approaches are often more successful because the message structure is preserved and thus the likelihood increases that the generated message is accepted by the SUT. Using fuzz testing principles not only for test data generation but also for test behaviour generation complements the traditional fuzz testing approaches. Behaviour fuzzing does not only reflect the generation of atypical messages but also changes the typical appearance and order of messages. For example a valid and approved sequence of messages can be turned into an atypical and unknown sequence by rearranging messages, repeating and dropping them or just by changing the type of message.

Behaviour fuzzing aims at finding flaws in design and vulnerabilities in systems that are not simply revealed by applying invalid input data. It focuses on misuse on a higher level of functionality. For example, a security requirement defines that a download may only be started after successful authentication. In a vulnerable system the download can be started additionally without any authentication. Such fault can be detected using typical input data but atypical behaviour, e.g. by simply omitting the authentication.

DIAMONDS develops model-based fuzzing approaches that use e.g. fuzzing operators on scenario models which are specified by sequence diagrams. In the following, a simplified example from the banking domain is used to illustrate how fuzzing operators are applied to sequence diagrams. For ease of understanding most parameters are omitted.

The sequence diagram in Figure 3 describes how a bank customer can perform a transfer order. The customers can either order a national or an international transfer (message 1). Afterwards the customer sends the name of the recipient of the transfer order, the amount to be transferred (message 2) as well as recipient's national bank account information (message 3) in case of a national transfer order or recipient's international bank account information (message 4) in case of an international transfer order. The transfer order must be authorized by the customer sending a valid transaction number TAN (message 5). If the customer accidentally sent an invalid TAN e. g. by mistyping it, he can try to enter a valid TAN up to two times again (message 7, combined fragment loop).

Applying fuzzing operators to the diagram, messages can be moved, removed, repeated, inserted or the type of a message can be changed to obtain an invalid sequence. Fuzzing operators perform a mutation on the diagram resulting in an invalid sequence in comparison with the original. One fuzzing operators performs only one mutation of a sequence diagram. For instance, a fuzzing operator can move

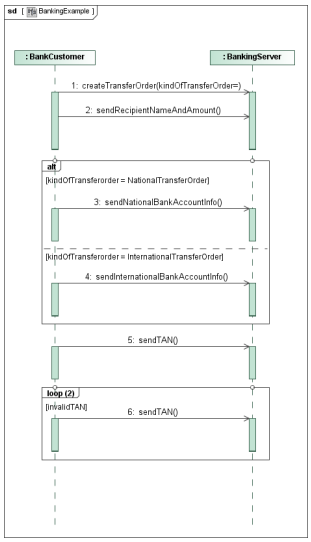


Figure 3: Transfer Order Sequence

message 5 after message 2. Another fuzzing operator can generate an invalid sequence of messages by negating interaction constraints of interaction operands. By negating the interaction constraint of the loop combined fragment, the sequences generated from the resulting sequence diagram contain at least two valid transaction numbers sent to the banking server (three if the second given TAN is valid). Figure 4 shows the results of the fuzzing operators from above.

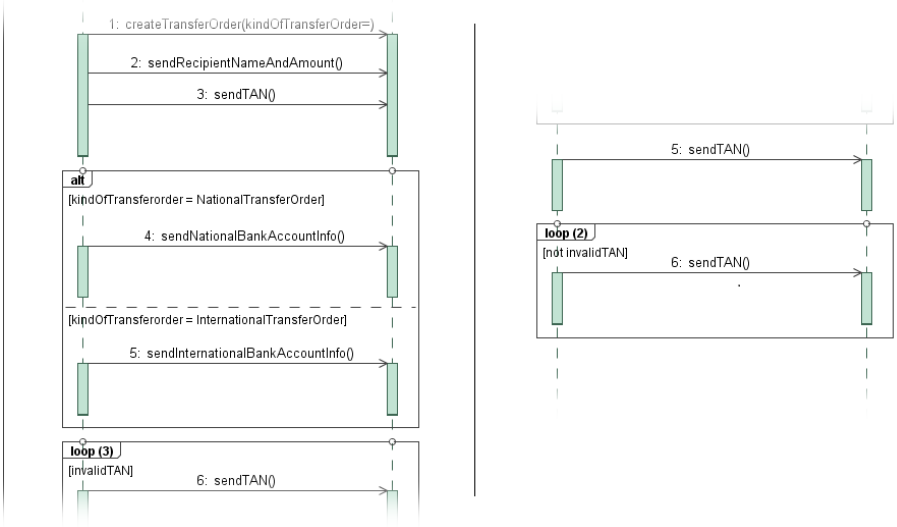


Figure 4: Fuzzed Transfer Order Sequence

Performing the above mentioned fuzzing operators leads to different sequence diagrams that are the basis for further test case generation. However, the main idea of fuzz testing approaches in general is the ability to automatically generate a large number of test cases. This is achieved by applying not only one fuzzing operator to a sequence diagram, but a set of fuzzing operators multiple times, e.g.

by applying a single fuzzing operator to several model elements of a sequence diagram or by applying several, possibly different fuzzing operators, one after another. The combination of fuzzing operators permits the generation of a large number of test cases.

3 Summary

Model-based security testing (MBST) is a relatively new field and especially dedicated to the systematic and efficient specification and documentation of security test objectives, security test cases and test suites, as well as to their automated or semi-automated generation. This paper provides an initial survey on model-based security testing by analyzing related work, discussing models that can be used for model-based security testing, and by outlining two main approaches that are being developed in the European ITEA2 project DIAMONDS by industrial and research partners from 6 countries:

- Risk-based security testing
- Model-based fuzzing

Details of risk-based security testing and model-based fuzzing are given in the DIAMONDS deliverables. While DIAMONDS is an ongoing project that is at the half of the project duration having reached 2 of 4 milestone, the methods are still under development and the analysis of the gains and the pros and cons of the methods is still to be done. However, initial versions of the methods have already been applied in selected case studies that demonstrated the potentials of the described approaches.

References

[1] Paul Baker, Zhen Ru Dai, Jens Grabowski, ystein Haugen, Ina Schieferdecker & Clay Williams (2007): *Model-Driven Testing: Using the UML Testing Profile*, 1 edition. Springer, Berlin. Available at `http://dx.doi.org/10.1007/978-3-540-72563-3`.

[2] David Basin, Jürgen Doser & Torsten Lodderstedt (2006): *Model driven security: From UML models to access control infrastructures*. *ACM Trans. Softw. Eng. Methodol.* 15, pp. 39–91. Available at `http://doi.acm.org/10.1145/1125808.1125810`.

[3] Mark Blackburn, Robert Busser & Aaron Nauman (2002): *Model-based approach to security test automation*. In: *International Software Quality Week*.

[4] Paul Gerrard & Neil Thompson (2002): *Risk Based E-Business Testing*. Artech House, Inc., Norwood, MA, USA.

[5] F. Y. Gu Tian-yang, Shi Yin-sheng & Yuan (2010): *Research on Software Security*. *Testing World Academy of Science* Engineering and Technology 69 2010.

[6] Matthias Güdemann, Frank Ortmeier & Wolfgang Reif (2007): *Using Deductive Cause Consequence Analysis (DCCA) with SCADE*. In: *Proceedings of SAFECOMP 2007*, Springer LNCS 4680.

[7] Ida Hogganvik (2007): *A Graphical Approach to Security Risk Analysis*. Ph.D. thesis, Oslo : University of Oslo, Department of Informatics.

[8] Jan Jürjens (2002): *UMLsec: Extending UML for Secure Systems Development*. In Jean-Marc Jézéquel, Heinrich Hussmann & Stephen Cook, editors: *The Unified Modeling Language*, *Lecture Notes in Computer Science* 2460, Springer Berlin / Heidelberg, pp. 1–9. Available at `http://dx.doi.org/10.1007/3-540-45800-X_32`.

[9] Jan Jürjens (2005): *Secure Systems Development with UML*. Springer. Available at `http://dx.doi.org/10.1007/b137706`.

[10] Jan Jürjens (2008): *Model-based Security Testing Using UMLsec*. *Electron. Notes Theor. Comput. Sci.* 220, pp. 93–104. Available at http://dl.acm.org/citation.cfm?id=1467086.1467133.

[11] Jan Jürjens & Guido Wimmel (2001): *Specification-Based Testing of Firewalls*. In Dines Bjørner, Manfred Broy & Alexandre V. Zamulin, editors: *Ershov Memorial Conference*, *Lecture Notes in Computer Science* 2244, Springer, pp. 308–316. Available at http://dx.doi.org/10.1007/3-540-45575-2_31.

[12] Rauli Kaksonen (2001): *A functional method for assessing protocol implementation security*. VTT Publications 448, VTT Technical Research Center of Finland.

[13] M. S. Lund, B. Solhaug & K. Stlen (2011): *Model-Driven Risk Analysis. The CORAS Approach*. ISBN: 978-3-642-12322-1, Springer.

[14] Sjouke Mauw & Martijn Oostdijk (2005): *Foundations of Attack Trees*. In: *International Conference on Information Security and Cryptology ICISC 2005. LNCS 3935*, Springer, pp. 186–198.

[15] Barton P. Miller, Lars Fredriksen & Bryan So (1990): *An Empirical Study of the Reliability of UNIX Utilities*. In: *In Proceedings of the Workshop of Parallel and Distributed Debugging*, Academic Medicine, pp. pages ix–xxi,.

[16] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry & Yves Le Traon (2008): *A Model-Based Framework for Security Policy Specification, Deployment and Testing*. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl & Markus Völter, editors: *MoDELS*, *Lecture Notes in Computer Science* 5301, Springer, pp. 537–552. Available at http://dx.doi.org/10.1007/978-3-540-87875-9_38.

[17] D.S. Nielsen (1971): *The Cause/Consequence Diagram Method as a Basis for Quantitative Accident Analysis*. Technical Report RISO-M-1374, Danish Atomic Energy Commission.

[18] K.A. Reay & University of Loughborough (2002): *Efficient fault tree analysis using binary decision diagrams/*. University of Loughborough. Available at http://books.google.de/books?id=_0SFGwAACAAJ.

[19] A. Takanen, J. DeMott & C. Miller (2008): *Fuzzing for software security testing and quality assurance*. Artech House information security and privacy series, Artech House. Available at http://books.google.de/books?id=tMuAc_y9dFYC.

[20] W E Vesely, F F Goldberg, N H Roberts & D F Haasl (1981): *Fault Tree Handbook*. *Office* (NUREG-0492), p. 209. Available at http://www.stormingmedia.us/37/3794/A379453.pdf.

[21] Linzhang Wang, Eric Wong & Dianxiang Xu (2007): *A Threat Model Driven Approach for Security Testing*. In: *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, IEEE Computer Society, Washington, DC, USA, pp. 10–. Available at http://dx.doi.org/10.1109/SESS.2007.2.

[22] Martin Weiglhofer, Bernhard K. Aichernig & Franz Wotawa (2009): *Fault-Based Conformance Testing in Practice*. *Int. J. Software and Informatics* 3(2-3), pp. 375–411. Available at http://www.ijsi.org/IJSI/ch/reader/view_abstract.aspx?file_no=375{&}flag=1.

[23] Guido Wimmel & Jan Jürjens (2002): *Specification-Based Test Generation for Security-Critical Systems Using Mutations*. In: *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '02, Springer-Verlag, London, UK, UK, pp. 471–482. Available at http://dl.acm.org/citation.cfm?id=646272.685812.

Reusing Test-Cases on Different Levels of Abstraction in a Model Based Development Tool

Jan Olaf Blech Dongyue Mou Daniel Ratiu
{blech,mou,ratiu}@fortiss.org
fortiss GmbH

Seamless model based development aims to use models during all phases of the development process of a system. During the development process in a component-based approach, components of a system are described at qualitatively differing abstraction levels: during requirements engineering component models are rather abstract high-level and underspecified, while during implementation the component models are rather concrete and fully specified in order to enable code generation. An important issue that arises is assuring that the concrete models correspond to abstract models. In this paper, we propose a method to assure that concrete models for system components refine more abstract models for the same components. In particular we advocate a framework for reusing test-cases at different abstraction levels. Our approach, even if it cannot completely prove the refinement, can be used to ensure confidence in the development process. In particular we are targeting the refinement of requirements which are represented as very abstract models. Besides a formal model of our approach, we discuss our experiences with the development of an Adaptive Cruise Control (ACC) system in a model driven development process. This uses extensions which we implemented for our model-based development tool and which are briefly presented in this paper.

1 Introduction

Formally defining relations between different models used in the development process of an entire system or a component is an important prerequisite to apply techniques that give confidence in the refinement process. Lifting properties – in our case single test-cases – to work on more refined models is one technique towards gaining this confidence. It is in general much easier and scalable to perform than, e.g., using formal verification techniques, and thus more convenient to use for a wider range of applications and developers that are not trained in the usage of formal methods.

Abstract models during an early phase in the software development process can be used to capture single requirements directly. Such models of single requirements concentrate on distinct aspects of a system and thereby are rather small. These models may represent functional requirements and are relatively easy to validate using test-cases and a simulation environment. At a later stage in the development process we can have a model that is more refined and may describe the entire system functionality with the interplay of several fine-grained requirements. A challenge is to use the existing test-cases to get confidence that the concrete implementation models refine the abstract requirement models.

In this work, we regard the relation between abstract models that may correspond to single requirements, and more concrete models that can already contain enough information to automatically generate an implementation. We regard the transformation of existing test-cases for the abstract models into test-cases for the concrete ones.

The initial motivation for this work – that we faced in a project – is a situation where manufacturers of larger systems (e.g., a car) give an abstract specification with abstract test-cases to sub-contractors that are responsible for developing a distinct component (e.g., a control system used in the car). Here, at

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 13–27, doi:10.4204/EPTCS.80.2

least the provided test-cases for the abstract specification must be respected by the implementation that the sub-contractor provides.

We present a formal framework for relating test-cases, report on our implementation experiences and regard a case-study on Adaptive Cruise Control (ACC) systems. Requirements ACC on systems are standardized in the ISO 15622 standard [8]. In this paper, we present some requirements and a model that captures the implementation and evaluate our test-case transformation using this case-study. The featured implementation and case study is integrated into our AutoFocus 3 [2] model-based development environment and its modeling language.

Our paper features the following contributions:

- A formal framework for relating models of components for different abstraction levels which aims especially at concretizing test-cases for AutoFocus 3.
- An implementation of the framework in AutoFocus 3.
- We exemplify our approach using a simplified variant of an ACC system.

Thus, we are addressing the issue of transforming existing test-cases and relating abstract and concrete models with each other. Derivation of new test-cases and evaluation of test-case quality is not subject to this paper.

1.1 Related Work

Relations between properties on abstract and concrete system representations have been studied comprehensively for temporal logics formula, e.g., in [9]. Our test-cases (test + expected result) may be regarded as properties if classified in the terms of this paper.

Abstractions in the context of model-based testing has been studied in [11]. Here, in contrast to our work, different abstraction levels of models are studied regarding suitability for deriving test-cases for a final implementation.

An early formal framework for relating test-cases, interactive systems and abstractions is described in [1]. Here, test-cases are regarded as contracts, which is similar to our view.

The usage of program analysis techniques on abstract system representation has been studied in the context of abstract testing [6], a kind of method for handling testing in an abstract interpretation framework.

The work presented in [13] presents a combination of abstract interpretation and model checking for testing. Here, somehow opposite to our work, concrete test-cases are abstracted for abstract system representations.

In addition to test-case abstraction and concretization, a large body of research has been done on other aspects of model-based testing. For a comprehensive overview and classification we refer to [12].

1.2 Overview

In Section 2 we describe our model-based development tool AutoFocus 3 and the semantics of the used language. The framework for relating models and transforming test-cases is presented in Section 3. Our case study and a small evaluation is presented in Section 4. Section 5 features a conclusion.

2 Our Model Based Development Environment

In this section we present the model based development tool: AutoFocus 3 that we use in the context of this work. In particular we present the semantics of the modeling language.

2.1 AutoFocus Semantics

Here we present a formal definition of our modeling language AutoFocus a dialect of FOCUS [5]. We follow the description given in [7]

In AutoFocus a system and its components are described by a *stream processing function*, which defines its syntactic interface as well as its behavior. Furthermore, the AutoFocus approach offers composition operators which allow to derive a larger system (the composed system) out of modularly defined functions.

2.1.1 Streams

Basically, the AutoFocus theory is based on the idea of timed data streams which are used to model the asynchronous interaction between a function and its environment. Streams represent histories of communication of data messages in a given time frame. Intuitively, a timed (data) stream can be thought of as a chronologically ordered sequence of data messages.

Definition 1 (Timed Stream)**.** *Given a set M of data messages, we denote a timed stream of elements from M by a mapping*

$$s : \mathbb{N} \to M.$$

For each time $t \in \mathbb{N}$, $s(t) = s.t$ denotes the message communicated at time t in a stream s and $s \downarrow t$ the prefix of the first t messages in the timed stream s, i.e., the messages communicated until (and including) time t.[1]

We have chosen so-called timed data streams that allow us to flexibly include the timing issues of functions whenever required. We base our approach on a simple notion of discrete time: we assume a model of time consisting of an infinite sequence of time intervals of equal length. Thus, time can be simply represented by the natural numbers $\mathbb{N}_+$. In each time interval a message $m \in M$ can be transmitted.

An exemplary timed stream s over the data set $\mathbb{B} = \{0,1\}$ is defined by the function $\forall t \in \mathbb{N} : s(t) = 1$. This means that in each time interval the stream contains the value 1, i.e., $s = (1\ 1\ 1\ \ldots)$.

2.1.2 Input/Output Channels and Channel Histories

Every stream processing function is connected to its environment by channels. The channels of a stream processing function are divided into disjoint sets of input channels $I = \{i_1, \ldots i_n\}$ and output channels $O = \{o_1, \ldots o_n\}$. Channels are used as identifiers for streams.

With every channel c, we associate a data type $Type(c)$ indicating the set of messages sent along this channel. To that end, we define the channel type by the following function:

$$Type : C \to \texttt{Type},$$

[1] The theory is originally defined for stream processing functions $s : \mathbb{N}_+ \to M^*$, which assign a sequence of messages to each time interval. In order to keep the paper at hand as understandable as possible, we decided for the simplification that only one message can be communicated within each time interval. A proper description of the original theory can be found in [5, 4]

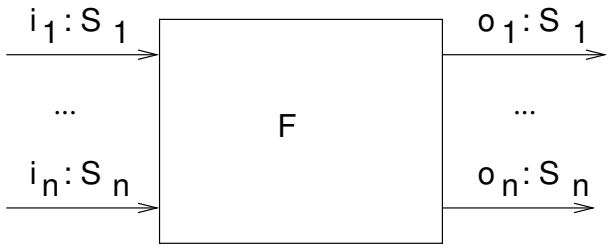


Figure 1: Graphical representation of a FOCUS function and its typed I/O channels

which maps each channel $c \in C$ to a data type $t \in \texttt{Type}$ from the set of all possible data types Type.

To describe the function's communication with its environment, each channel is associated with a stream which represents the messages communicated over this channel (cf. Figure 1). A mapping that associates a stream to any channel from a set of channel C is called *(channel) history* of C.

Definition 2 (Channel History)**.** *Let C be a set of typed channels. A channel history h is a mapping*

$$h : C \to (\mathbb{N} \to M),$$

such that $h(c)$ is a stream of the type Type(c) for each $c \in C$.

The set of all histories over the set of channels C is denoted by $\mathbb{H}(C)$.

2.1.3 Specification of Stream Processing Functions

The black-box specification of a stream processing function consists of a syntactic interface and its semantics.

A stream processing function is connected to its environment exclusively by its syntactic interface consisting of input/output channels. This interface indicates which types of messages can be exchanged.

Definition 3 (Syntactic Interface)**.** *The syntactic interface of a function is denoted by*

$$(I \blacktriangleright O),$$

where I and O denote the sets of typed input and output channels, respectively.

For a function with syntactic interface $(I \blacktriangleright O)$, the set of all syntactically correct history pairs is denoted by

$$\mathbb{H}(I) \times \mathbb{H}(O).$$

However, the syntactic interface tells nothing about the interface behavior of the function.

The behavior (semantics) of the stream processing function is given by the mapping of histories of the input channels to histories of the output channels. Thereby, we distinguish between total and partial functions. While the behavior of a total function is defined for *all* syntactically correct inputs, the behavior of a partial functions is defined for a *subset* of the inputs.

Definition 4 (Semantics Relation)**.** *The semantics of a total stream processing function with syntactic interface $(I \blacktriangleright O)$ is given by a relation*

$$F : \mathbb{H}(I) \to \mathscr{P}(\mathbb{H}(O))$$

that fulfills the following timing property for all its input histories.

Let be $x_1, x_2 \in \mathbb{H}(I), y_1, y_2 \in \mathbb{H}(O)$, and $t \in \mathbb{N}_+$. The timing property is specified as follows:

$$x_1 \downarrow t = x_2 \downarrow t \Rightarrow \{y_1 \downarrow (t+1) : y_1 \in F(x_1)\} = \{y_2 \downarrow (t+1) : y_2 \in F(x_2)\}.$$

By mapping into the power-set of $\mathbb{H}(O)$, Definition 4 allows to specify *nondeterministic* behavior. For an input history, there is a set of output histories that represent all possible reactions of the function to the input history. If a function defines exactly one output history for every input history, the function is called deterministic; if the set of output histories has several members for some input history, then the function is called nondeterministic.

The timing property expresses that the set of possible output histories for F for the first $t+1$ intervals only depends on the inputs of the first t time intervals. In other words, the processing of messages within a function takes at least one time interval. Functions that fulfill this timing property are called time-guarded or *strictly causal*. Strict causality is a crucial prerequisite for the composability of functions.

If we replace the expression $(t+1)$ by t in Definition 4 above (i.e., the outputs in the first t intervals depend on the inputs in the first t intervals), messages are processed by the function without time delay. Such functions are called *weakly causal*.

Stream processing functions are used to represent components in our terminology. They can contain other stream processing functions thereby forming composed components.

Moreover, it is important to notice that stream processing functions in the AutoFocus realization of FOCUS are defined as finite automata comprising internal states and not in a functional way working on entire streams. The definition is rather element wise, thus, consuming one element for each input channel at a time and generating an output element for each channel.

For further information, we refer to [5, 4] for a discussion of partial stream processing functions, weak and strong causality, and a deeper discussion of the semantics of FOCUS.

2.2 Tool Integration and Proposed Usages

The AutoFocus modeling language has been implemented in a model based development tool: AutoFocus 3 [2]. AutoFocus 3 supports structuring textual requirements and expressing them through models. It allows the graphical modeling of system components, their functionality and deployment aspects. It comprises code generation and integrates formal verification and validation tools. AutoFocus 3 is based on Eclipse RCP framework and comes with a plug-in mechanism for new functionality.

AutoFocus 3 aims at the development of embedded systems, e.g., in the automotive industry. It can be used for the entire development process. In a typical workflow with AutoFocus 3, project partners specify requirements in a textual way using AutoFocus, create corresponding models and establish test-cases based on these requirements. In the course of the project, partners provide more detailed models and an implementation. It is important, that the test-cases for the models representing higher abstraction levels of a system component can be transformed into test-cases for the more refined models.

3 An Abstraction Framework for Models and Test-cases

In this work, we want to relate abstract and concrete models and lift test-cases from more abstract models to more concrete ones. Characteristic for our framework are the facts that both abstract and concrete models are given in the same modeling language. In AutoFocus, a model does typically represent a (potentially composed) component. In AutoFocus a component's only means of interaction with its environment is via its channels. Thus, we do not have to deal with the possibility of additional side effects which facilitates a formal relation.

An important issue is the correctness of the lifting and if the refinement between models has been done correctly. Here we present basic definitions for test-cases and two formalisms for relating abstract

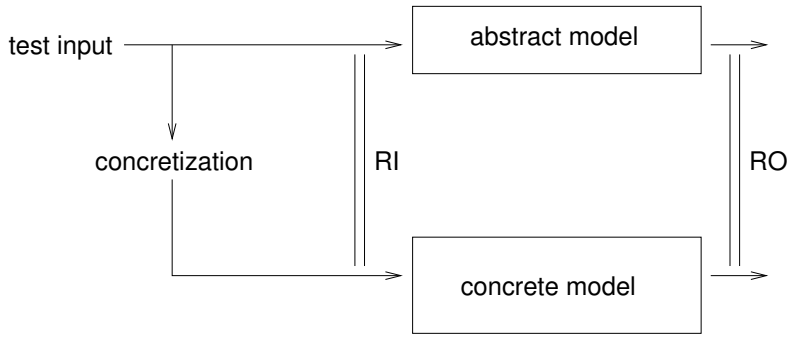


Figure 2: Our testing scenario

and concrete models which we have applied and evaluated in our model-based development environment:

1. A formalism for relating inputs of test-cases of their expected results using mathematical relations.
2. A formalism for relating the stream domains of abstract and concrete components using Galois connections.

3.1 Formal Definition of Test-cases

Here we present some basic definitions for test-cases. Test cases comprise test-input and expected results.

Definition 5 (Test-input)**.** *A test-input for an AutoFocus Component with syntactic interface* $(I \blacktriangleright O)$ *is a function* $c \mapsto s$ *with* $c \in I$ *and* s *a correctly typed stream (cf. Section 2.1).*

Definition 6 (Test expected-results)**.** *An expected-result of a test-case for an AutoFocus component with syntactic interface* $(I \blacktriangleright O)$ *is a set of tuples* (c,s) *with* $c \in O$ *and* s *a correctly typed stream.*

For the deterministic systems that we regard in this paper expected results can also be written as functions from output channels to streams.

Definition 7 (Test-case)**.** *A test-case for an AutoFocus component is a tuple of test-input and expected-result .*

3.2 Relating Test-inputs and Expected-results

Figure 2 shows our first formalism for relating models and test-cases: As a prerequisite, we establish two abstraction relations RI and RO which capture the relation between abstract and concrete component and compare the inputs of the two components and their results with each other. These can be stated in a formal way. Then we create an AutoFocus component which transforms abstract test-cases into concrete ones.

When such an existing test input for an abstract model is transformed into a concrete one we compare these test inputs and the expected-results using the relations thereby validating the correctness of test-case transformation.

Formally, we regard an abstract component with semantics function C_a and syntactic interface $(I_a \blacktriangleright O_a)$ featuring typed input channels I_a and output channels O_a. The concrete model has a semantics functions C_c and syntactic interface $(I_c \blacktriangleright O_c)$ featuring input channels I_c, and output channels O_c. The relation RI has the type $I_a \times I_c \to bool$, $\mathsf{RO} : O_a \times O_c \to bool$. These two relations have to be created for every pair of abstract and concrete models.

The formal definition of correctness of a test-case transformation is stated in the following definition.

Definition 8 (Corresponding test-inputs)**.** *Two test-inputs t_a, t_c are corresponding with respect to input and output correctness relations* RI, RO *for abstract and concrete AutoFocus components with semantic functions C_a, C_c typed accordingly iff*

$$\mathsf{RI}(t_a,t_c) \rightarrow \mathsf{RO}(C_a(t_a),C_c(t_c))$$

Here, we assume that it is relatively easy to ensure that these the relations have been stated correctly. Checking that the relations have been done correctly can be done either manually or by implementing an AutoFocus component which performs the check and emits a boolean stream of check values.

3.3 Relating the Domains of Components Using Galois Connections

A second way to relate abstract and concrete components with each other is to formalize a relationship between their domains.

Both abstract component (syntactic interface $(I_a \blacktriangleright O_a)$) and concrete component (syntactic interface $(I_c \blacktriangleright O_c)$) operate on domains of streams. When regarding the abstract input and output domains together as a single abstract domain and the concrete input and output domains as a single concrete domain and in case the abstract component is refined by the concrete one it is reasonable to relate abstract and concrete domain with each other via a Galois connection. Galois connection are used to represent refinements between different domains and capture their relation in a formal way (cf., e.g., [6]).

This implies the existence of two functions $f : I_c \cup O_c \rightarrow I_a \cup O_a$ and $g : I_a \cup O_a \rightarrow I_c \cup O_c$. Intuitively f lifts sets of concrete input and output streams to abstract ones and g does it the other way round. Furthermore, in order to illustrate the refinement between the two domains the functions have to fulfill the following conditions:

$$\forall\, T_c^{io}\ T_a^{io}\ .\ f(T_c^{io}) \subseteq T_a^{io} \text{ iff } T_a^{io} \subseteq g(T_c^{io})$$

The functions f and g capture the nature of the refinement. We formalize them in order to gain a precise description of the concretization allowing a transformation of test-cases.

In our proposed scenario we regard the transformation of abstract test-cases to concrete ones. The process of concretizing a test-case in general allows for multiple concrete test-cases for a given abstract one. Once we have established the function f, we formalize a parameterized family (parameter p) of functions f_p^{-1} to perform this task for the test-input data. At least the part of f_p^{-1} is implemented as an AutoFocus component that covers the test-input. Its semantics function is denoted F_p^{-1}. In case of a restriction to the input streams all instantiations of the component have the syntactic interface $(I_a \blacktriangleright I_c)$.

In accordance with the Galois connection f is in general too restrictive for comparing the expected-results of test-cases, thus, we use g or g^{-1} for this purpose.

3.4 Implementing Test-case Transformation and Checking

For a practical implementation in our tool the concretization is implemented using an AutoFocus component which corresponds to the relation RI. Checking the results of test-inputs is done using the relations RO. Both RI and RO and the related test-case concretization can be constructed based on the functions f and g and their inverses.

More concretely, for checking the output relation RO we realize an AutoFocus component and take a semantic function based on g or g^{-1} (if g^{-1} exists). The realization is shown in Figure 3. For convenience reasons, we embed it into a component which just checks that the output streams are in relation. Thus, this component has the syntactic interface $(O_a \times O_c \blacktriangleright \mathit{boolstream})$ that checks whether the output streams

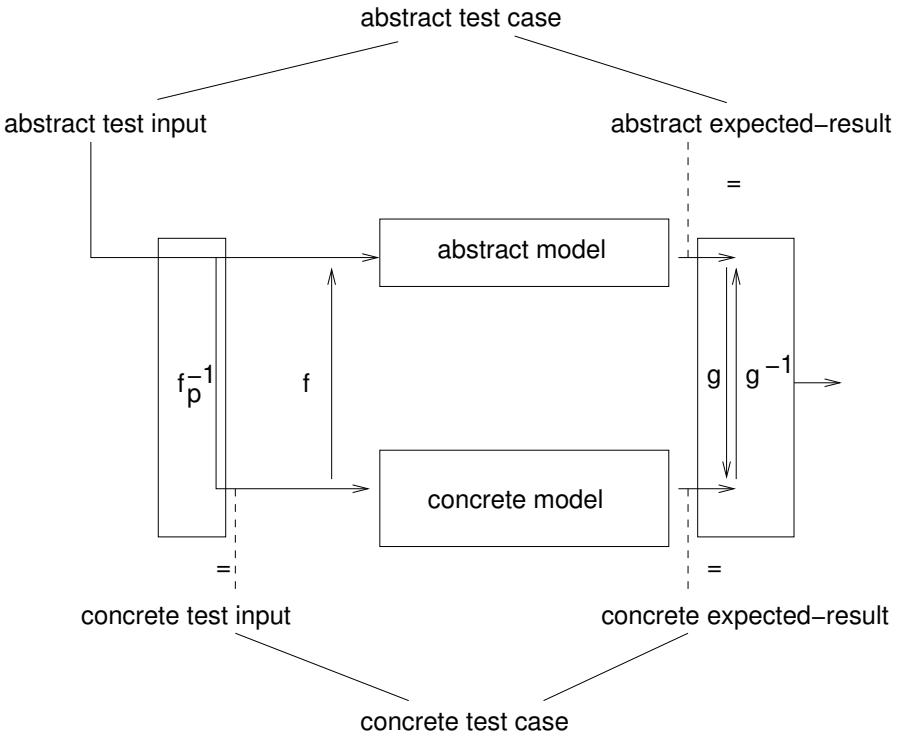


Figure 3: Realization of the testing scenario

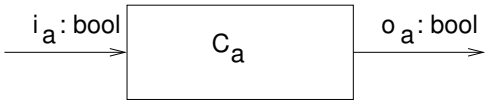


Figure 4: An abstract component

are compatible and emits a boolean value accordingly. The bool stream must be interpreted in a way, such that a single false occurring in the streams turns the entire comparison into false.

3.5 An Example

Consider the abstract component in Figure 4 and a concretization in Figure 5. The concrete component shall encode and abstract a 64 bit floating point value – e.g., an input from a sensor – into an 8 bit integer representation and the fact whether it is a positive or a negative number shall be preserved. One basic requirement specified by the abstract component could be that positive input values (including 0) are encoded into positive output values (including 0) and negative input values are encoded as negative output values.

Establishing the relation based approach For using the first approach, we can establish the two relations defined on sets of streams for the channels i_c, i_a, o_c, o_a:

$$\mathsf{RI}\,(I_a, I_c) = \forall\, t \in \mathbb{N}_+ \,,\, (\, I_a(i_a).t \wedge I_c(i_c).t \geq 0\,) \vee (\, \neg I_a(i_a).t \wedge I_c(i_c).t \leq 0\,)$$

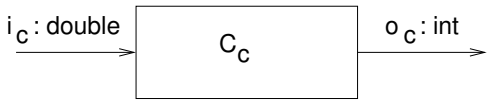


Figure 5: A concrete component

and

$$\mathsf{RO}\ (O_a, O_c) = \forall\, t\ \in\ \mathbb{N}_+\ ,\ (\ O_a(o_a).t\ \wedge\ O_c(o_c).t\ \geq\ 0\ )\ \vee\ (\ \neg O_a(o_a).t\ \wedge\ O_c(o_c).t\ \leq\ 0\ )$$

A test case with $[true; false; true]$ as input stream for the i channel will deliver the output $[true; false; true]$ for the abstract component. A correct concretization may be $[2.5; -3.6; 0.3]$ and might generate the (correct) output $[2; -4; 0]$ on the concrete component.

Using the Galois connection approach For the second approach we relate the two models by using the functions f and g.

For example f can be defined as:

$$f(T_c^{io}) = \{(abs(i_c), abs(o_c)) | (i_c, o_c) \in T_c^{io}\}$$

abs is a function that works on streams and abstracts the floating point and integer values to bools, in a way corresponding to the example above. While

$$f(\{([2.5; -3.6; 0.3], [2; -4; 0])\}) = \{([true; false; true], [true; false; true])\}$$

the opposite way allows in general multiple concretizations,

$$f_p^{-1}(T_a^{io}) = \{p(i_c, o_c) | (i_c, o_c) \in T_c^{io}\}$$

e.g. for a concrete implementation f,

$$f^{-1}(\{([true; false; true], [true; false; true])\}) = \{([2.5; -3.6; 0.3], [2; -4; 0])\}$$

In this case g can be realized in the following way

$$g(T_a^{io}) = \{(i_a, o_a) | \exists\ i_c\ o_c\ (abs(i_c), abs(o_c)) \in T_a^{io}\}$$

e.g.,

$$g(\{([true; false; true], [true; false; true])\}) =$$
$$\{([2.5; -3.6; 0.3], [2; -4; 0]), ([2.4; -3.8; 0.4], [2; -4; 0]), ([2.5; -3.6; 0.3], [2.5; -4.4; 0]), ...\}$$

Note that in general $f^-1 \subseteq g$ holds. The transformation of test-inputs and checking the correspondence of the results can be done using AutoFocus components build from f_p^{-1} and g functions.

Complexity of test-case comparison and limitations The proposed method uses entire abstract and concretized streams. Thus, components, that realize the abstraction and concretization functions may contain internal states if they are defined in an element-wise way on the streams.

Another characteristic is that input and output streams are regarded independently. This is justified by the fact that the relation between input and output streams is covered by the abstract model and correctness of test-cases and the concrete model is always regarded with respect to this abstract model.

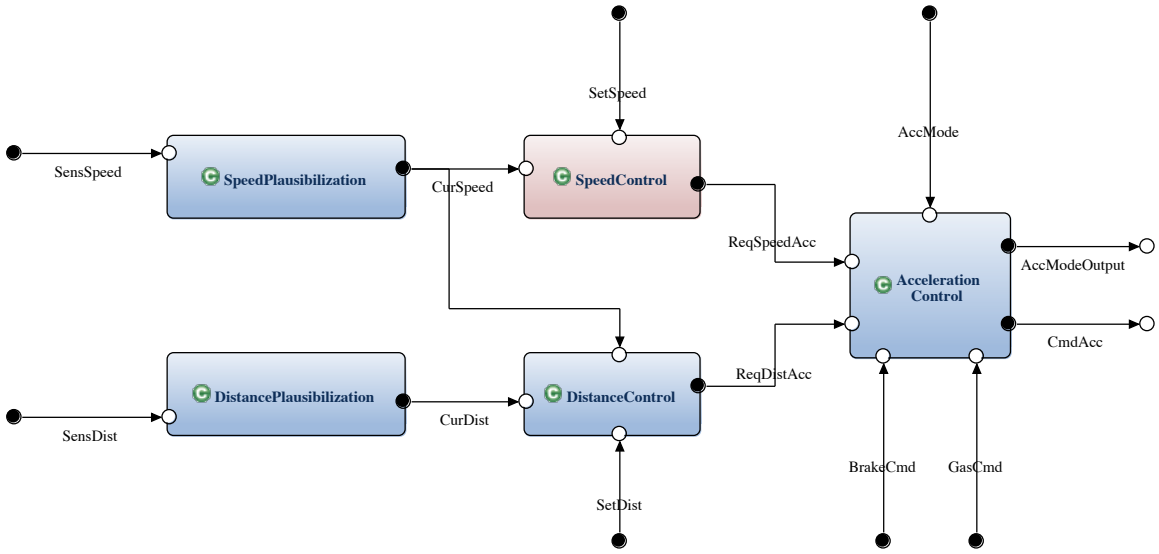


Figure 6: An ACC model in AutoFocus

4 ACC Case Study

We have chosen a simplified variant of an Adaptive Cruise Control System(ACC) as the case study. The ACC system controls the speed of the car and takes a desired default speed, a distance to the car in front and inputs like pressing the gas or brake pedal into account. The entire development process is carried out using AutoFocus 3. In particular, textual requirements are structured and specified, models are build from these requirements and test-cases are generated for these models. Several refined models for components are created in the development process using AutoFocus 3 until finally an implementation is created. Here, we start with requirements of an ACC systems which are taken from the ISO 15622 standard [8] "Intelligent transport systems – Adaptive Cruise Control systems – Performance requirements and test procedures".

Figure 6 shows the top-level component structure including input and output streams of the ACC that is regarded in this paper as realized in AutoFocus. This model works on several input streams: *SensSpeed* representing current speed values, *SensDist* representing the distance to the next car in front of the car with the ACC. Furthermore, streams that represent user input to set an acceleration mode (*AccMode*), a desired speed value (*SetSpeed*), a desired distance to the next car (*SetDist*), and pressing the brake or gas pedal (*BrakeCmd* and *GasCmd*) are included. Output streams comprise values for showing the current Mode (*AccModeOutput*) and an acceleration command which can be negative when braking (*CmdAcc*) or positive when an acceleration is considered favorable by the ACC.

The ACC model consists of 5 components. The SpeedPlausibilisation component (left top) and DistancePlausibilisation component (left bottom) measure takes the input speed and distance and delivers the calibrated values to the speed control component (middle top) and distance control component (middle bottom). Both components compute the suitable acceleration values according to the given speed and distance and send the results to the acceleration control component (right). When the acceleration control component is turned on and thus in active mode, it decides whose value should be respected. When it is switched to stand-by mode, both values will be discarded and only user's inputs are being processed. For safety consideration, drivers can provide their inputs any time (accelerate or brake) and

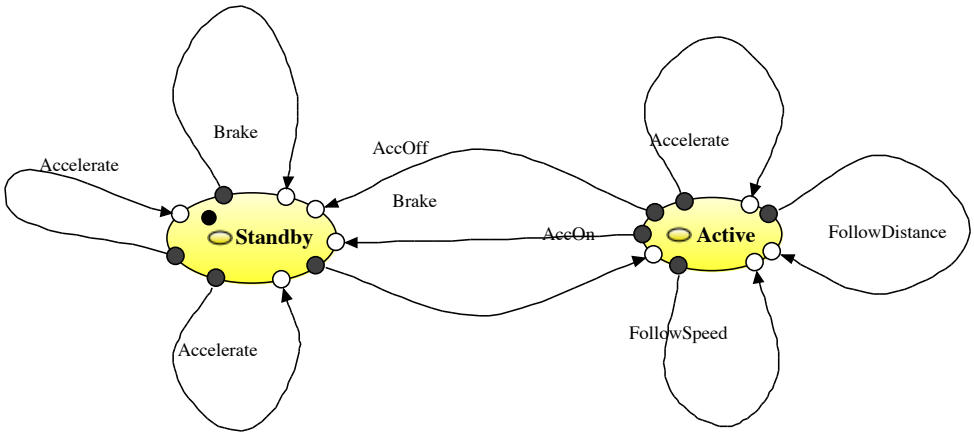


Figure 7: An abstract specification

this forces the acceleration control component to switch itself to stand-by mode.

Remember that components are stream processing functions and are composed from subcomponents. A simplified version of the ACC system can be downloaded with the standard AutoFocus 3 distribution [2].

One high-level behavioral specification of the acceleration control component is shown in Figure 7 and also modeled in AutoFocus 3. It consists of two states. The left state represents the Standby-Mode and the right state is for Active-Mode. The 3 transitions between the states represent the possible state switch of the component. The driver can turn on the ACC (Active-Mode) via a switch, which is represented by the AccOn transition, or turn off the ACC (Standby-Mode) any time by explicitly using the switch, which is the AccOff transition, or implicitly giving a manual brake command, which is the Brake transition. If the component is in the Active-Mode, the ACC system will control the car to follow the given speed and distance rules, if this is for some reason not desirable it does the acceleration according to the driver's command. If it is in the Standby-Mode, it only reacts on the driver's commands.

4.1 ACC Example Requirements

In the ISO 15622 standard requirements are explicitly given in the informal text format.

Here, we present two example requirements which we discuss in more details:

1. To ensure the driver can override the ACC, a requirement about the reaction of the ACC on the brake event is specified:
   Braking by the driver shall deactivate the ACC function at least if the driver initiated brake force demand is higher than the ACC initiated brake force.
2. Requirements can also imply constraints on relations between components:
   When the ACC is active, the vehicle speed shall be controlled automatically either to maintain a time gap to a forward vehicle, or to maintain the set speed, whichever speed is lower. The change between these two control modes is made automatically by the ACC system.

Other requirements posses a similar level of complexity.

In our approach, the requirement must first be formalized as a model. Figure 8 shows our first ACC requirement as an AutoFocus 3 component. In the upper part, the abstract component is shown,

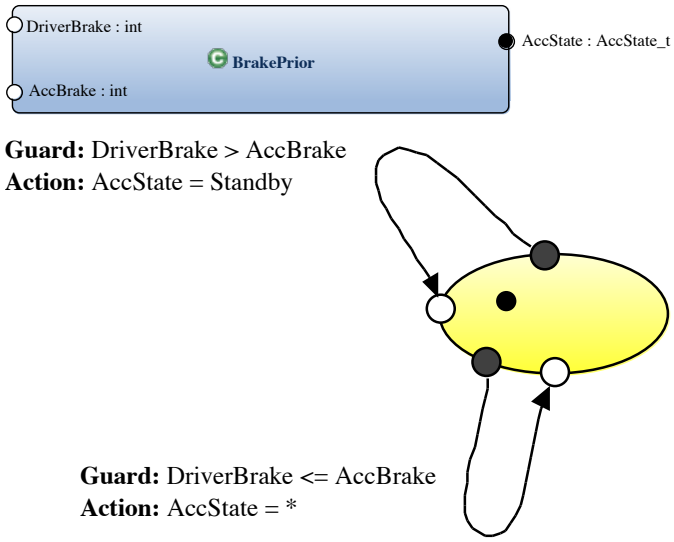


Figure 8: An ACC requirement in AutoFocus

the lower part defines a simple automaton. The abstract model has less input streams: *DriverBrake* indicating whether the brake padel is down and to what degree, *AccBrake* indicating the braking value currently calculated by some other ACC Components and AccState indicating whether the ACC shall currently actively control the car's speed. The first transition on the top left represents that the ACC system must be automatically set to Standby-Mode, whenever the driver gives brake command and the value is larger than the value computed by the ACC system. If the driver's brake value is smaller than the value computed by the ACC system or the driver gives no brake command, the system state is not concerned.

The second example requirement can be modeled with an AutoFocus component with two input ports $in_{distance}$, in_{speed} and one output port. The two input ports represent suitable acceleration values for maintaining a time gap ($in_{distance}$) and for maintaining speed set by the driver (in_{speed}). The output port represents the selected acceleration value according to the inputs. The internal behavior of the component can be described with just one formula: $out = min(in_{distance}, in_{speed})$.

4.2 Concretization of a Test-case

An example test-case about the requirement from Figure 8 comprises streams for

$$\text{DriverBrake } [21,51,78,100,91],\ \text{AccBrake } [79,100,100,91,51]$$

and the corresponding stream for AccState

$$[Active, Active, Active, Active, Standby].$$

Output of the stream processing function is delayed by one time step.

Relating abstract and concrete models Our formalizations of the f and f_p^{-1} (for different parameters) functions and the RI and RO relations realize the following constraints:

- Per time tick, the values of *AccBrake* are either equal to *ReqDistAcc* or *ReqSpeedAcc* (if we take the second requirement at this stage into account, we already know that it is advisable to chose the least of the two values per time tick),

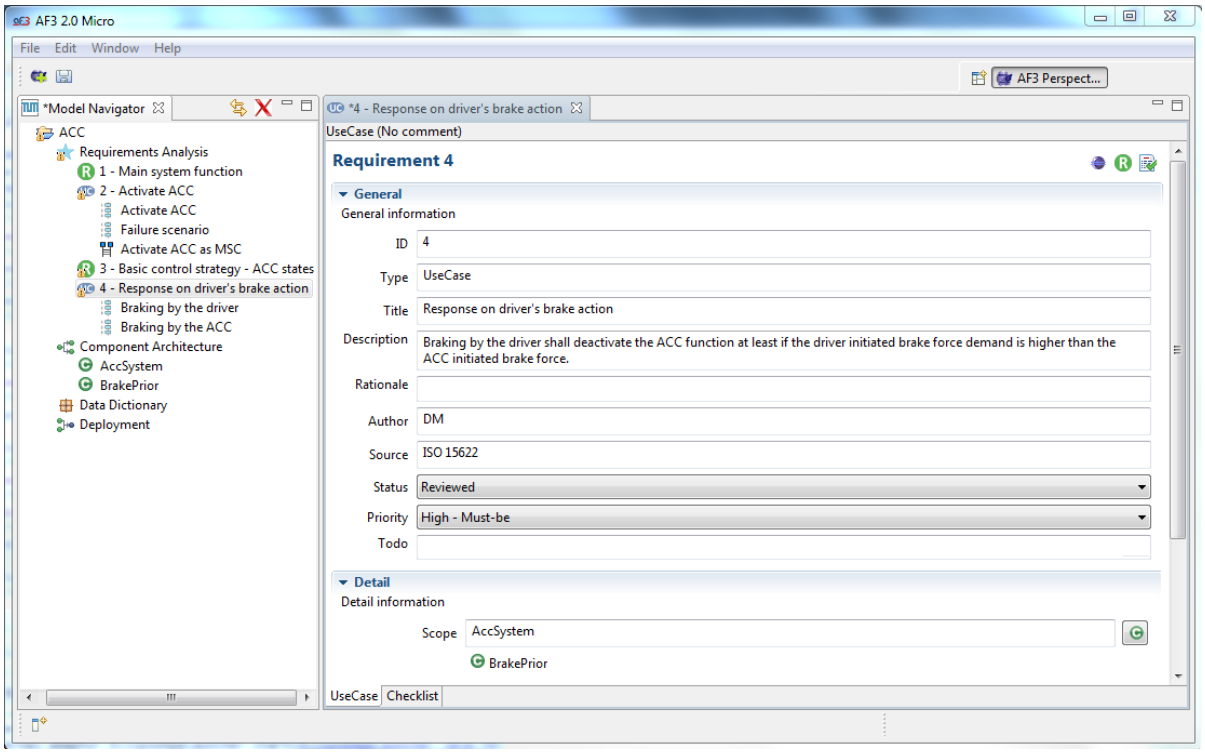


Figure 9: Textual requirement in AutoFocus 3

- *AccMode* does not have a correspondence in the abstract domain an thus does correspond to a parameter values,
- *BrakeCmd* in the concrete domain does correspond to the *AccState* stream in the abstract domain: the ACC must be switched to stand-by mode if a certain brake force is applied,
- *GasCmd* corresponds to a parameter value.

The g function is less restrictive than f_p^{-1}, e.g., most parameterized values may correspond to any possible parameter.

Comparing abstract and concrete test-results In our case study we created an AutoFocus component that abstracts the results of concrete case study output streams element wise. These are compared to the results of the abstract case study and a corresponding boolean value is emitted. Thus, we are computing g^{-1} and in some cases g.

4.3 Implementation and Evaluation

We have implemented a test-case generation and an abstraction framework for AutoFocus and evaluated it using the ACC case-study described in Section 4. AutoFocus 3 allows assigning models to textual requirements. A screen-shot of handling the first example requirement in our tool is shown in Figure 9. Figure 10 shows a screen-shot of our plug-in to specify and handle abstract test-cases and their concretization. Once one has specified components that realize f_p^{-1} and g^{-1} or g test cases can be handled

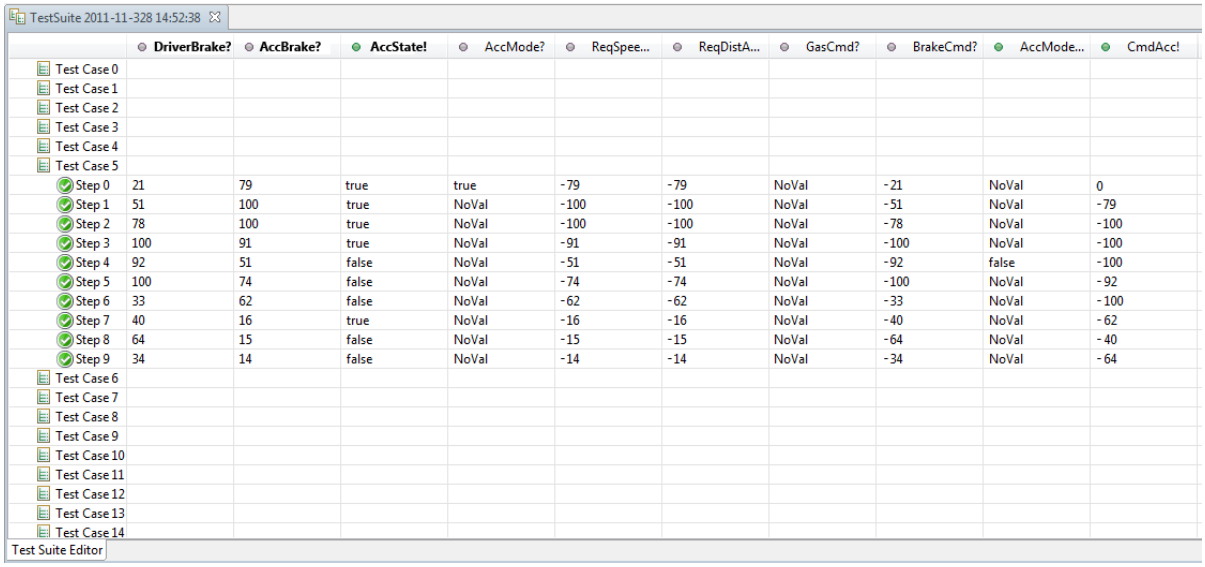


Figure 10: Handling a test-case in AutoFocus 3

in a user-friendly way: on the left side, one can select a test-case, which comprises values for different time steps, thus each column specifies an input stream.

Our implementation works together with an existing test-case generation tool for AutoFocus 3 described in [10]. This test-case generator uses Eclipse based constraint logic programming to derive test-cases. The test-case generator can be used to generate test-cases for the abstract model. Furthermore, the implementation can also be used to derive a family of different concrete test cases for a single abstract test-case.

Regarding our framework described in Section 3, for a requirement, we are stating the RI, RO relations from Section 3.2 or the Galois connection from Section 3.3 formally using a mathematical definition first. In case of using the Galois connection approach, we are implementing the f_p^{-1} (parameterized) and g functions for test-case abstraction by writing AutoFocus components that realize these functions or the required parts. It is important to notice that we can not simple replace f_p^{-1} by g or vice versa since $f_p^{-1} \subseteq g$ holds in general. The implemented components realizing f_p^{-1} and g in the regarded cases are relatively simple. In particular it is possible to derive f from f_p^{-1}. It is thus possible to verify the Galois connection property, thereby, guaranteeing that the concrete model is a refinement of the abstract model.

However, we do not have a formal framework for carrying out correctness proofs yet. Such a framework could be established using the semantics of AutoFocus and its formalization in a higher-order theorem prover and remains subject to future work.

5 Conclusion

In this paper we presented a framework for relating components at different abstraction levels with each other. We regarded transforming existing test-cases for abstract models to more concrete models and applying them to these concrete models in the context of our model-based development environment and tool AutoFocus 3. We described an evaluation of the approach using an ACC based case study. Our experiences indicate that reusing test-cases for different abstraction levels in a seemless model-based

development process is feasible for software development in embedded systems.

Different directions are interesting for future work. As next steps, we are interested to take more benefit from our work on relating abstract and concrete models using Galois connections. The investigation of compositionality aspects of tests for different components in a system is a goal. We are also interested in transforming other properties like, e.g., invariants from an abstract model to a concrete one (cf. [3] for a similar approach that we investigated).

As a more challenging long term goal, we are interested in automatically generating the AutoFocus components that relate abstract and concrete models with each other. Although this is an undecidable problem, some heuristics for relatively simple models would be a great benefit. An additional goal is a further improvement of the quality of test-case generation in AutoFocus.

References

[1] B. K. Aichernig. Test-design through abstraction – a systematic approach based on the refinement calculus. Journal of Universal Computer Science, / (8):710-735, August 2001, doi:10.3217/jucs-007-08-0710.

[2] The AutoFocus 3 Development Team, *http://af3.fortiss.org*.

[3] J. O. Blech, A. Hattendorf, J. Huang. An Invariant Preserving Transformation for PLC Models. IEEE International Workshop on Model-Based Engineering for Real- Time Embedded Systems Design, 2011, doi:10.1109/ISORCW.2011.46.

[4] M. Broy, I. H. Krüger, and M. Meisinger. A formal model of services. ACM Transactions on Software Engineering and Methodology, ACM 2007 (TOSEM), doi:10.1145/1189748.1189753.

[5] M. Broy, K. Stølen. Specification and development of interactive systems: FOCUS on streams, interfaces, and refinement. Springer-Verlag, 2001.

[6] P. Cousot and R. Cousot. Abstract Interpretation Based Program Testing. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 248 and electronic proceedings `http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`, L'Aquila, Italy, July 31 – August 6 2000. Scuola Superiore G. Reiss Romoli.

[7] A. Harhurin, J. Hartmann and D. Ratiu Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Systems. Technical Report, TUM-I0924, Technische Universität München, 2009.

[8] International Organization for Standardization (ISO). ISO 15622 standard: Intelligent transport systems – Adaptive Cruise Control systems – Performance requirements and test procedures.

[9] C. Loiseaux and S. Graf and J. Sifakis and A. Bouajjani and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. Formal Methods in System Design, 1995, doi:10.1007/BF01384313.

[10] C. Pfaller. Anforderungsorientierter modellbasierter Softwaretest reaktiver Systeme (Requirements-Oriented Model-Based Software Test of Reactive Systems) Phd thesis, Technische Universität München, 2010.

[11] W. Prenninger, A. Pretschner. Abstractions for Model-Based Testing. Proc. 2nd Intl. Workshop on Test and Analysis of Component Based Systems (TACoS'04), Barcelona, March 2004. Electronic Notes in Theoretical Computer Science 116:59–71, 2005, doi:10.1016/j.entcs.2004.02.086.

[12] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of model-based testing. Working paper series. University of Waikato, Department of Computer Science. No. 04/2006. Hamilton, New Zealand: University of Waikato. 2006.

[13] G. Yorsh, Th. Ball and M. Sagiv. Testing, abstraction, theorem proving: better together! Proceedings of the 2006 international symposium on Software testing and analysis. ACM, 2006 (ISSTA'06), doi:10.1145/1146238.1146255.

Applying SMT Solvers to the Test Template Framework

Maximiliano Cristiá
CIFASIS and UNR
Rosario, Argentina
cristia@cifasis-conicet.gov.ar

Claudia Frydman
LSIS-CIFASIS
Marseille, France
claudia.frydman@lsis.org

The Test Template Framework (TTF) is a model-based testing method for the Z notation. In the TTF, test cases are generated from test specifications, which are predicates written in Z. In turn, the Z notation is based on first-order logic with equality and Zermelo-Fraenkel set theory. In this way, a test case is a witness satisfying a formula in that theory. Satisfiability Modulo Theory (SMT) solvers are software tools that decide the satisfiability of arbitrary formulas in a large number of built-in logical theories and their combination. In this paper, we present the first results of applying two SMT solvers, Yices and CVC3, as the engines to find test cases from TTF's test specifications. In doing so, shallow embeddings of a significant portion of the Z notation into the input languages of Yices and CVC3 are provided, given that they do not directly support Zermelo-Fraenkel set theory as defined in Z. Finally, the results of applying these embeddings to a number of test specifications of eight cases studies are analysed.

1 Introduction

The Test Template Framework (TTF) is a model-based testing (MBT) method for the Z notation, specially well-suited for unit testing [30]. The Z notation is a formal specification language based on first-order logic with equality and Zermelo-Fraenkel set theory [29, 19]. Our group was the first in providing tool support for the TTF by implementing Fastest [11, 8], and in extending the TTF beyond test case generation [10, 9].

Within the TTF, each operation of a Z specification is analysed to produce test cases to later test its implementation. This analysis is performed by partitioning the input space of the operation. Partitioning, in turn, is conducted through the application of one or more testing tactics. Each element of the resulting partition is an equivalence class. In this context, equivalence classes are called *test classes*, *test objectives*, *test templates* or *test specifications* in the literature. The latter will be used in this paper. Test specifications obtained in this way can be further subdivided into more test specifications by applying other testing tactics. The net effect of this technique is a progressive partition of the input space into more restrictive test specifications. One of the features that makes the TTF particularly appealing for the Z community, is that all of its main concepts are expressed in Z. Hence, the engineer has to know only one notation to perform the analysis down to the generation of abstract test cases.

Each test specification is characterized by a Z predicate. Finding a test case for a test specification in the TTF means, thus, finding a witness satisfying its predicate. Clearly, this is a problem of satisfiability at the presence of a complex and rich mathematical theory. Currently, Fastest implements a rough algorithm to solve this problem [11]. On the other hand, Satisfiability Modulo Theory (SMT) solvers are tools that, precisely, solve the problem of satisfiability for a range of mathematical and logical theories [25]. In this paper, we present the first results of applying two SMT solvers, Yices [14] and CVC3 [2], to the problem of finding test cases from test specifications within the TTF.

Applying a SMT solver to this problem is not a trivial task, in part, due to the fact that, as far as we have investigated, currently there is no SMT solver natively supporting the Zermelo-Fraenkel set

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 28–42, doi:10.4204/EPTCS.80.3

theory. Hence, one needs to rest on defining a shallow embedding of that theory in the language of a SMT solver. In doing so, a key question arises: is the language of a SMT solver expressive enough to allow an embedding of Zermelo-Fraenkel set theory? Then, if that embedding is possible, is it the only one? Will the chosen embedding solve all the satisfiable test specifications appearing in the TTF and real Z specifications? Which SMT solver and which embedding will be the best in satisfying more test specifications in less time? Finally, some questions more specific to our project: will that SMT solver be better than Fastest in finding test cases? Or should the SMT solver complement Fastest in this task?

In this paper we give first answers to all these questions. In Section 2 we describe some issues about the Z notation that pose some requirements on the expressiveness of SMT solvers' languages. Section 3 shows the complexity of typical test specifications derived by applying the TTF, and Section 4 briefly describes the algorithm implemented in Fastest to search for test cases. A research plan for the application of SMT solvers to this problem is established in Section 5. Sections 6 and 7 present the embeddings for Yices and CVC3 and the results of an empirical assessment of them, respectively. Finally, in Sections 8 and 9 we compare our work with other approaches and give our conclusions.

2 The Z Notation

In this section we do not pretend to introduce the Z notation but only to highlight some peculiarities of its type system—for a comprehensive presentation of Z there are many fine textbooks [27, 21]. An important component of the Z notation is the Z Mathematical Toolkit (ZMT) [29]. The ZMT defines a number of mathematical data structures and operations on them. It contains all the elements of the Zermelo-Fraenkel set theory and other elements built on them. In this context, we will refer to the ZMT as a synonym of first-order logic with equality and Zermelo-Fraenkel set theory.

Z is a typed formalism. $\mathbb{Z}$ is the only built-in type in the language. Specifiers can introduce basic types as they wish by simply declaring them as: $[X]$. The structure of the elements of such a type are unknown. It is also possible to introduce so-called free types, which are recursive data types. In their simplest form they are just enumerations: $Y ::= y_1 | \ldots | y_n$.

Basically, Z has three type constructors. If X is a type, then $\mathbb{P}X$ builds the type of all the sets whose elements are of type X. If X and Y are types then, $X \times Y$ is the type of ordered pairs or Cartesian product[1]. Finally, if $X_1, \ldots, X_k$ are k types, then $[x_1 : X_1; \ldots; x_k : X_k]$ is the type of records whose fields are $x_1, \ldots, x_k$. In Z records are called schema types, or just schemas, and are central to the notation—they are used to specify states, operations, properties, etc. These type constructors can be applied recursively to form more and more complex types.

The ZMT also defines a number of synonyms for some important sets. The set of all binary relations between X and Y, noted as $X \leftrightarrow Y$, is defined as $\mathbb{P}(X \times Y)$. Furthermore, the ZMT next defines the set of partial functions from X to Y, $X \pfun Y$, and the set of total functions, $X \to Y$, as:

$$X \pfun Y == \{f : X \leftrightarrow Y | \forall x : X; \; y_1, y_2 : Y \bullet x \mapsto y_1 \in f \land x \mapsto y_2 \in f \Rightarrow y_1 = y_2\}$$

$$X \to Y == \{f : X \pfun Y | \mathrm{dom} f = X\}$$

That is, in Z functions are sets of ordered pairs. In other words, functions are built up from more basic elements, are not always total, they are extensional, and can be higher-order—i.e. a function can have another function as an argument. Being sets of ordered pairs, set theory operators can be applied to them: if $f : X \pfun Y$; $x : X$; $y : Y$ then the following are all type correct $x \mapsto y \in f, f \cup \{x \mapsto y\}, f \setminus \{x \mapsto y\}$,

[1] In Z an ordered pair is usually written as $x \mapsto y$ as a synonym of (x, y).

etc. However, they are also functions so function application is also defined: $f\,x = f(x)$. Therefore, Z functions have two characters: they are functions but they are also sets. The Z type system cannot guarantee that, for instance $f \cup \{x \mapsto y\}$, is still a partial function, it can only guarantee that it is a binary relation. Moreover, $f\,x$ is not always defined since it might be the case that $x \notin \mathrm{dom} f$. That is, Z cannot guarantee that function application is always correct. All this is crucial to the complexity of embedding Z in other languages because usually functions are types in their own and are total, like in Yices [14], sometimes they are just first-order objects like in Z3 [24] and they may be non-extensional and total like in CVC3 [2] and Coq [6].

$\mathbb{P}$ builds all the sets of a given type, both finite and infinite. Therefore, the ZMT defines the set of finite sets of a type:

$$\mathbb{F}X == \{S : \mathbb{P}X \mid \exists n : \mathbb{N} \bullet \exists f : 1\,..\,n \to S \bullet \mathrm{ran} f = S\}$$

over which the cardinality operator, #, can be applied. That is, # cannot be applied to $A : \mathbb{P}X$ unless you can prove that A is actually in $\mathbb{F}X$. The ZMT also defines the sets of finite partial functions and sequences:

$$X \twoheadrightarrow Y == \{f : X \nrightarrow Y \mid \mathrm{dom} f \in \mathbb{F}X\}$$
$$\mathrm{seq}X == \{f : \mathbb{N} \twoheadrightarrow X \mid \mathrm{dom} f = 1\,..\,\#f\}$$

The last issue we want to remark about the Z notation is that set theory operators are polymorphic. In other words, symbols like $\cup$, $\cap$, $\in$ and $\varnothing$, can be applied to any type.

3 Test Specifications and Test Cases in the TTF

Test specifications and test cases in the TTF are represented as Z schemas. In its more complex form a Z schema has two parts: the declaration part, where variables are declared; and the predicate part, where a predicate depending on that variables can be written. The next schemas are test specifications borrowed from two of our case studies:

$DetectReferenceEvent_{18}^{NR}$

$now, fa : \mathbb{N};\ ot : REVENT \nrightarrow \mathbb{N}$
$tli, tls, X : REVENT \to \mathbb{N}$
$sysState : STATUS;\ e? : REVENT$

$e? = LiftOff$
$sysState = normal$
$e? \notin \mathrm{dom}\, ot$
$now \in tli\, e?\,..\,tls\, e?$
$X\, e? \leq fa$
$ot \neq \varnothing$
$\{e? \mapsto now\} \neq \varnothing$
$ot \cap \{e? \mapsto now\} = \varnothing$
$1 < now < 3$

$RetrieveEData_{24}^{SP}$

$mem : \mathrm{seq}\, MDATA$
$m : \mathbb{N}$
$d? : \mathrm{seq}\, MDATA$

$43 < m + \#d?$
$mem \neq \varnothing$
$\{i : 1\,..\,\#d? \bullet m + i \mapsto d?\, i\} \neq \varnothing$
$\mathrm{dom}\, mem \cap \mathrm{dom}\{i : 1\,..\,\#d? \bullet m + i \mapsto d?\, i\} \neq \varnothing$
$\neg\ \mathrm{dom}\{i : 1\,..\,\#d? \bullet m + i \mapsto d?\, i\} \subseteq \mathrm{dom}\, mem$
$\neg\ \mathrm{dom}\, mem \subseteq \mathrm{dom}\{i : 1\,..\,\#d? \bullet m + i \mapsto d?\, i\}$

As it can be seen, test specifications are conjunctions of atomic predicates making heavy use of sets, functions, sequences and set theory operators. A test case is, then, a schema further restricting its

test specification so the declared variables can take only one value. For example, the following schema represents a test case generated from $DetectReferenceEvent_{18}^{NR}$.

$$
\begin{array}{|l}
\hline
DetectReferenceEvent_{18}^{TC} \\
\quad DetectReferenceEvent_{18}^{NR} \\
\hline
tli = \{LiftOff \mapsto 2, ThrustDrop1E \mapsto 5, ThrustDrop2E \mapsto 4, ThrustDrop3E \mapsto 10\} \\
tls = \{LiftOff \mapsto 10, ThrustDrop1E \mapsto 12, ThrustDrop2E \mapsto 14, ThrustDrop3E \mapsto 16\} \\
X = \{LiftOff \mapsto 3, ThrustDrop1E \mapsto 5, ThrustDrop2E \mapsto 7, ThrustDrop3E \mapsto 9\} \\
e? = LiftOff \\
sysState = normal \\
now = 2 \\
fa = 10 \\
ot = \{ThrustDrop1E \mapsto 3\} \\
\hline
\end{array}
$$

Note how schema inclusion is used to link a test case with its corresponding test specification.

4 A Simple Algorithm for Searching Test Cases

Before searching test cases from test specifications, Fastest's users can run a command to eliminate unsatisfiable test specifications. The method behind this command has been extensively described elsewhere [8]. This method has proved to be efficient and effective in eliminating most of the unsatisfiable test specifications. Hence, when users want to find test cases from test specifications, most of them are satisfiable. Fastest implements a very simple algorithm to search test cases from test specifications, which has been introduced in another paper [11]. At the time we started Fastest (early 2007) SMT solvers were not an option since most of them were being developed at the same time. Then, we implemented a primitive algorithm that can be regarded as a "brute force ZMT solver"[2]. Fastest builds a finite model for each test specification by calculating the Cartesian product between a very small finite set of values bound to each variable declared in the test specification. Later, Fastest evaluates the test specification for some elements in the finite model. These finite models are calculated by considering the following heuristics:

- Only the types of variables are considered when building the finite model; i.e. the structure of the predicate appearing in the test specification is not taken into account.
- There is a configuration variable, *FSS*, whose value sets the size of the finite sets for basic types, $\mathbb{Z}$ and $\mathbb{N}$. *FSS* must be strictly positive—usually it is 2 or 3.
- There is a configuration variable, *MAX*, whose value sets the maximum size for a finite model.
- The finite sets for types $\mathbb{N}$ or $\mathbb{Z}$ are built from the first *FSS* numerical constants appearing in the test specification. If there are no such constants then $[0\,..\,FSS-1]$ is chosen for $\mathbb{N}$ and $[-(FSS \operatorname{div} 2 + (FSS \bmod 2 - 1))\,..\,(FSS \operatorname{div} 2)]$ for $\mathbb{Z}$.
- The finite sets for enumerated types are their elements.
- The finite sets for basic types are built by generating *FSS* constant names of each type.
- If a variable declared in the test specification does not appear in its predicate, then the finite set for that variable is any singleton—since the value of such a variable has no influence whatsoever on the evaluation of the predicate.

[2]The 'Z' in 'ZMT solver' is not a mistake, but an indication that our algorithm is only for the Z Mathematical Toolkit.

- If the predicate of a test specification contains an atomic predicate of the form $var = val$, where *var* is a variable and *val* is a constant value, then the finite set for *var* is just $\{val\}$—since it will be impossible to satisfy the predicate with any other value.
- The finite sets for the types or sets that result from applying a type constructor or by following a ZMT definition—i.e. $\times$, $\mathbb{P}$, $\pfun$, etc.—to other types or sets, are built recursively from the finite sets considered for its arguments.
- Given that test specifications are conjunctions of atomic predicates, the algorithm reduces the initial finite model to the subset satisfying the first atomic predicate. This subset is used as the finite model on which the second atomic predicate is evaluated, and the algorithm reduces it once more to the subset satisfying this second predicate. This continues until the last atomic predicate is considered, in which case the first element satisfying it is returned; or until an atomic predicate cannot be satisfied, in which case "unknown" is returned.

  During this step: (a) *MAX* is considered to put a limit on the number of elements of the finite model to be explored; and (b) the evaluation of the predicate on a particular element of the finite model is performed by the ZLive component of the CZT project [15, 23].

Although this algorithm might appear inefficient and is certainly inelegant, it has proved to find an average of 80% of test cases from satisfiable test specifications [11]. However, SMT solvers can be good complements or alternatives to this algorithm, as we will show shortly.

5 Contribution of this Paper

Replacing the algorithm described in the previous section, is not a trivial task because, just to begin with, no SMT solver works directly with the Z notation. Therefore, at a bare least we need to write a translator from Z into the language of the chosen SMT solver, and another from the output language back to Z—for converting the witness found by the SMT solver into Z—, when even the subset of Z supported by Fastest is a complex language. Not to mention that it might be necessary to try out different SMT solvers with different shallow embeddings. Hence, we plan to attack this problem as follows:

1. Chose some SMT solvers that are powerful and stable as to be used for the problem at hand.
2. Define one or more shallow embeddings for them.
3. Apply the embeddings to the satisfiable test specifications that were not solved by Fastest.
4. Analyse the results.
5. If the combination of SMT solver and shallow embedding works well for these test specifications—i.e., it finds many test cases fast—, then see whether it also finds test cases for those test specifications for which Fastest works well.
6. Since ZLive has some limitations, see if the shallow embedding can overcome them.
7. Complete and optimise the embedding.
8. If everything goes well, write the traducers.
9. Measure the end-to-end computing time—i.e., translating from Z to the SMT solver, executing the SMT solver, and translating back the results to Z—to compare it with the current algorithm.
10. Investigate whether the SMT solver can be used to eliminate unsatisfiable test specifications.

Until step 8 all the work is manual and many alternatives should be constantly evaluated. For example, is a single SMT solver good enough? Is better to use many of them because some solve some test specifications while others solve the rest? Must the current algorithm be replaced or used as another solver? At the end, would it be better to write a decision procedure for the ZMT and include it in some SMT solver instead of using a shallow embedding?

In this paper we address steps 1 to 4. More specifically, the problem attacked in this paper is, thus, using SMT solvers to find witnesses satisfying those test specifications for which Fastest failed—two of which are shown in Section 3. Our contributions are: (i) defining shallow embeddings of a significant portion of the ZMT for two mainstream SMT solvers, namely Yices and CVC3; and (ii) running Yices and CVC3 on 69 satisfiable test specifications (borrowed from eight cases studies, three of which are real industrial problems) written with the shallow embeddings, to measure the efficiency and effectiveness of the embeddings and the SMT solvers for this particular testing problem. The embeddings shown in this paper are not only useful for our problem but also for others as they embed the Zermelo-Fraenkel set theory in general [22].

6 Shallow Embeddings of Z into Two SMT Solvers

In this section we present two shallow embeddings of a significant portion of the ZMT for Yices [14] and CVC3 [2]—in Section 8 we explain why we have chosen these two SMT solvers. The embeddings are given by means of embedding rules of the following form:

$$\text{name}\ \frac{\textit{Z notation}}{\mathsf{SMT\ solver\ syntax}}$$

where the text above the line is some Z term and the text below the line is one or more, either Yices or CVC3, sentences; the name of the rule identifies the Z term being considered. Some Z features are omitted because they are outside the scope of this paper; and some rules are not given because they can be easily deduced from the others (for example, we give a rule for set intersection but not for set union).

The files resulting from applying these embeddings to 69 test specifications along with the Z test specifications themselves are available at: `www.fceia.unr.edu.ar/~mcristia/smt-ttf-cs.tar.gz`.

6.1 Notation

We decided to describe the embeddings in terms of the input languages of Yices and CVC3 because we would like readers to be able to check all the empirical data mentioned in Section 7. We do not use the SMT-LIBv2 [4] language because it does not support all the features of all SMT solvers—for instance, Yices' lambda expressions and CVC3's instantiation patters. We believe that, in general, the input languages of both SMT solvers are rather easy to understand for readers knowledgeable in formal methods.

Yices uses a language similar to SMT-LIBv2. That is, operators and type constructors are all prefix. For instance, $x+y$ is written $(+\,x\,y)$, and a function from X to Y is declared as $(\rightarrow X\,Y)$. `nat`, `int` and `bool` are all built-in types, with their obvious meanings. Yices support lambda expressions to define functions, as in lambda calculus. The keyword `select` is used to access members of record-types; it is also a prefix operator.

CVC3 uses a more human-readable input language. All the reserved words are written in capitals. The most difficult construction is the definition of an array. If A is an array then it is possible to associate a value for each of its components by means of the construction $\mathsf{ARRAY}\,(x : T) : expr(x)$, where T is the

type of the indexes of A, and *expr* is an expression of the type of the components of A which may depend on x. The result is an array in which the value of the component with index x is the result of $expr(x)$.

6.2 Yices

The most relevant rules of the shallow embedding of Z into Yices[3] are given in Figure 1. We are going to discuss only those rules that deserve some attention. As it can be seen sets, functions, partial functions, binary relations, sequences and finite sets are all represented, essentially, with Yices uninterpreted functions. In Yices uninterpreted functions[4] are total, extensional and higher-order, making them a good choice to represent ZMT's mathematical structures. In our opinion, there are no other elements in Yices better than functions on which to build the embedding.

Basic types are embedded as type definitions thus preserving the Z semantics in that there is no clue about the structure of their elements. The embedding defines a set of type X as a function from X to bool. If $A : \mathbb{P}X$ and $x : X$, the interpretation is trivial: $x \in A \Leftrightarrow (A\,x)$. Note that these two rules imply that a set may be infinite. Also, note that Yices' type system impedes us to define polymorphic operators (cf. rules $\varnothing$, $\subseteq$, etc.). The workaround is to define one per type. This is not a serious problem because the intention is that the embedding will be transparent for Fastest users.

A partial function is represented as a record with two fields: *dom*, is a Yices function representing the domain of the function—as with sets—; and *law* is the actual map between the types. The intention is that $(law\,x)$ is meaningful if and only if $(dom\,x)$ is true. However, this intention cannot be guaranteed unless the Z specification is consistent—and not only type correct. If the Z specification is verified in a system like, for instance, Z/EVES [28], then some proof obligations should have been discharged proving that all partial functions are correctly applied. Besides, Fastest eliminates test specifications where a partial function is explicitly applied outside its domain—i.e., for example where $x \notin \mathrm{dom} f \wedge \ldots f\,x \ldots$ holds. Our embedding assumes these two hypothesis. This representation of partial functions has two advantages: (i) the domain is a set as in Z; and (ii) it is easy to apply a function to its argument. However, it has a disadvantage: (partial) functions are not sets; in other words, the embedding for (partial) functions is semantically different from the embedding for sets. For instance, if we have $f : X \pfun Y$ and $R : X \rel Y$, then at the Z level $f = R$ is type-correct, but its representation through the embedding is not. Nevertheless, this can be overcome by calculating the "set" corresponding to a (partial) function. For example (assuming $f : X \pfun Y$):

$$(\mathsf{define}\,fSet :: (\to X\,Y\,bool)\ (\mathsf{lambda}\,(x :: X\,y :: Y)\,(\mathsf{and}\,((\mathsf{select}\,f\,dom)\,x)\,(=((\mathsf{select}\,f\,law)\,x)\,y))))$$

Then, at the Yices level we can write $(=fSet\,R)$ for $f = R$, but we still use f for function application; for instance, $((\mathsf{select}\,f\,law)\,x = y_1)$. We have tried other representations for partial functions, sets and functions but in our opinion this is the best one for our purposes. For instance, the Yices' manual suggests representing partial functions through dependent types:

$$(\mathsf{define}\,f :: (\mathsf{tuple}\,dom :: (\to X\,bool)\,(\to (\mathsf{subtype}\,(x :: X)\,(dom\,x))\,Y)))$$

However, it has a problem in the context of embedding Z specifications. In effect, if $x : X$ then $f\,x$ is type-correct in Z, but $((\mathsf{select}\,f\,2)\,x)$ is not in Yices—because the type of x is X and not $(\mathsf{subtype}\,(x :: X)\,(dom\,x))$. This representation may be good for other theories of partial functions.

[3] Actually we used Yices 1.

[4] From now on we will just say "functions".

$$\mathbb{Z}\ \frac{\mathbb{Z}}{\mathsf{int}} \qquad \mathbb{N}\ \frac{\mathbb{N}}{\mathsf{nat}} \qquad \text{basic types}\ \frac{[X]}{(\mathsf{define}-\mathsf{type}\,X)}$$

$$\text{free types}\ \frac{X ::= c_1 | \ldots | c_n}{(\mathsf{define}-\mathsf{type}\,X\,(\mathsf{scalar}\,c_1 \ldots c_n))} \qquad \times\ \frac{x : Y \times Z}{(\mathsf{define}\,x :: [Y,Z])}$$

$$\mathbb{P}\ \frac{A : \mathbb{P}\,X}{(\mathsf{define}\,A :: (\rightarrow X\,\mathsf{bool}))} \qquad \text{ranges}\ \frac{a\,..\,b}{(\mathsf{lambda}\,(i :: \mathsf{int})\,(\mathsf{and}\,(\leq a\,i)\,(\leq i\,b)))}$$

$$\leftrightarrow\ \frac{R : X \leftrightarrow Y}{(\mathsf{define}\,R :: (\rightarrow X\,Y\,\mathsf{bool}))} \qquad \rightarrow\ \frac{f : X \rightarrow Y}{(\mathsf{define}\,f :: (\rightarrow X\,Y))}$$

$$\nrightarrow\ \frac{f : X \nrightarrow Y}{(\mathsf{define}\,f :: (\mathsf{record}\,dom :: (\rightarrow X\,\mathsf{bool})\,law :: (\rightarrow X\,Y)))}$$

$$\varnothing\ \frac{\varnothing : X}{(\mathsf{define}\,emptysetX :: (\rightarrow X\,\mathsf{bool})\,(\mathsf{lambda}\,(x :: X)\,\mathsf{false}))}$$

$$\cap\ \frac{A,B : \mathbb{P}\,X \qquad A \cap B}{\begin{array}{l}(\mathsf{define}\,capX :: (\rightarrow (\rightarrow X\,\mathsf{bool})\,(\rightarrow X\,\mathsf{bool})\,(\rightarrow X\,\mathsf{bool}))\\ \quad (\mathsf{lambda}\,(A :: (\rightarrow X\,\mathsf{bool})\,B :: (\rightarrow X\,\mathsf{bool}))\,(\mathsf{lambda}\,(x :: X)\,(\mathsf{and}\,(A\,x)\,(B\,x)))))\end{array}}$$

$$\subseteq\ \frac{A,B : \mathbb{P}\,X \qquad A \subseteq B}{\begin{array}{l}(\mathsf{define}\,subseteqX :: (\rightarrow (\rightarrow X\,\mathsf{bool})\,(\rightarrow X\,\mathsf{bool})\,\mathsf{bool})\\ \quad (\mathsf{lambda}\,(A :: (\rightarrow X\,\mathsf{bool})\,B :: (\rightarrow X\,\mathsf{bool}))\,(\mathsf{forall}\,(x :: X)\,(\Rightarrow (A\,x)\,(B\,x)))))\end{array}}$$

$$\mathbb{F}\ \frac{A : \mathbb{F}\,X}{\begin{array}{l}(\mathsf{define}\,A :: (\mathsf{record}\,set :: (\rightarrow X\,\mathsf{bool})\,bij :: (\rightarrow X\,\mathsf{nat1})\,card :: \mathsf{nat}))\\ (\mathsf{assert}\,(\mathsf{forall}\,(x :: X)\,(\Longleftrightarrow ((\mathsf{select}\,A\,set)\,x)\,(\leq ((\mathsf{select}\,A\,bij)\,x)\,(\mathsf{select}\,A\,card)))))\\ (\mathsf{assert}\,(\mathsf{forall}\,(n :: \mathsf{nat1}\,x_1 :: X\,x_2 :: X)\\ \quad (\Rightarrow (\mathsf{and}\,(\leq n\,(\mathsf{select}\,A\,card))\\ \qquad\qquad ((\mathsf{select}\,A\,set)\,x_1)\\ \qquad\qquad ((\mathsf{select}\,A\,set)\,x_2)\\ \qquad\qquad (= ((\mathsf{select}\,A\,bij)\,x_1)\,n)\\ \qquad\qquad (= ((\mathsf{select}\,A\,bij)\,x_2)\,n))\\ \qquad\quad (= x_1\,x_2))))\end{array}}$$

$$\mathsf{seq}\ \frac{s : \mathsf{seq}\,X}{\begin{array}{l}(\mathsf{define}\,s :: (\mathsf{record}\,dom :: (\rightarrow \mathsf{nat1}\,\mathsf{bool})\,law :: (\rightarrow \mathsf{nat1}\,X)\,card :: \mathsf{nat}))\\ (\mathsf{assert}\,(\mathsf{forall}\,(n :: \mathsf{nat1})\,(\Longleftrightarrow (\leq n\,(\mathsf{select}\,s\,card))\,((\mathsf{select}\,s\,dom)\,n))))\end{array}}$$

Figure 1: Embedding rules for Yices.

As it can be seen, finite sets are harder to represent. We embed them by representing the definition of finiteness given in Section 2—i.e., a set has cardinality n if there is a bijection between itself and the first n natural numbers. That is, a finite set is a record with three fields: *set*, is the actual set; *bij*, is intended to be a bijection from a subset of its domain onto a subset of its range; and *card*, is the cardinality of the set. To keep *set* finite and consistent with the other two fields we assert two axioms. The first one says that an element is in *set* if and only if $bij\,x$ is less than or equal to *card*. This ensures that the image of *bij* for those $x : X$ such that $set\,x$ is true, has a finite number of elements. The second axiom asserts that the inverse of *bij* for all the natural numbers less than or equal to *card*, is a function. Therefore, *bij* is a bijection between the range $[1\,..\,card]$ and all $x : X$ such that $set\,x$ is true. Observe, that this representation is compatible with the one for sets. In effect, if $A : \mathbb{P}X$; $B : \mathbb{F}X$ and $B \subseteq A$, then at the Yices level we can simply say $(subseteqX\,(\mathsf{select}\,B\,set)\,A)$ (cf. rule $\subseteq$).

The rules in Figure 1 are completed by a rule saying that each Z atomic predicate appearing in a test specification must be embedded as an `assert` command. This is justified because a test specification is a conjunction of atomic predicates and a sequence of `assert` commands is also a conjunction. Therefore, checking the satisfiability of a test specification is performed by executing a `check` command.

6.3 CVC3

The most relevant rules of the shallow embedding of Z into CVC3 are given in Figure 2. Due to space restrictions we write BV1 for $\mathsf{BITVECTOR}(1)$, 0 for 0bin0 and 1 for 0bin1. As it can be seen, the embedding is essentially the same to the previous one, the main difference being that it uses arrays instead of functions. Although CVC3 supports functions, they are not extensional nor higher-order making them less useful to represent the ZMT. On the other hand, CVC3 provides a general theory of extensional, higher-order arrays. In particular they can be indexed by any type, finite or infinite. Therefore, in this case we opted for arrays as the main mathematical structure for the embedding. For sets and the like we used arrays of bit vectors of size one, because in CVC3 arrays cannot have Boolean components. This makes the embedding more verbose than the one for Yices. Note, however, that we have used functions for defining set theory operators like intersection and subset. We believe that this embedding deserves no further comments due to its similarities with respect to the previous one.

6.4 A Variant

Besides the embeddings shown in Figures 1 and 2, we also tried out a variant for each of them. In this variant the rules for basic types are replaced by the following ones:

$$\text{Yices}\,\frac{[X]}{(\mathsf{define}-\mathsf{type}\,X\,(\mathsf{scalar}\,x_1\,x_2\,x_3))} \qquad \text{CVC3}\,\frac{[X]}{\mathsf{DATATYPE}\,X = x_1|x_2|x_3\,\mathsf{END}}$$

In other words, a basic type is replaced by a type with only three values. Fastest proceeds in a similar fashion as we have explained in Section 4. Given that the elements of a basic type have an uncertain structure, we have observed that in many test specifications there is no need in having all of them. Changing the rules in this way may have a great impact on the effectiveness of the SMT solvers because all the quantifications over X become finite. It is a known fact that SMT solvers turn out to be incomplete at the presence of quantifications over infinite sets. Therefore, in this way it may be possible to avoid a number of such quantifications thus increasing the likelihood of finding more test cases.

$$\mathbb{Z}\frac{\mathbb{Z}}{\mathsf{INT}} \qquad \text{basic types}\frac{[X]}{X : \mathsf{TYPE}}$$

$$\text{free types}\frac{X ::= c_1|\ldots|c_n}{\mathsf{DATATYPE}\, X = c_1|\ldots|c_n\, \mathsf{END}} \qquad \times\frac{x : Y \times Z}{x : [Y,Z]}$$

$$\mathbb{P}\frac{A : \mathbb{P}X}{A : \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}} \qquad \text{ranges}\frac{a\,..\,b}{[a..b]}$$

$$\leftrightarrow\frac{R : X \leftrightarrow Y}{A : \mathsf{ARRAY}\, [X,Y]\, \mathsf{OF}\, \mathsf{BV1}} \qquad \rightarrow\frac{f : X \rightarrow Y}{A : \mathsf{ARRAY}\, X\, \mathsf{OF}\, Y}$$

$$\mathbb{N}\frac{\mathbb{N}}{\mathsf{NAT} : \mathsf{TYPE} = \mathsf{SUBTYPE}(\mathsf{LAMBDA}\,(x : \mathsf{INT}) : 0 \leq x)}$$

$$\twoheadrightarrow\frac{f : X \twoheadrightarrow Y}{f : [\#\, dom : \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}, law : \mathsf{ARRAY}\, X\, \mathsf{OF}\, Y\, \#]}$$

$$\varnothing\frac{\varnothing : X}{emptysetX : \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1} = (\mathsf{ARRAY}\,(y : Y) : 0)}$$

$$\cap\frac{A,B : \mathbb{P}X \qquad A \cap B}{\begin{array}{l} capX : (\mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}, \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}) \rightarrow \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1} \\ \mathsf{ASSERT}\, \mathsf{FORALL}\,(A,B : \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}) : \\ \quad capX(A,B) = (\mathsf{ARRAY}\,(x : X) : \mathsf{IF}\, A[x] = 1\, \mathsf{AND}\, B[x] = 1\, \mathsf{THEN}\, 1\, \mathsf{ELSE}\, 0\, \mathsf{ENDIF}) \end{array}}$$

$$\subseteq\frac{A,B : \mathbb{P}X \qquad A \subseteq B}{\begin{array}{l} subseteqX : (\mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}, \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}) \rightarrow \mathsf{BOOLEAN} \\ \mathsf{ASSERT}\, \mathsf{FORALL}\,(A,B : \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}) : \\ \quad subseteqINT(A,B) \iff \mathsf{FORALL}\,(x : \mathsf{INT}) : A[x] = 1 \Rightarrow B[x] = 1 \end{array}}$$

$$\mathbb{F}\frac{A : \mathbb{F}X}{\begin{array}{l} A : [\#\, set : \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}, bij : \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{NAT1}, card : \mathsf{NAT}\, \#] \\ \mathsf{ASSERT}\, \mathsf{FORALL}\,(x : X) : A.set[x] = 1 \iff A.bij[x] \leq A.card \\ \mathsf{ASSERT}\, \mathsf{FORALL}\,(n : \mathsf{NAT1}, x_1, x_2 : X) : \\ \quad (n \leq A.card\, \mathsf{AND}\, A.set[x_1] = 1\, \mathsf{AND}\, A.set[x_2] = 1\, \mathsf{AND}\, A.bij[x_1] = n\, \mathsf{AND}\, A.bij[x_2] = n) \\ \quad \Rightarrow x_1 = x_2 \end{array}}$$

$$\text{seq}\frac{s : \mathrm{seq}\, X}{\begin{array}{l} s : [\#\, dom : \mathsf{ARRAY}\, X\, \mathsf{OF}\, \mathsf{BV1}, law : \mathsf{ARRAY}\, \mathsf{NAT1}\, \mathsf{OF}\, X, card : \mathsf{NAT}\, \#] \\ \mathsf{ASSERT}\, \mathsf{FORALL}\,(n : \mathsf{NAT1}) : n \leq s.card \iff s.dom[n] = 1 \end{array}}$$

Figure 2: Embedding rules for CVC3.

Case study	Embeddings of Figure 1 and 2				Variant described in Section 6.4			
	Yices		CVC3		Yices		CVC3	
	Sat	**Unk**	**Sat**	**Unk**	**Sat**	**Unk**	**Sat**	**Unk**
Savings accounts (3)	8		8		8		8	
Savings accounts (1)		2		2		2		2
Launching vehicle		8	8			8	8	
Plavis		29		29		29		29
SWPDC		13		13		13		13
Scheduler		4		4		4		4
Security class		4		4		4		4
Pool of sensors	1		1		1		1	
Totals	9	60	17	52	9	60	17	52

Table 1: Results of running Yices and CVC3 on 69 test specifications.

7 Empirical Assessment

Since we started with the Fastest project we used a number of case studies (Z specifications) to test and validate different aspects of the tool [11, 8, 12, 10, 7]. At the moment we have eleven case studies to test the test case generation algorithm described in Section 4. Fastest finds 100% of the test cases for two of the eleven case studies. Of the remaining nine, we discarded one for the present experiments because it has very long test specifications as to write them by hand. Therefore, we assessed Yices and CVC3 and the embeddings with test specifications from eight case studies. The 69 test specifications chosen for this assessment are those for which Fastest was unable to find a test case, although they are satisfiable. The satisfiability of these test specifications was determined by manual inspection.

All these experiments were conducted on the following platform: an Intel Centrino Duo of 1.66 GHz with 1 Gb of main memory, running Linux Ubuntu 10.04 LTS with kernel 2.6.32-35-generic. As we have said, the original Z test specifications, and their translation to Yices and CVC3 can be downloaded from `http://www.fceia.unr.edu.ar/~mcristia/smt-ttf-cs.tar.gz`. The translation of each test specification is saved in a file ready to be loaded into Yices or CVC3. We also provide scripts to run the experiments. The results can be analysed with simple `grep` commands.

The first experiment started by manually writing each test specification according to the embedding rules shown in Figures 1 and 2. Then Yices and CVC3 were fed with each of them followed by a check-sat command. The output was redirected to files to be analysed later. The second experiment consisted in applying the variant embedding described in Section 6.4. After a check-sat command they both return either "satisfiable" or "unknown"—because "unsatisfiable" is impossible as all the test specifications are satisfiable. In both cases "unknown" means that the SMT solver cannot decide whether the formula is satisfiable or not. When the answer is "satisfiable" both solvers return a witness satisfying the formula. Furthermore, if the answer is "unknown" they return a "potential witness". That is, they are not sure whether the formula is satisfiable or not but they "believe" it is and return a possible witness.

The results of these experiments are shown in Table 1. Column **Sat** (**Unk**) is the number of test specifications for which the SMT solver returned "satisfiable" ("unknown"). As it can be seen, the variant embedding produced exactly the same results for both SMT solvers. Also it is easy to see that CVC3 discovered all the test cases discovered by Yices plus eight more. However, while Yices, in both experiments, took no more than 3 seconds in processing the 69 test specifications, CVC3 took around 7 minutes in doing the same. In turn, Fastest takes 6.5 minutes to process the same test specifications, but,

as we have already said, it discovers no test case. Yices could not solve all the test specifications that include a quantification or a lambda expression over an infinite set; CVC3 could not solve all the test specifications that include a quantification over an infinite set. It is very important to remark that these quantifications or lambda expressions appear due to the embeddings; they are not present in the original Z test specifications. The conclusions about these experiments are listed in Section 9.

8 Related Work

We chose Yices and CVC3 for this work after evaluating all the SMT solvers that participated in the SMT-COMP 2010, 2009 and 2008, that is 22 tools [4, chapter 5]. The evaluation was based on the following criteria: (i) the tool must be documented as to be used by a novice user, specifically its input language must be thoroughly described; (ii) the tool must be stable and actively developed; (iii) the tool must run on Linux; (iv) the tool must be clearly identified as a SMT solver, specifically it must return the witness satisfying a formula; (v) the tool must be freely available to the general public; and (vi) the tool must work with a general logical system, in particular it must support: (a) quantified formulas; (b) basic and enumerated types; and (c) a general theory of extensional uninterpreted functions or arrays (index and values over general types). This evaluation yielded only three candidates: Yices, CVC3 and Z3 [24]. In this paper we report on the results of applying Yices and CVC3; Z3 will be approached soon. Most of the evaluated SMT solvers do not support quantified formulas over a sufficiently general mathematical theory. veriT supports such theories but it does not provide a witness if a formula was satisfied [13]; Alt-Ergo looks powerful as to fulfil our needs but it is not documented as to start the project with it [5]. None of the evaluated SMT solvers implement decision procedures for a theory of sets.

Model-based testing (MBT) techniques and tools for constraint solving or satisfiability have been integrated, and SMT developers have proposed to use their tools for test case generation. The results of some of these works encouraged us to follow the same ideas but applied to the TTF and Z. For example, Leonardo de Moura, in a tutorial given at Automated Formal Methods (AFM) 2006, includes test case generation as one of the applications for Yices. Different people at Microsoft Research have integrated Z3 into MBT or testing tools. Veanes and colleagues [32] use Z3 to generate test cases for parametrized SQL queries. In this work the authors use a language which supports finite sets, but not the other ZMT elements. Pex [31] and SAGE [17] are testing tools developed at Microsoft Research which integrate Z3. The first one generates unit tests for .NET applications, and the second one is a fuzz tester for security vulnerabilities. Grieskamp et al. [18] use Spec Explorer integrated with Z3 to generate combinations of parameter values. This parameters appear in the actions of labelled transition systems abstracting the system-under-test. Besides, Galler et al. [16] integrate Z3 in jCAMEL so it can derive test cases for programs annotated with contracts. However, Z3 is used only for integer parameters. As it can be seen, these works deal with formalisms quite different from and usually less expressive than Z.

The satisfiability algorithm presented in Section 4 is similar in conception to approaches like the Alloy Analyser which also defines a finite model for a given predicate and tries to see whether is it possible to satisfy it within this model or not [20]. The Alloy Analyser uses some SAT solving techniques.

Peleska et al. [26] apply the SONOLAR SMT solver for generating test cases from a modelling formalism based on Harel's Statecharts. This formalism is less expressive than Z and does not include any set or related theory. SONOLAR is one of the SMT solvers we evaluated but we could not use it since it does not support quantified formulas nor a general theory of arrays or uninterpreted functions.

Kröning et al. [22] propose to add a new theory to the SMT-Lib standard [1], as the standard format for formulas involving sets and finite sets, mappings and lists. Their proposal originates in VDM but

they acknowledge that it can be applied to other formalisms such as Z. However, they do not give the embedding of this theory in the language of any concrete SMT solver; they suggest that arrays can be used to encode it. Besides, the theory described in that work is not exactly Z—for example, they deal with finite mappings and not functions as in Z.

9 Conclusions and Future Work

In this paper we have proposed shallow embeddings for two SMT solvers, Yices and CVC3, as a method for finding test cases from Z test specifications. These test specifications are generated by Fastest, a tool implementing the Test Template Framework. Given that these test specifications are predicates of first-order logic and Zermelo-Fraenkel set theory, SMT solvers looked as promissory tools to solve this problem. Besides, we experimented with these embeddings and the SMT solvers by manually codifying 69 satisfiable test specifications. Based solely on these experimental results we can conclude:

- CVC3 works better than Yices, as the former found around the double of test cases.
- Given that CVC3 discovered exactly the same test cases than Yices, combining both tools does not seem to be fruitful, unless Yices is used first given that it runs faster than CVC3.
- However, CVC3 discovered test cases for only 25% of the test specifications. It seems a poor result since it outperforms the rough algorithm implemented by Fastest in only 17 test cases. Assuming CVC3 would also discover all the test cases that Fastest currently does—we are not sure it will, though—, it would be an overall increment of 4%, given that we work with 475 satisfiable test specifications. Furthermore, the time spent by CVC3 and Fastest in procssing these 17 test cases is roughly the same.
- An issue that deserves more attention is the chances of using the potential witnesses returned by the SMT solver when the answer is "unknown". After a manual inspection we observed that many of them are indeed witnesses. It might be possible to invoke ZLive to confirm that these witnesses satisfy their corresponding test specifications. This constitutes a first sign that combining Fastest with SMT solvers may be a good option.
- The previous item brings in another issue. Is it trivial to automatically translate back to Z the witnesses returned by the SMT solver? At a first glance, the witnesses returned by Yices are far easier to parse than those returned by CVC3—actually Yices returned a total of 1,221 lines of text while CVC3 returned 33,145 lines; the main difference lies in the potential witnesses: less than 1,000 lines for Yices and more than 32,000 lines for CVC3.
- Replacing the embedding of Z basic types by enumerated types (as described in Section 6.4) proved to be useless, in spite of looking promising at first. The problem may lay in the fact that this variant still produces formulas with quantifications over the integers or the naturals—i.e. infinite quantifications. It does not seem promising to change $\mathbb{Z}$ or $\mathbb{N}$ for a finite subset—like, for instance $[-10\,..\,10]$—because each literal has its own properties. For example, if a test specification mentions "43" then not considering it in some way may lead to an "unsatisfiable" answer. This is not the case for basic types, as all of their elements have only one property: equality.

In summary, we will keep exploring combining SMT solvers with Fastest since they discovered some test cases that Fastest did not, their execution times are at least as good as Fastest's, and there are chances that potential witnesses become more test cases. Our next step is to see whether the embedding for CVC3

finds all the test cases that Fastest currently finds and study the translation of the witnesses returned by CVC3. If the results of this step are not good, then we will consider proposing a decision procedure for formulas of the ZMT. We also plan to repeat the work reported here with the Z3 SMT solver.

References

[1] Clark Barrett, Aaron Stump & Cesare Tinelli (2010): *The SMT-LIB Standard: Version 2.0*. Technical Report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[2] Clark Barrett & Cesare Tinelli (2007): *CVC3*. In Werner Damm & Holger Hermanns, editors: *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, *Lecture Notes in Computer Science* 4590, Springer-Verlag, pp. 298–302. Berlin, Germany.

[3] Karin Breitman & Ana Cavalcanti, editors (2009): *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*. *Lecture Notes in Computer Science* 5885, Springer. Available at `http://dx.doi.org/10.1007/978-3-642-10373-5`.

[4] David R. Cok (2011): *The SMT-LIBv2 Language and Tools: A Tutorial*. GrammaTech, Inc.

[5] Sylvain Conchon & Evelyne Contejean: *Alt-Ergo*. Available at `http://alt-ergo.lri.fr`. Last access: November 2011.

[6] Coq Development Team (2008): *The Coq Proof Assistant Reference Manual, Version 8.2*. LogiCal Project.

[7] Maximiliano Cristiá, Pablo Albertengo, Claudia Frydman, Brian Plüss & Pablo Rodríguez Monetti (2011): *Applying the Test Template Framework to Aerospace Software*. In: *Proceedings of the 34th IEEE Annual Software Engineering Workshop*, IEEE Computer Society, Limerik, Irland. — to be published.

[8] Maximiliano Cristiá, Pablo Albertengo & Pablo Rodríguez Monetti (2010): *Pruning Testing Trees in the Test Template Framework by Detecting Mathematical Contradictions*. In José Luis Fiadeiro & Stefania Gnesi, editors: *SEFM*, IEEE Computer Society, pp. 268–277.

[9] Maximiliano Cristiá, Diego Hollmann, Pablo Albertengo, Claudia S. Frydman & Pablo Rodríguez Monetti (2011): *A Language for Test Case Refinement in the Test Template Framework*. In Shengchao Qin & Zongyan Qiu, editors: *ICFEM*, *Lecture Notes in Computer Science* 6991, Springer, pp. 601–616. Available at `http://dx.doi.org/10.1007/978-3-642-24559-6_40`.

[10] Maximiliano Cristiá & Brian Plüss (2010): *Generating Natural Language Descriptions of Z Test Cases*. In John D. Kelleher, Brian Mac Namee, Ielka van der Sluis, Anja Belz, Albert Gatt & Alexander Koller, editors: *INLG*, The Association for Computer Linguistics, pp. 173–177. Available at `http://www.aclweb.org/anthology/W10-4218`.

[11] Maximiliano Cristiá & Pablo Rodríguez Monetti (2009): *Implementing and Applying the Stocks-Carrington Framework for Model-Based Testing*. In Breitman & Cavalcanti [3], pp. 167–185. Available at `http://dx.doi.org/10.1007/978-3-642-10373-5_9`.

[12] Maximiliano Cristiá, Valdivino Santiago & N.L. Vijaykumar (2010): *On Comparing and Complementing two MBT approaches*. In Fabián Vargas & Erika Cota, editors: *LATW*, IEEE Computer Society, pp. 1–6.

[13] David Déharbe & Pascal Fontaine: *The veriT Solver*. Available at `http://www.verit-solver.org`. Last access: November 2011.

[14] Bruno Dutertre & Leonardo de Moura (2006): *System Description: Yices 1.0*. In: *Proceedings of the 2nd SMT competition, SMT-COMP'06*, Seattle, USA.

[15] Leo Freitas, Mark Utting, Petra Malik & Tim Miller: *Community Z Tools (CZT) Project*. Available at `http://czt.sourceforge.net`. Last access: November 2011.

[16] Stefan J. Galler, Bernhard Peischl & Franz Wotawa (2008): *Challenging Automatic Test Case Generation Tools with Real World Applications*. In: *Proceedings of the IASTED International Conference on Software Engineering and Applications*, pp. 21–26.

[17] Patrice Godefroid: *SAGE*. Available at `http://channel9.msdn.com/blogs/peli/automated-whitebox-fuzz-testing-with-sage`. Last access: November 2011.

[18] Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof & Myra B. Cohen (2009): *Interaction Coverage Meets Path Coverage by SMT Constraint Solving*. In: *Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, TESTCOM '09/FATES '09, Springer-Verlag, Berlin, Heidelberg, pp. 97–112. Available at `http://dx.doi.org/10.1007/978-3-642-05031-2_7`.

[19] ISO (2002): *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics*. Technical Report ISO/IEC 13568, International Organization for Standardization.

[20] Daniel Jackson (2006): *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[21] Jonathan Jacky (1996): *The way of Z: practical programming with formal methods*. Cambridge University Press, New York, NY, USA.

[22] Daniel Kröning, Philipp Rümmer & Georg Weissenbacher (2009): *A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard*. In: *Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE 22*.

[23] Petra Malik & Mark Utting (2005): *CZT: A Framework for Z Tools*. In Helen Treharne, Steve King, Martin C. Henson & Steve A. Schneider, editors: *ZB*, *Lecture Notes in Computer Science* 3455, Springer, pp. 65–84. Available at `http://dx.doi.org/10.1007/11415787_5`.

[24] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *TACAS*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340. Available at `http://dx.doi.org/10.1007/978-3-540-78800-3_24`.

[25] Robert Nieuwenhuis, Albert Oliveras & Cesare Tinelli (2006): *Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)*. *J. ACM* 53, pp. 937–977. Available at `http://doi.acm.org/10.1145/1217856.1217859`.

[26] Jan Peleska, Elena Vorobev & Florian Lapschies (2011): *Automated test case generation with SMT-solving and abstract interpretation*. In: *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, Springer-Verlag, Berlin, Heidelberg, pp. 298–312. Available at `http://dl.acm.org/citation.cfm?id=1986308.1986333`.

[27] B. Potter, D. Till & J. Sinclair (1996): *An introduction to formal specification and Z*. Prentice Hall PTR Upper Saddle River, NJ, USA.

[28] Mark Saaltink (1997): *The Z/EVES System*. In J.P. Bowen, M.G. Hinchey & D. Till, editors: *ZUM '97: The Z Formal Specification Notation*, pp. 72–85.

[29] J. M. Spivey (1992): *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

[30] P. Stocks & D. Carrington (1996): *A Framework for Specification-Based Testing*. *IEEE Transactions on Software Engineering* 22(11), pp. 777–793.

[31] The Pex Team: *Pex*. Available at `http://research.microsoft.com/en-us/projects/pex/`. Last access: November 2011.

[32] Margus Veanes, Pavel Grigorenko, Peli de Halleux & Nikolai Tillmann (2009): *Symbolic Query Exploration*. In Breitman & Cavalcanti [3], pp. 49–68. Available at `http://dx.doi.org/10.1007/978-3-642-10373-5_3`.

Exact Gap Computation for Code Coverage Metrics in ISO-C

Dirk Richter
Martin-Luther-University of Halle-Wittenberg, Germany
richterd@informatik.uni-halle.de

Christian Berg
Martin-Luther-University of Halle-Wittenberg, Germany
christian.berg@student.uni-halle.de

Test generation and test data selection are difficult tasks for model based testing. Tests for a program can be meld to a test suite. A lot of research is done to quantify the quality and improve a test suite. Code coverage metrics estimate the quality of a test suite. This quality is fine, if the code coverage value is high or 100%. Unfortunately it might be impossible to achieve 100% code coverage because of dead code for example. There is a gap between the feasible and theoretical maximal possible code coverage value. Our review of the research indicates, none of current research is concerned with exact gap computation. This paper presents a framework to compute such gaps exactly in an ISO-C compatible semantic and similar languages. We describe an efficient approximation of the gap in all the other cases. Thus, a tester can decide if more tests might be able or necessary to achieve better coverage.

1 Introduction

Tests are used in model based testing to identify software defects. High quality test generation and test data selection can be difficult tasks when the test has to satisfy a lot of requirements or cannot be created automatically because of the undecidability of the halting problem in Turing powerful languages. Given requirements for a test suite (set of tests) are functional or non-functional (e.g. execution times, runtime, usage of memory, correctness or a minimum value of a code coverage metric). Code coverage metrics quantify the quality of a test suite rather imprecisely and guide testers only. There is a gap between the feasible and theoretical maximal possible code coverage value. Sometimes demanded requirements are unsatisfiable because of gaps. Unnecessary additional tests will be computed while not all requirements are satisfied. This enlarges the test suite and introduces redundancy. Fortunately these problems (caused by metric imprecisions) can be solved by computing these gaps, which is not possible for Turing powerful languages in general. Therefore this paper presents suitable models in a new C-like syntax. These models allow to use an ISO-C compatible semantic. In this paper we show how to compute such gaps exactly for these models using formal verification techniques resp. software model checking ideas. The paper is organized as follows: at first we clarify basics and used notations; we then present our framework, apply it to some common coverage metrics and illustrate this on some examples. Finally we discuss related work and present a summary and conclusions.

2 Basics

2.1 Code Coverage Metrics γ

Let $T_P = 2^{tests}$ be the set of all possible sets of tests and P a program written in a common programming language such as C, C++ or Java. Each $t = \{\alpha_1, \alpha_2, ...\} \in T_P$ is a test suite with tests α_i for program P. The function $\gamma^P : T_P \to [0,1]$ is a code coverage metric, if γ^P is monotonically increasing. The program P can be omitted, if it is well-defined by the context.

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 43–57, doi:10.4204/EPTCS.80.4

In this paper some common code coverage metrics for functions, statements, decisions, branches and conditions will be considered as examples. Other ones (e.g. linear code sequence and jump coverage, jj-path coverage, path coverage, entry/exit coverage or loop coverage) can be adapted in a similar way.

The function coverage metric $\gamma_f^P(t) := |func(t)|/|func(P)|$ is the ratio of functions $func(t)$ that has been called in the test suite t, to all functions $func(P)$ in P [12].

The statement coverage metric $\gamma_s^P(t) := |stats(t)|/|stats(P)|$ is the ratio of statements $stats(t)$ that has been executed in the test suite t, to all statements $stats(P)$ in P [12]. To distinguish the same statement s on different program points l_1 and l_2, we annotate each statement s with unique labels l_1 and l_2 from program P, so that $l_1 : s \in stats(P)$ and $l_2 : s \in stats(P)$.

```
1  char x,y;
2  void b() {
3   lb0: if (y<x)return;
4   lb1: x=y/x;
5   lb2: return; }
6
7  void c() {
8   lc0: if (x == 0)
9          lc1: x = 10;
10    else lc2: b();
11  lc3: return; }
12
13 void a() {
14  la0: x=x+1;
15  la1: c();
16  la2: return; }
17
18 void main() {
19  l0: x=0;
20  l1: y=1;
21  l2: b();
22  l3: a();
23  l4: return; }
```

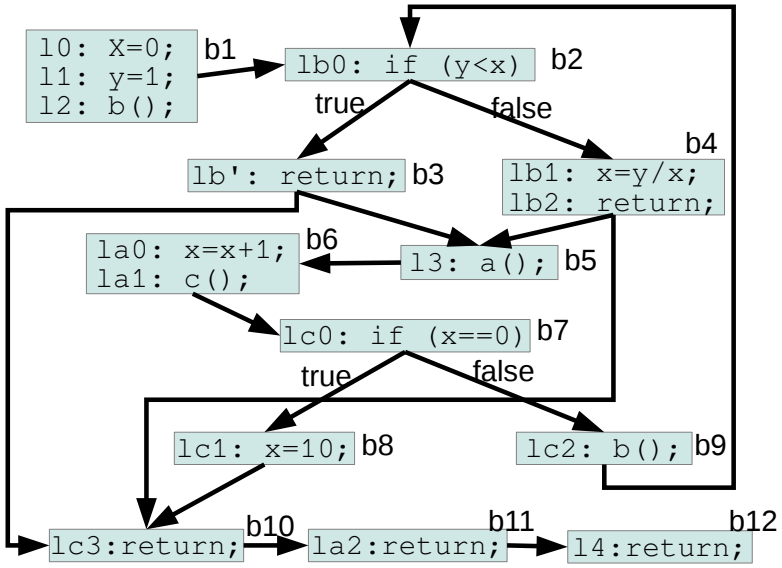


Figure 1: SPDS example P_1 in ISO-C syntax and corresponding BBICFG

Let $blocks(P)$ be all basic blocks [1] in P. Every program point has a surrounding basic block. The basic block inter-procedural control flow graph $BBICFG_P = (blocks(P), edges(P))$ (see Fig. 1) consists of the basic blocks $blocks(P)$ as nodes and edges $edges(P) \subseteq blocks(P)^2$, where $(b_1, b_2) \in edges(P)$ iff there is an execution path of length 1 from the end of block b_1 to the entry of block b_2 (execution of the last statement of block b_1). The decision coverage metric $\gamma_d^P(t) := |edges(t)|/|edges(P)|$ is the ratio of executed edges $edges(t)$ of the control flow graph $BBICFG_P$ for t, to all edges $edges(P)$ in P.

The branch coverage metric $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$ is the ratio of basic code blocks $blocks(t)$ executed during test suite t, to all basic blocks $blocks(P)$ in P [13]. Even if all basic blocks are covered by test suite t and $\gamma_b^P(t) = 1$, there can be uncovered branching edges in the basic block inter-procedural control flow graph $BBICFG_P$. Thus $\gamma_d^P(t) < 1$ is possible in this case.

Let $bExpr(l)$ be the set of all Boolean sub-expressions on label l of program P and

$$BExpr(P) := \{(l, bExpr(l)) \bullet l \in labels(P)\}. \tag{1}$$

The condition or predicate coverage metric $\gamma_c^P(t) := |exval(t,P)|/(2 \cdot |BExpr(P)|)$ is the ratio of evaluations of boolean sub-expressions $exval(t,P) \subseteq BExpr(P) \times \{true, false\}$ of the test suite t, to all evaluations of boolean sub-expressions in P [12]. The relation $exval(t,P)$ describes the evaluations of sub-expressions e on label l under test suite t, such that $((l,e),true) \in exval(t,P)$ iff there is a test $\alpha \in t$ where e can be evaluated to $true$ on label l under test α. When boolean operations are not short circuited, condition coverage does not necessarily imply decision coverage.

2.2 Code Coverage Metric Gap δ

Let $\gamma : T_P \to [0,1]$ be a code coverage metric for a program P. The code coverage metric gap $\delta_\gamma(P) \in [0,1]$ is the smallest difference between the coverage ratio of a test suite $t \in T_P$ and the theoretical maximal value 1:

$$\delta_\gamma(P) := \inf_{t \in T_P} (1 - \gamma(t)). \tag{2}$$

Let $\delta_x(P)$ denote $\delta_{\gamma_x}(P)$, where $x \in \{c,d,s,f,b\}$. Obviously dead code can cause $\delta_\gamma(P) > 0$. If some evaluations of boolean (sub-)expressions cannot be realized, $\delta_\gamma(P) > 0$ is possible without dead code (e.g. condition or decision coverage). In Turing powerful programming languages the gap $\delta_\gamma(P)$ can not be computed in general, because the halting problem is undecidable. In this case the gap $\delta_\gamma(P)$ can be approximated only. We show how to compute the exact gap $\delta_\gamma(P)$ for an ISO-C compatible semantic by adequate modeling.

2.3 Suitable Models

A more expressive model describing the program behaviour allows for a more accurate approximation of the gap $\delta_\gamma(P)$. If the model is not Turing powerful and the model behaviour is equivalent to the program behaviour, the gap $\delta_\gamma(P)$ can be exactly computed. Therefore we defined an ISO-C compatible semantic using pushdown systems (PDS) [9]. The split of the ISO-C language definition into platform-independent semantics and platform-specific semantics has a serious implication for deciding the halting problem of ISO-C programs: whether a C program halts or not depends on the platform-specific semantic. Thus, even though the halting problem for a C program is decidable for a platform-specific semantic, the halting property can become undefined if no specific platform is assumed [9]. Now we present an extension of the PDS used in [9] to symbolic pushdown systems (SPDS) using an ISO-C like syntax. SPDS use a more compact representation and define the PDS configurations and transitions symbolically.

A SPDS is a tuple $S = (vgbl, func)$, where $vgbl$ is a finite set of variables (global variables in ISO-C) and $func$ is a set of functions (pairwise different names) with an initial function $main \in func$. Each variable v has an integer type[1] $bits(v) \in \mathbb{N}_{\geq 1}$ and a fixed length $len(v) \in \mathbb{N}_{\geq 1}$. Every variable v is an array. A function is a tuple $(f, param, vlcl, stats)$, where $(f, param)$ is a function signature with a unique function name f and a finite list of parameter variables $param$. The set $vlcl$ is a finite set of variables (local variables in ISO-C), such that $param \subseteq vlcl$. The body of f is a finite list of statements $stats$. Each statement $l : s \in stats$ has a unique label $l \in labels(f)$, and $fst(f) \in labels(f)$ is the label of the first statement in the list $stats$. We use $vgbl = vgbl(S), func = func(S), param = param(f), vlcl = vlcl(f)$

[1] The Boolean constants $false$ and $true$ are represented via 0 and $\neq 0$ like ISO-C.

and $stats = stats(f)$ respectively, if S or f are well-defined by the context. Let denote $func(l)$ the function f for which $l \in labels(f)$. Further let be

$$vars := vgbl \cup \bigcup_{f \in func} vlcl(f)\ ,\ stats(S) := \bigcup_{f \in func} stats(f) \text{ and } labels(S) := \bigcup_{f \in func} labels(f). \quad (3)$$

Similar to ISO-C the SPDS variables are used to build expression $Expr$ using constants and operators. The priority and associativity are the same as in ISO-C.

An expression $e \in Expr$ can be strictly evaluated to an integer number $[\![e]\!]_g^{c_f} \in \mathbb{Z} \cup \{\bot\}$ using valuation functions for global and local variables $g : vgbl \times \mathbb{Z} \to \mathbb{Z}$ and $c_f : vlcl(f) \times \mathbb{Z} \to \mathbb{Z}$. The symbol $\bot$ denotes arithmetic exception (e.g. division-by-zero or index-out-of-bounds). The functions $g(v,i)$ and $c_f(v,i)$ return the current value of variable v at index i (value of $v[i]$). The evaluation functions g and c_f can be omitted, if they are well-defined by the context. A variable usage $v[i]$ of variable $v \in vars$ with index $i \in \mathbb{Z}$ is evaluated as

$$[\![\ v[i]\ ]\!]_g^{c_f} := \begin{cases} g(v,i) & v \in vgbl \wedge 0 \le i < len(v) \\ c(v,i) & v \in vlcl(f) \wedge 0 \le i < len(v) \\ \bot & otherwise. \end{cases} \quad (4)$$

For $e \in Expr$ and a statement $l : s \in stats(f)$, s has one of the following forms:

- $v[e_1] = e_2$; corresponds to writing the value $[\![e_2]\!]$ into the variable v at index $[\![e_1]\!]$.
- $f(v_1, \ldots, v_n)$; corresponds to a function call (call by value), iff $(f, param)$ is a signature, where $param = [p_1, p_2, \ldots, p_n]$, $v_i \in vars$ and $bits(v_i) \le bits(p_i)$ for all $1 \le i \le n$.
- $return$; corresponds to a function return.
- $if\ (e)\ goto\ l'$; corresponds to a conditional jump[2] to label $l' \in labels(f)$.

The exception of a dynamic type mismatch occurs for "$v[e_1] = e_2$;" and the system terminates, if $[\![e_2]\!] = \bot$ or the type $bits(v)$ is too small to store the value $[\![e_2]\!]$ or $[\![e_1]\!] \notin \{0, 1, \ldots, len(v)\}$. We denote $v = e$ for $v[0] = e$ and v for usages of $v[0]$ to emulate syntactically non-array variables. The system terminates on statement "$if\ (e)\ goto\ l'$;" too, if $[\![e]\!] = \bot$. The predefined function $rand(e)$ returns a random number between 0 and $[\![e]\!]$ for $e \in Expr$, whereby $rand(\bot) = \bot$. Further ISO-C statements and variations for other languages can be mapped to these basic statements in the modeling phase. All variables (global and local) are uninitialized and have initially a random value. A test α for S is a subset of global variables with predefined values for label $fst(main)$. A configuration $s = (g, [(l_n, c_n), (l_{n-1}, c_{n-1}), \ldots, (l_1, c_1)])$ of S represents a state of the underlying Kripke structure with the current execution label $l_n \in labels$, the valuation $g : vgbl \times \mathbb{Z} \to \mathbb{Z}$ of global variables and the stack content. The stack content consist of a list of function calls with current execution labels $l_i \in labels(S)$ and valuations for local variables $c_i : vlcl(func(l_i)) \times \mathbb{Z} \to \mathbb{Z}$. The head of s is $head(s) = (g, (l_n, c_n))$. The set of all possible configurations is $conf(S)$. A *run* of S is a sequence of consecutive configurations beginning with an initial configuration $(g_{init}, [fst(main), c_{init}]) \in conf(S)$. SPDS are (like PDS) not Turing powerful and can be used to model the behaviour of (embedded) ISO-C programs. There is no restriction on the recursion depth.

[2] intra-procedural

3 Exact Gap Computation Framework

Let $\gamma : T_P \to [0,1]$ be a code coverage metric for a program P. Our framework to compute the gap $\delta_\gamma(P)$ consists of the following steps:

1. If necessary, create a SPDS model S with ISO-C compatible semantic for program P.
2. Modify the model S to a SPDS model S' to enable gap analysis for the code coverage metric γ.
3. Compute exact variable ranges for some new variables in S'.
4. Conclude the exact size of the gap $\delta_\gamma(S)$ in S for the code coverage metric γ.
5. Conclude the size of the gap $\delta_\gamma(P)$ in P.

3.1 SPDS Modeling (step 1)

If the given program P is not written in ISO-C (e.g. Java) or P has another platform-specific semantic, we create a SPDS model S for P by abstraction. Otherwise the behaviour of S is the same of P by mapping all the ISO-C statements to the basic SPDS-statements of section 2.3 using abbreviations (described in this section). Java can be handled using the tool JMoped [7]. Often other languages and corresponding statements can be mapped to the basic SPDS-statements in a similar fashion. For simplification we present some common mappings, which are abbreviations for previously defined basic SPDS-statements. We sketch the ideas only, because of limited space. Fig. 1 shows the SPDS example P_1 in ISO-C syntax, where "`char x`" in line 1 is an abbreviation for "`int x(8)[1]`" to declare an integer array of type $bits(x) = 8$ and $len(x) = 1$.

Omitted Returns and Labels: If there is no return statement at the end of a function body, its existence is assumed during interpretation of the symbolical description of S. The same holds for statements without labels, such that each statement in the SPDS has a unique label after interpretation.

Parameter Expressions: Basic SPDS-statements allow variables to be passed as parameters in function calls. We can simulate to pass expressions by temporary local variables. Let "$l : f(e_1, e_2, \ldots, e_n);$" be a function call with expressions $e_i \in Expr$, where $(f, param)$ is a signature with $param = [p_1, p_2, \ldots, p_n]$. We introduce new local SPDS-Variables $pe_i \notin vars$ with type $bits(pe_i) = bits(p_i)$ and $len(pe_i) = 1$. These variables are used to evaluate the expressions before the function call: $pe_i = e_i$. Instead of e_i now pe_i is passed to f using the basic SPDS-statement $f(pe_1, pe_2, \ldots, pe_n)$. The function call "$l : f(e_1, e_2, \ldots, e_n)$" is interpreted as "$l : pe_1 = e_1; pe_2 = e_2; \ldots pe_n = e_n; f(pe_1, pe_2, \ldots, pe_n)$". Now only basic SPDS-statements are used. The code coverage metrics are adapted accordingly. For example the statements $pe_i = e_i;$ are ignored for the statement coverage metric.

Return Values: A function can return a value. This value can be used to set a variable "$v = f(...);$". If a function returns an expression e via "$return\ e$", a new global variable $ret_f \notin vars$ is introduced. The type of ret_f equals the return type of function f and $len(ret_f) = 1$. The statement "$return\ e;$" is interpreted as "$ret_f = e; return;$". On the other hand the assignment "$v = f(...);$" is interpreted as "$f(...); v = ret_f;$" to store the return value of f in v.

Function Calls in Expressions: If there is a function call $f(..)$ in an expression e, this function is evaluated in a temporary local variable. Boolean operations in SPDS are strict and not short circuited. Short circuited expressions (also increments `i++` and decrements `i--`) can be mapped to strict expressions without side effects by several conditional statements. Thus every function call $f(..)$ in an expression e will be definitively evaluated during the evaluation of e. Accordingly it is safe to do every call before the evaluation of e. Sometimes the order of this evaluation is implementation defined (as in ISO-C) and depends on the source language (e.g. `i++*i++`). Thus we use priority and associativity for calculating

this order. The intermediate representation of a compiler can be used, too, to achieve a mapping to SPDS.
Unconditional Jump: The statement "$goto\ l$;" is mapped to "$if\ (1)\ goto\ l$;". The dead branching edge to the following statement is ignored by the decision coverage etc.
Skip Statement: Particularly low level languages have often a "no operation" statement. We use the statement *skip*, which does not change variable settings. The statement "$l : skip$;" can be interpreted using the conditional branch "$l : if\ (0)\ goto\ l$;". If there is a global variable $v \in vars$, this can also be interpreted as "$l : v = v$;". The former needs no consideration for the coverage metrics.
Random Numbers: In ISO-C the function $rand()$ returns a pseudo random value between 0 and the constant $RAND\_MAX$, where $RAND\_MAX$ depends on the system. We can map this behaviour using the SPDS function $rand(RAND\_MAX)$. A similar mapping for random numbers is possible in other languages.
Conditional Statements: Let $s1$ and $s2$ be lists of statements. The conditional statement "$if\ (e)\ s1\ else\ s2$;" is interpreted as "$if\ (e)\ goto\ l_1; s2; goto\ l'; l_1 : s1; l' : skip$;", where $l_1, l' \notin labels$. "$if\ (e)\ s$;" is an abbreviation for "$if\ (e)\ s\ else\ skip$;".
Local Variable Definitions: A local variable can be defined during an assignment of a basic block or a loop header. Such local variable definitions are mapped to local variables of the surrounding function. Renaming can be done easily if necessary.
Loops: A for-loop of the form "$for\ (init;\ cond;\ inc)\ body$;" is interpreted as "$init; l : body; inc; if\ (cond)\ goto\ l$;", where $l \notin labels$. The *do* and *while* loops are interpreted in a similar way.
Modular Arithmetic and Integer Overflow: The ISO-C standard says that an integer overflow causes "undefined behaviour", meaning that compilers conforming to the standard can generate any code: from completely ignoring the overflow to aborting the program. Our solution terminating the system is conform to the ISO-C standard. Evaluations of expressions in SPDS are not restricted to arithmetic bounds, but dynamic type mismatches are possible for assignments $v = e$. In the case of modeling nonterminating modular arithmetic the modulo operator % can be used to shrink the expression e to fit the size $bits(v)$. Hence, a dynamic type mismatch does not occur.
Dynamic Memory and Pointers: In ISO-C a certain amount of the heap can be reserved using the function $malloc(int)$. It returns an address on the heap. The heap is finite, because the number of addresses is finite. This behaviour is simulated using a global array *heap* of type $bits(heap) = 8$ with length $len(heap) = m$ and a global variable ptr with type $bits(ptr) = \lceil log_2(m) \rceil$ and $len(ptr) = 1$, which points to the next free space in the heap array. The function $malloc(int)$ can be implemented as shown in Listing 1 with 1024 heap elements respectively, which needs a 10 bit variable ptr for accessing. A memory exceptions occurs (label *memout*), if there is not enough memory left.

```
1 int heap(8)[1024];
2 int ptr(10);
3
4 int(10) malloc(int n(10)) {
5   if (ptr >= 1024-n) goto memout;
6   ptr = ptr+n;
7   return ptr-n; }
```

Listing 1: Malloc as SPDS in ISO-C like syntax

Once reserved space can be reused, because a garbage collector and a function $free$ can be implemented in SPDS. A pointer is a SPDS variable used as an index of the heap array and an address is just another index (returned by the address operator &). Variables placed in the heap array support the address operator in contrast to the other SPDS variables. If putting a **local** variable of a function f into the heap

array, the recursion of f will be bound, because of a finite maximal heap size. Coverage metrics have to adapt to these additional SPDS functions, statements and variables to be able to compute the correct gap.
Call by Reference: Instead of passing a variable as a function parameter, a pointer can be used to indirectly access variable values in the heap. Thus call by reference can be simulated. Unfortunately this results in bounding the recursion, too.
Dynamic Arrays: Array semantic in ISO-C is defined by pointers and access to its elements is defined by pointer arithmetic. Thus $malloc(int)$ can be used for this purpose.

Other constructs and statements from other languages (e.g. classes, structs, objects, dynamic parameter lists, etc.) can be mapped in a similar way. If an arithmetic exception occurs, the SPDS ends and the corresponding ISO-C program P can have undefined behaviour according to the language specification. P can terminate, which is a complying behaviour. Therefore this behaviour is used for our modeling process. Other implementation defined behaviour can be modeled similarly.

3.2 Extraction of Exact Variable Ranges (step 3)

In step 2 a SPDS S' is created for the SPDS S by slightly modifying S (explained in the next section). For a PDS B an automaton $Post^*(B)$ can be computed, which accepts all reachable configurations of B[18]. Thus for the SPDS S' a similar automaton $Post^*(S')$ can be created, because S' is just a symbolical PDS. This is a basic step in symbolic model checking using Moped [14]. We use the $Post^*$ algorithm of the model checker Moped for our implementation by mapping our SPDS definitions to the input language Remopla[3] [17]. The set of reachable heads $h(S') := \{(g,(l,c)) \bullet (g,[(l,c)...]) \in Post^*(S')\}$ is finite because of finite variable types. Thus exact variable ranges can be extracted from $h(S')$. Let $v \in vars$ and $l \in labels$, then $range_l^{S'}(v) := \{[\![v]\!]_g^c \bullet (g,(l,c)) \in h(S')\}$ is the exact variable range of v. The notation S' can be omitted, if S' is well-defined by the context. For all values $k \in range_l(v)$ there is a run of S', such that $[\![v]\!] = k$ on label l and vice versa.

$h(S')$ and $range_l(v)$ can be computed symbolically out of $Post^*(S')$ using Ordered Binary Decision Diagrams (OBDD) operations. The computation of $h(S')$ is a straightforward OBDD restriction operation in $Post^*(S')$ and results in a characteristic function $q : \{0,1\}^n \to \{0,1\}$ represented as an OBDD. The input vectors of q are heads $h(S')$ encoded as finite Bit sequences. The computation of $range_l(v)$ uses cofactors. A cofactor of q is $q[x_i = b](x_1,x_2,\ldots,x_n) := q(x_1,x_2,\ldots,x_{i-1},b,x_{i+1}\ldots,x_n)$ [8]. The positive cofactor is $q[x_i = 1]$ and the negative cofactor is $q[x_i = 0]$. A characteristic function $r : \{0,1\}^m \to \{0,1\}$ for $range_l(v)$ can be computed using cofactors:

Lemma 3.1 *Let k be the starting index of the encoding of v on label l in q and let m be the length of the encoding. Then $r(y_1,y_2,\ldots,y_m) = 1$ is valid, iff $q[x_k = y_1][x_{k+1} = y_2]\ldots[x_{k+m-1} = y_m]$ is not always 0 (not the empty OBDD).*

The proof is a consequence of the definitions. The computation of exact variable ranges is more time-consuming than model-checking the reachability in S' [8]. Fortunately $range_l(v)$ can be approximated using static data flow analyses and test suites. This is the case for focusing on efficiency or unbounded recursion depth in combination with unbounded parallelism. For further reference in comparisons, explanations, and proofs see [8].

3.3 SPDS Supplementation (step 2) and Exact Gap Inference (step 4)

Now we show exemplary, how to apply our framework to common code coverage metrics.

[3] e.g. we map integers to nonnegative numbers, as Remopla does not support negative integers

3.3.1 Function Coverage Gap $\delta_f(S)$

We supplement S with a new global variable $v \notin vgbl(S)$ using type $bits(v) = 1$ and $len(v) = 1$ without any assignment or reading usage on v to ensure the existence of at least one global variable in S'. By construction this variable v has a random undefined value $[\![v]\!] \in \{0,1\}$ on each label $l \in label(S')$ resp. on each program point. The exact function coverage gap $\delta_f(S)$ can be concluded from the exact ranges of variables in S' as follows:

Lemma 3.2

$$\delta_f(S) = 1 - \frac{|\{f \in func(S) \bullet range^{S'}_{fst(f)}(v) \neq \emptyset\}|}{|func(S)|} \tag{5}$$

Proof (sketch): The new global variable v does not influence the model behaviour. All variable evaluations and reachable labels in S' are the same in S. Further it is $vgbl(S') = vgbl(S) \cup \{v\}$, $func(S') = func(S)$ and $func(S) \neq \emptyset$ because of $main \in func(S)$. The main observation is, that a label $l \in labels(S)$ is unreachable or dead, iff $range^{S'}_l(v) = \emptyset$. Thus a function f can be called, iff $range^{S'}_{fst(f)}(v) \neq \emptyset$. Choose a test suite $t' \in T_S$ such that $|func(t')|$ is maximal. With the maximality of t' we have

$$|func(t')| \geq |\{f \in func(S) \bullet range^{S'}_{fst(f)}(v) \neq \emptyset\}|. \tag{6}$$

Every function $f \in func(t')$ has a cover witness test $\alpha \in t'$, so that the label $fst(f)$ is reachable under test α. Thus it is $range^{S'}_{fst(f)}(v) \neq \emptyset$. On the other hand we obtain

$$|func(t')| \leq |\{f \in func(S) \bullet range^{S'}_{fst(f)}(v) \neq \emptyset\}|, \tag{7}$$

because each $f \in func(S)$ with $range^{S'}_{fst(f)}(v) \neq \emptyset$ has at least one test α' (not necessarily $\in t'$) to cover the function f, which can be detected by an evaluation of v. Accordingly it is

$$\sup_{t \in T_S} |func(t)| = |\{f \in func(S) \bullet range^{S'}_{fst(f)}(v) \neq \emptyset\}|, \tag{8}$$

which is equivalent to

$$\inf_{t \in T_S} (1 - \gamma_f(t)) = 1 - \frac{|\{f \in func(S) \bullet range^{S'}_{fst(f)}(v) \neq \emptyset\}|}{|func(S)|}. \tag{9}$$

□

The exact branch coverage gap $\delta_b(S)$ can be computed similarly. Instead of function entry points, just the block entry points are considered.

3.3.2 Statement Coverage Gap $\delta_s(S)$

The SPDS S' will be supplemented with a new variable $v \notin vgbl(S)$ and the type $bits(v) = 1$ and $len(v) = 1$ similarly to the function coverage gap. The exact statement coverage gap $\delta_s(S)$ can be computed:

Lemma 3.3

$$\delta_s(S) = 1 - \frac{|\{l : s \in stats(S) \bullet range^{S'}_l(v) \neq \emptyset\}|}{|stats(S)|} \tag{10}$$

Proof (sketch): Similar to Lemma 3.2 prove $\sup_{t \in T_S} |stats(t)| = |\{l : s \in stats(S) \bullet range^{S'}_l(v) \neq \emptyset\}|$.

□

3.3.3 Decision Coverage Gap $\delta_d(S)$

The branch coverage uses the nodes of the control flow graph $BBICFG_S$ and the decision coverage uses the edges. The execution of an edge $(b_1,b_2) \in edges(S)$ in the control flow graph $BBICFG_P$ depends on several conditions such as arithmetic overflow, division-by-zero or boolean expressions for conditional branches. To compute the exact decision coverage gap, we introduce a new global variable $v_{in} \notin vars(S)$ into S' with type $bits(v_{in}) = 1 + \lceil log_2(|blocks(S)|) \rceil$ and $len(v_{in}) = 1$. Each label l belongs to a basic block b_l, which can be identified by a unique number $n_{b_l} \in \mathbb{N}_{\geq 0}$. This number is assigned to the variable v_{in} to detect the past basic block for a statement. The type $bits(v_{in})$ is big enough to store every unique identifier n_{b_l}. Each statement "$l : s$" $\in stats(S)$ is modified[4] to "$l : v_{in} = n_{b_l};\ l' : s$" in S', where $l' \notin labels(S)$ is unique. So it is possible to determine the past basic block on label l using the exact range of the SPDS variable $v_{in} \in vgbl(S')$. The exact decision coverage gap $\delta_d(S)$ can be computed:

Lemma 3.4

$$\delta_d(S) = 1 - \frac{|\{(a,b) \in edges(S) \bullet n_a \in range^{S'}_{fst(b)}(v_{in})\}|}{|edges(S)|} \tag{11}$$

Proof (sketch): Let $a,b \in blocks(S)$ be basic blocks. By construction it is $n_a \in range^{S'}_{fst(b)}(v_{in})$, iff there is an execution path from the end of basic block a directly to the first label $fst(b)$ of basic block b in S. This is equivalent to the existence of a test α, such that $(a,b) \in edges(\alpha)$. Thus we have

$$\exists \alpha \in t' : (a,b) \in edges(\alpha) \Leftrightarrow n_a \in range^{S'}_{fst(b)}(v_{in}) \tag{12}$$

for a chosen $t' \in T_S$, where $|edges(t')|$ is maximal. Hence it is

$$\sup_{t \in T_S} |edges(t)| = |\{(a,b) \in edges(S) \bullet n_a \in range^{S'}_{fst(b)}(v_{in})\}|, \tag{13}$$

which shows (11) similar to Lemma 3.2.

□

3.3.4 Condition Coverage Gap $\delta_c(S)$

For the condition coverage all boolean sub-expressions (conditions $BExpr(S)$) on each label are considered. The theoretical maximal value can be achieved, when every condition $(l,e) \in BExpr(S)$ can be 1 ($true$) and 0 ($false$). For each boolean sub-expression $b \in B = \bigcup_{l \in labels(S)} bExpr(l)$ we introduce new boolean global variables $v_b \notin vars(S)$ with type $bits(v_b) = 1$ and $len(v_b) = 1$ into S'. Let further $bExpr(l) = \{e_1, e_2, \ldots e_n\}$ be the set of all boolean sub-expressions on label l. Each statement "$l : s$" $\in stats(S)$ with $bExpr(l) \neq \emptyset$ will be modified to "$l : v_{e_1} = e_1;\ v_{e_2} = e_2;\ \ldots v_{e_n} = e_n;\ l' : s$" in S', where $l' \notin labels(S)$ is unique. The statement "$l : s$" $\in stats(S)$ will be modified to "$l : skip;\ l' : s$" in S', if $bExpr(l) = \emptyset$. Hence the existence of label $l' \in labels(S')$ is guaranteed. Thus the exact condition coverage gap $\delta_c(S)$ can be computed:

Lemma 3.5

$$\delta_c(S) = 1 - \frac{\sum\limits_{\substack{l \in labels(S) \\ e \in bExpr(l)}} |range^{S'}_{l'}(v_e)|}{2 \cdot |BExpr(S)|} \tag{14}$$

[4] Additionally this can be done using the native synchronous parallelism in SPDS without an extra label: "$l : v_{in} = n_{b_l},\ s$".

Proof (sketch): Choose a $t' \in T_S$ such that $|exval(t', S)|$ is maximal. Then it is

$$((l,e),b) \in exval(t',S) \tag{15}$$
$$\Leftrightarrow \quad \text{expression } e \text{ can be evaluated to } b \in \{0,1\} \text{ on label } l \text{ in } S \tag{16}$$
$$\Leftrightarrow \quad b \in range_{l'}^{S'}(v_e). \tag{17}$$

This proves (14) similar to Lemma 3.2, because of

$$\sup_{t \in T_S} |exval(t,S)| = \sum_{\substack{l \in labels(S) \\ e \in bExpr(l)}} |range_{l'}^{S'}(v_e)|. \tag{18}$$

□

3.4 Conclusion for $\delta_\gamma(P)$ based on $\delta_\gamma(S)$ (step 5)

The computed gap $\delta_\gamma(S)$ is exact ($\delta_\gamma(S) = \delta_\gamma(P)$), if the behaviour of S is equivalent to the behaviour of P. Hence step 1 does not abstract nor simplify the program behaviour. This is the case for our ISO-C compatible semantic [9] on C programs. If S abstracts from the behaviour of P, $\delta_\gamma(S)$ is only an approximation for $\delta_\gamma(P)$. The approximation degree depends on the degree of this abstraction.

4 Gap Approximation using δ_γ^- and δ_γ^+

The gap can be approximated by abstracting the program P to a simpler behavior of SPDS S as shown above. On the other hand the exact variable ranges $range_l(v)$ can be approximated, too. This is a more practical approach particularly for huge software systems. Let $range_l^+(v)$ be an over- and $range_l^-(v)$ an under-approximation of $range_l(v)$. The sets $range_l^-(v)$ can be realized using a test suite $t \in T_P$. All occurring variable values during the tests $\alpha \in t$ can be used as lower bound for $range_l(v)$. On the other hand $range_l^+(v)$ can be realized using a conservative data flow analysis. This usually results in additional variable values, which never can be achieved. Both δ_γ^- and δ_γ^+ can be defined similar to δ_γ using $range_l^-(v)$ and $range_l^+(v)$ instead of $range_l(v)$. It is easy to realize, how to bound the gap δ_γ using $range_l^-(v)$ and $range_l^+(v)$:

Lemma 4.1 $\delta_\gamma^+ \leq \delta_\gamma \leq \delta_\gamma^-$.

Obviously the gap approximation is perfect and an exact gap is found, if $\delta_\gamma^+ = \delta_\gamma^-$. In this case it is not necessary to compute exact variable ranges.

5 Exemplary Illustration

As a comparative measurement of our method the values calculated by gcov [22] [5] are presented at the end of this section. The free tool gcov calculates the code coverage during an execution, which can be used to track the code coverage of a test suite.

To show the concepts presented so far, we use example P_1 in Fig. 1 and example P_2 in Fig 2. The constructs in the presented ISO-C code are automatically mapped to SPDS-statements as described in section 3.1. P_1 contains an arithmetic exception, caused by a division-by-zero. Hence P_1 contains a lot of dead code and any test suite with at least one test would be complete (i.e. there is no way to cover more code). There is no test necessary ($t = \emptyset$), because the variables x and y are initialized on

[5] `http://gnu.org/software/gcov`

labels $l0$ and $l1$. Thus it is $range_l(x) = range_l(x)^- = \{0\}$ and $range_l(y) = range_l(y)^- = \{1\}$ for all $l \in L$, where $L = \{l0, l1, l2, lb0, lb1\}$. It is $range_l(x) = range_l(x)^- = range_l(y) = range_l(y)^- = \emptyset$ for all $l \in labels(P_1) \setminus L$. All the conditions in conditional branches are considered to be statements (see BBICFG in the right part of Fig. 1), because a conditional branch can contain a statement (e.g. `x=0` in "`if (x=0)...`").

Thus it is $|stats(P_1)| = 15$, $|blocks(P_1)| = 12$, $|edges(P_1)| = 15$, $BExpr(P_1) = \{(lc0, x == 0), (lb0, y < x)\}$ and $exval(t, P_1) = \{((lb0, y < x), false)\}$. Additionally P_1 is supplemented with variables $v, v_{in}, v_{x==0}$ and $v_{y<x}$ to program P_1', where the *range* values can be computed accordingly. An inter-procedural conservative interval analysis [20, 21] can detect $range_l(v) = range_l(v_{in}) = \emptyset$ for all $l \in L' = \{lb0', lc2\}$ and $range_l(v) = \{0, 1\}$ for all $l \in labels(P_1') \setminus L'$. This is used to compute δ_f^+, δ_b^+ and δ_s^+. The edges $(b2, b3), (b7, b9), (b9, b2), (b3, b10), (b3, b5) \in edges(P_1)$ are never executed, which is discovered by the interval analysis. This results in $\delta_d^+(P_1) = \frac{5}{15}$. Thus, the coverage metrics and gaps of Table 1 can be calculated for P_1 as described in the previous sections. The values were obtained using our current implementation of the program described in [8]. Computing the values presented in Table 1 takes less than two seconds on a modern Core i7 CPU equipped with 8 GiB RAM. Table 1 also contains the approximated values as presented in section 4. Although the coverage metrics are far less than 100 %, the test suite t is complete. Additional tests can not improve these coverages as confirmed by the gaps.

	$\gamma_f(t)$	δ_f^+	δ_f	δ_f^-	$\gamma_s(t)$	δ_s^+	δ_s	δ_s^-	$\gamma_d(t)$	δ_d^+	δ_d	δ_d^-
P_1	0.5	0.0	0.5	0.5	0.33	0.13	0.67	0.67	0.13	0.33	0.87	0.87
P_2	0.5	0.0	0.0	0.5	0.36	0.0	0.09	0.64	0.20	0.0	0.20	0.80
	$\gamma_b(t)$	δ_b^+	δ_b	δ_b^-	$\gamma_c(t)$	δ_c^+	δ_c	δ_c^-				
P_1	0.25	0.17	0.75	0.75	0.25	0.50	0.75	0.75				
P_2	0.38	0.0	0.13	0.62	0.75	0.0	0.13	0.25				

Table 1: Code Coverages and Gaps for P_1 in Fig. 1 and P_2 in Fig. 2

$label(P_2)$	$range_*^-(x)$	$range_*^-(y)$	$range_*^-(z)$	$range_*^-(w)$
m1,m2,m3,m5',lc0,lc1,lc2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
m0,m4	$\{0,1,10\}$	$\{1,5\}$	$\{0,1,10\}$	$\{1,5\}$
m5,m6	$\{0\}$	$\{1,5\}$	$\{0,1,10\}$	$\{1,5\}$

Table 2: $range_l^-$ in P_2 using test suite t for P_2 in Fig. 2

The test suite t discovers $range_l^-(v) = \{0, 1\}$ for each reachable label l in t.

Most compilers, i.e. GCC and CL[6] from Microsoft, are not able to do a flow-sensitive, context-sensitive inter-procedural analysis needed for a more precise lower bound in this example. The abstract interpretations done in a compiler or analysis tool do not yield such a precise lower bound, as most other tools are essentially model-checkers. Hence, the lower bound on the range for `x` and `y` would include all possible values at label `lb1` in P_1. Thus the lower bound on the function gap would be 0.

Additionally, using the tool gcov to compute the coverage of the test suite of example P_1, no coverage is achieved by any test suite, because gcov does not take arithmetic exceptions into account resulting in 0% coverage. Of more practical relevance is the calculation of the coverage gap for non-arithmetic errors. For instance example P_2 in Fig. 2 has a difficult condition $(x < y \;\&\&\; z > w)$. The variables `x`, `y`, `z` and `v` of P_2 (Fig. 2) are global variables of type $char = bits(..) = 8$. The whole block below $(m1 - m3)$ becomes dead, if the condition on label $m0$ evaluates to $false$. Thus `commit` is not called and

[6] Shipped with Microsoft Visual Studio

```
1  char x,y,z,w;
2  void commit(){
3  lc0: y = x + w;
4  lc1: main();
5  lc2: return; }
6
7  void main(){
8  m0: if(x<y && z>w) {
9  m1:    x = x + 1;
10 m2:    commit();
11 m3:    x = 127; }
12 m4: x=z*(y+x-y)-x*z;
13 m5: if(x==127) y=0;
14 m6: return; }
```

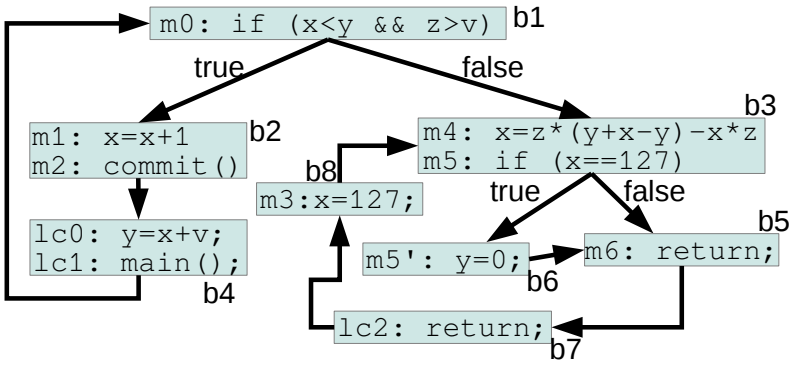


Figure 2: SPDS example P_2 in ISO-C syntax for dead code by bad condition + corresponding BBICFG

the (indirect) recursion not started. Additionally, for all possible test cases, the condition ($x == 127$) on label $m5$ never evaluates to $true$.

Let $t = \{(0,1,0,1),(1,1,1,1),(10,5,10,5)\}$ be a test suite with (x,y,z,w) being the values set before calling `main`. It is $|stats(P_2)| = 11$, $|blocks(P_2)| = 8$, $|edges(P_2)| = 10$, $BExpr(P_2) = \{(m0, x < y), (m0, z > w), (m0, x < y \&\& z > w), (m5, x == 127)\}$ and $exval(t, P_2) = \{((m0, x < y), true), ((m0, x < y), false), ((m0, z > w), true), ((m0, z > w), false), ((m0, x < y \&\& z > w), false), ((m5, x == 127), false)\}$. The test $\alpha = (0,1,1,0)$ would be a good candidate for the test suite t, because $\gamma_s(\{\alpha\}) = 91\%$ is perfect (proofed by the gap δ_s). Table 1 contains the calculated coverage metrics and gaps for P_2. As one can see from the third line of Table 1, the exact gap in the existing code is rather small: it consists of the condition $x == 127$ on label $m5$ and the following code block. This is one of the examples in which our method can instruct the tester to expand the test suite. More code can not be covered, because in each test the variables x and z as well as y and v are aliases. Although most of the code in the example is alive. As seen in the previous example, the approximated lower and upper bounds are not perfect. An upper bound on the gap δ_f of called functions, is $\delta_f^- = 1 - 0.5$, whereas from the two available functions one was called during the execution of test suite t. It is $range_l(v) = \{0,1\}$ for every label $l \in labels(P_2')$ in P_2' (supplementation of P_s). P_2 is also supplemented with variables $v_{in}, v_{x==0}$ and $v_{y<x}$, so that the *range* values can be computed and approximated using an inter-procedural conservative interval analysis ($range^+$). The results for the variables are shown in Table 2 and 1.

Contrary to gcov the computation of the code coverages followed the C Program and did not rely on any symbolic assembler. Such abstractions might cause more coverage shown than the actual coverage in ISO-C. The statement coverage reported by gcov corresponds to γ_s. Values close or exactly corresponding to γ_X can be obtained from gcov for these particular examples. Not all values will match γ_X, because gcov uses a different definition for decision and branch coverage and relies on symbolic assembler output.

6 Related Work

A lot of research is done to get a better coverage for a test suite [4]. However an important point is often missing: often it is impossible to cover 100% of the code in practice, because of gaps.

To the best of our knowledge no research has been done to compute provable exact gaps used in code coverage. Conservative strategies underestimate the coverage gap [5]. Current research only approximates the gap. For instance [3] presents a method to automatically add tests by computing a gap of code covered by the test suite and possible code coverage. The authors miss the important point of having code which will and can never be used. In [3] the emphasis is on large scale projects, but especially in such large projects there is code which cannot be executed and should be removed by the compiler. As [11] describes, some code parts are more important than others. Testing parts of a program which will never be executed is then a loss of resources. Gittens et al. use a domain expert to categorize the code, i.e. for which parts of the source code their tool should generate tests automatically. Our gap computation presented in this paper could be used to automatically categorize the code and not depend on a domain expert. Another project of interest is [16] by Kicillof et al., which shows how to create checkable models. The focus of Kicillof et al. are models which can be created by stakeholders or maybe even marketing experts, and thus is directed at their specific problems at Microsoft. Most other research concerning the computation of gaps in coverage targets the pre-silicon design validation, i.e. [6, 19].

Both papers on pre-silicon design validation are not concerned with testing gaps. They rather check if a specification can be achieved. However, our paper is concerned with languages similar to C, not any Register Transfer Language (RTL) or even specifications.

As Regehr correctly writes in [15], such specifications, which are checked in [6, 19] might have been wrong in the first place. One solution proposed by Regehr for finding errors in specifications is having more people to look over these. A different solution uses our tool and computes parts of the realized specification that are never used, thus giving hints to erroneous specifications.

Whereas Berner et al. are targeting the user of an automatic test system [4] our method targets the automatic test system itself. Berner et al. describe *lessons learned* from their experience with code coverage analysis tools and automatic test generation tools and propose a list of rules to be followed when introducing and using an automatic test tool. Our research was not concerned with usability and group dynamics in a programming environment.

To the best of our knowledge the current research in testing, be it concolic[7] or model-based, is not concerned with the actual problems of code coverage gaps. Gap coverage analysis is not only useful in test case generation but also in the verification of functional correctness. Imagine the case of a dead function granting more user rights, it is easy to use a buffer overflow to trigger this functionality. Similar methods have been used by the CCC for analyzing and using a trojan horse [10][8].

Another important tool, which might be able to compete with our method is Frama-C [5][9]. Frama-C is a conservative analysis tool, which is able to find dead code, execute a static value analysis and, contrary to gcov, is able to detect runtime-errors triggered for instance by division-by-zero. One of the differences between Frama-C and the method we propose in this paper is the theory behind it. In contrast to Frama-C [5] our method uses exact computation, does not overapproximate the values and does not rely on an experienced user. Our exact value analysis produces neither false negatives nor false positives as in Frama-C. Although their value analysis sometimes detects that a function does not terminate, it cannot be used to prove that a function terminates in general.

Frama-C provides sophisticated plugins, but not all of them handle recursion properly. No sophisticated examples can be handled by Frama-C's value analysis. Some of the examples tested even cause runtime-errors in Frama-C itself, thus it is not reliable[10].

[7] interwoven concrete and symbolic execution

[8] especially the section *Upload- und Execute-Mechanismus*

[9] `http://frama-c.com`

[10] It should be noted, that these runtime-errors should vanish in future versions

As our review of the research indicates, none of current research done in testing is concerned with *exact* gap computation.

7 Summary and Conclusions

This paper presents a framework to compute exact gaps between the feasible and theoretical maximal possible code coverage value. For specifying programs in an ISO-C semantic we use a very powerful model, namely SPDS. The power of SPDS allows to model an ISO-C compatible semantics for programs without abstraction. Therefore we are able to do an exact value analysis using model checking techniques and so we obtain exact gaps. We describe how to efficiently approximate the gap in all the other cases. When using flow-sensitive, path-sensitive, inter-procedural and context-sensitive data flow analyses for approximating the exact values one can also use a model-checking tool. The biggest problems of using a model-checker are false positives or false negatives caused by abstraction. Instead, our approach does not rely on such heavy abstraction and does not cause false alarms on our ISO-C compatible semantic. Thus user input or feedback is not required to decide about false alarms. A lot of computing power is required for using such powerful models. Due to smaller programs and smaller data types our approach is still practical for embedded systems.

Having combined the best parts of model-checking and static analyses we use expansive model-checking only when needed (e.g. the gap approximation bounds are not small enough). Thus the computation of $Post^*$ is needed only if the gap approximation using static analysis and a test suite is not exact ($\delta^- \neq \delta^+$).

Using our method a lot of metrics can be compared better among each other now, because of exactly specified gaps. Our method allows the testing of non-functional requirements, too. For example the worst case execution times (WCET) using a WCET metric[11] can be computed.

Our current research considers the practical relevance of exact gap computation for verification of software especially in the area of compiler correctness. Additionally we are considering other areas of research to apply the computation of exact values and exact gaps. For example the computation of exact value ranges can be used for verification of components [2].

References

[1] Frances E. Allen (1970): *Control flow analysis*. *SIGPLAN Not.* 5, pp. 1–19, doi:10.1145/390013.808479.

[2] Andreas Both, Dirk Richter (2010): *Automatic Component Protocol Generation and Verification of Components*. In: *36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 94–101, doi:10.1109/SEAA.2010.30.

[3] Mauro Baluda, Pietro Braione, Giovanni Denaro & Mauro Pezzè (2010): *Structural coverage of feasible code*. In: *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, ACM, New York, NY, USA, pp. 59–66, doi:10.1145/1808266.1808275.

[4] Stefan Berner, Roland Weber & Rudolf K. Keller (2007): *Enhancing Software Testing by Judicious Use of Code Coverage Information*. In: *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, IEEE Computer Society, Washington, DC, USA, pp. 612–620, doi:10.1109/ICSE.2007.34.

[5] Pascal Cuoq & Virgile Prevosto: *Frama-C's value analysis plug-in*. CEA LIST, Software Reliability Laboratory, Saclay, F-91191.

[11] e.g. $\gamma_{WCET}(t) := \max_{a \in t} runtime(a)$ with supplemetation $tick = tick + 1$ on each statement, such that $\gamma_{WCET}(t) = \max(\cup_{l \in labels(S)} range_l(tick))$.

[6] A. Das, P. Basu, A. Banerjee, P. Dasgupta, P. P. Chakrabarti, C. Rama Mohan, L. Fix & R. Armoni (2004): *Formal verification coverage: computing the coverage gap between temporal specifications*. In: *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, ICCAD '04, IEEE Computer Society, Washington, DC, USA, pp. 198–203, doi:10.1109/ICCAD.2004.1382571.

[7] Dejvuth Suwimonteerabuth, Stefan Schwoon, Javier Esparza (2005): *jMoped: A Java Bytecode Checker Based on Moped*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science (LNCS)* 3440, Springer-Verlag Berlin Heidelberg, pp. 541–545. `http://www.springerlink.com/content/32p4x035k3rll5nh/`.

[8] Dirk Richter (2009): *Rekursionspraezise Intervallanalysen*. In: *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, Maria Taferl. `http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1726`.

[9] Dirk Richter, Raimund Kirner, Wolf Zimmermann (2009): *On Undecidability Results of Real Programming Languages*. In: *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, Maria Taferl. `http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1726`.

[10] Chaos Computer Club e.V.: *Analyse einer Regierungs-Malware*. Available at `http://www.ccc.de/system/uploads/76/original/staatstrojaner-report23.pdf`.

[11] Mechelle Gittens, Keri Romanufa, David Godwin & Jason Racicot (2006): *All code coverage is not created equal: a case study in prioritized code coverage*. In: *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, CASCON '06, ACM, New York, NY, USA, doi:10.1145/1188966.1188981.

[12] Glenford J. Myers (2011): *The Art of Software Testing*. 3nd edition, John Wiley and Sons, ISBN 1118031962.

[13] Ira D. Baxter (2001): *Branch Coverage For Arbitrary Languages Made Easy: Transformation Systems to the Rescue*. In: *IW APA TV2/IC SE2001*. `http://techwell.com/sites/default/files/articles/XUS1173972file1_0.pdf`.

[14] Javier Esparza, Stefan Schwoon (2001): *A BDD-based model checker for recursive programs*. Lecture Notes in Computer Science, 2102:324–336, Springer-Verlag Berlin Heidelberg.

[15] John Regehr: *Who Verifies the Verifiers?* `http://blog.regehr.org/archives/370`. Personal Blog entry of Prof. John Regehr, Computer Science Department, University of Utah, USA.

[16] Nicolas Kicillof, Wolfgang Grieskamp, Nikolai Tillmann & Victor Braberman (2007): *Achieving both model and code coverage with automated gray-box testing*. In: *Proceedings of the 3rd international workshop on Advances in model-based testing*, A-MOST '07, ACM, New York, NY, USA, pp. 1–11, doi:10.1145/1291535.1291536.

[17] S. Kiefer, S. Schwoon, D. Suwimonteerabuth (2006): *Introduction to Remopla*. Institute of Formal Methods in Computer Science, University of Stuttgart.

[18] S. Schwoon (2002): *Model-Checking Pushdown Systems*. Dissertation, Technical University of Munich. `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/schwoon.html`.

[19] Arnab Sinha, Pallab Dasgupta, Bhaskar Pal, Sayantan Das, Prasenjit Basu & P. P. Chakrabarti (2009): *Design intent coverage revisited*. *ACM Trans. Des. Autom. Electron. Syst.* 14, pp. 9:1–9:32, doi:10.1145/1455229.1455238.

[20] Steven S. Muchnick (1997): *Advanced compiler design and implementation*. San Francisco, Calif.: Morgan Kaufmann Publishers.

[21] Zhendong Su & David Wagner (2005): *A class of polynomially solvable range constraints for interval analysis without widenings*. *Theoretical Computer Science* 345(1), pp. 122 – 138, doi:10.1016/j.tcs.2005.07.035. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004).

[22] William Von Hagen (2008): *The Definitive Guide to GCC*. APress, ISBN 1590595858.

Using Built-In Domain-Specific Modeling Support to Guide Model-Based Test Generation

Teemu Kanstrén
VTT
Oulu, Finland
teemu.kanstren@vtt.fi

Olli-Pekka Puolitaival
F-Secure
Oulu, Finland
olli-pekka.puolitaival@fsecure.com

We present a model-based testing approach to support automated test generation with domain-specific concepts. This includes a language expert who is an expert at building test models and domain experts who are experts in the domain of the system under test. First, we provide a framework to support the language expert in building test models using a full (Java) programming language with the help of simple but powerful modeling elements of the framework. Second, based on the model built with this framework, the toolset automatically forms a domain-specific modeling language that can be used to further constrain and guide test generation from these models by a domain expert. This makes it possible to generate a large set of test cases covering the full model, chosen (constrained) parts of the model, or manually define specific test cases on top of the model while using concepts familiar to the domain experts.

1 Introduction

Model-based testing (MBT) as an advanced test automation concept has been gaining increasing interest in the industry in recent years. MBT can be defined in various ways, and in this paper we follow the definition by Utting and Legeard [14] as "Generation of test cases with oracles from a behavioral model". MBT can be a powerful approach in generating test cases to cover various aspects of the system under test from the test models. However, typically several different types of expertise are required to build useful test models and to effectively generate test cases from these models.

Firstly, creating models for MBT is essentially a programming activity, describing the system under test (SUT) at a high level and from the testing viewpoint. Although graphical modeling notations exist, such as using UML state charts, at some level the user always needs to describe the behavioral aspects in terms of some form of programming language constructs [14]. This aspect is highlighted by the use of the term model program to describe the test models [3].

Thus we can say that programming skills are required to produce the test models. Additionally, each MBT tool has its own notation for describing the test aspects (e.g., defining what constitutes a test step) on top of this test model [15], requiring the user to also have expertise in the specific modeling notations of the used tools. However, having programming skills and knowing the tool notations is typically not enough but one also requires domain expertise in order to be able to build useful test models. Typically all the required expertise can only be found in combination of several experts.

In this paper we view MBT from the viewpoint of domain-specific modeling (DSM) and how we can integrate support for more effective modeling and test generation for MBT tools with the help of DSM concepts. In DSM terminology, the person with the programming and tool expertise is termed here as the language expert and the person with domain knowledge as the domain expert.

We discuss the use of a MBT tool that transforms the test model into a DSM language, and how this can be used to more effectively guide test generation. We show how the language developer can use our

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 58–72, doi:10.4204/EPTCS.80.5

OSMOTester MBT tool framework to develop test models using a full (Java) programming language and all the benefits it provides (reuse of skills, test libraries, IDE integration with debugging, refactoring and more). It is our experience that this type of a modeling language provides a powerful modeling basis and is as easier if not easier to learn than custom semi-graphical notations of many MBT tools. The OSMOTester toolset provides a framework for creating these test models, and from these test models automatically creates a DSM language that the domain expert can use to guide test generation.

The rest of this paper is structured as follows. Section 2 presents the background concepts of model-based testing and domain-specific modeling in more detail, and also discusses related work. Section 3 presents the OSMOTester toolset and the approach it supports in detail. Section 4 provides discussion on the topic. Finally, section 5 concludes the paper.

2 Background

This section briefly presents the relevant background concepts for this paper and discusses related work.

2.1 Model-Based Testing

As described before, model-based testing as discussed here is about using behavioral models as a basis to generate tests for a system. This requires various tools and expertise to provide a useful result. Several different MBT tools exist, each with their own set of features and test generation algorithms [15]. These also use various notations that are suited for generic representation of system behavior, such as state-based, transition-based, and function-based notations, and various combinations of such notations [15]. As noted, practically all of them require representing the system under test in terms of some programming language constructs to enable test generation with input data and oracles from the model.

In this regard, we borrow the term model program from [3] to describe the test models. Grieskamp et al. [3] define the model program as using guarded-update rules to modify the global data state. When a rule is invoked, a transition between the data states takes place and at the same time a method on the SUT test adapter is invoked. As a result, test steps (sometimes called actions) take place in an order defined by the model, the model traversal algorithms, and as executed by the test adapter.

With regards to the required expertise, it can be said that several different forms of expertise are required to produce useful models. Domain knowledge is required to produce test cases that link with the SUT, produce meaningful input data and evaluate the results in a useful way. Modeling expertise is required to understand how different aspects of the SUT behavior can be effectively described in the test models. Tool knowledge is required to understand the notation of the MBT tools used and to build models that effectively make use of the provided features of the tools modeling language and allow the tool to effectively generate test cases from these models. Test expertise in general is required to understand not only the vertical target domain but also the horizontal domain of test automation and how they should map together to produce a useful overall test environment.

In a practical context, with complexity of modern systems, it is unrealistic to expect one or few persons to have all this knowledge of different domains. The target domain expert is usually an expert on the application domain and should not be expected to know all the details of MBT to produce useful test cases. A test expert in the target domain may be an expert in applying test automation concepts in the specific domain with the help of the domain expert. Another expert may be an expert in using specific MBT tools and in expressing system behavior in terms of behavioral models.

Different combinations of experts typically exist for different target domains and systems, some

cross-domain experts in several areas, and several experts may be available in any domain. However, effectively working together is required by the different experts to produce useful test models that in the end allow for effective test generation for the target system. With sufficient resources, the overall MBT process can be carried out using only MBT tools and models, for example, using purely model programs as test models, with collaboration of the different (language and domain) experts at all points of the process. However, it is also useful to provide support for the domain experts to perform independent exploration of the test target with the help of the test models. To support this, we look at the concepts of domain-specific modeling.

2.2 Domain-Specific Modeling

Domain-specific modeling can be defined as using models to raise the level of abstraction, using domain concepts to describe the solution [8]. From these models, it is possible to automatically generate the required product (code) as both the modeling language and the generators are created specifically for the company and the domain (application). The DSM approach consists of two main parts:

- Language and generator development, and
- Solution modeling.

In the first part, the modeling notation and the transformation from that notation to the actual end product are defined. The first part involves the domain expert and the language developer. The second part involves the developers who work on developing the actual end products. In case of DSM, this also includes domain experts as they can use the higher abstraction level of the domain specific language to develop products as well.

DSM works best and is most cost-effective when developing several end products which share characteristics but also vary in different ways. In this case the same modeling language can be used to create several products, justifying the cost of language development. In traditional domains the target systems have typically been product families, in which different products can be modeled using the same domain specific language. In the test automation domain, we observe that even a single system typically has numerous varying test cases, based on the same base language.

In our previous work we have used DSM to express test cases directly [11] and also to express test models in MBT with a transformation to a specific chosen MBT tool [6]. This means the domain expert is able to build test cases and test models using familiar domain concepts. However, these approaches require a language expert to define the language in a DSM tool which has no built-in support for the MBT tool notation, making the language development challenging without advanced editing support, requiring specific DSM tool skills, making development harder and complicating maintenance.

In this paper we describe a modeling approach for MBT where the DSM language creation is integrated into the model built using the MBT tool with only little or no extra effort required from the language developer. As the model is built on an existing popular programming language, it allows for re-use of skills, development environments, and other assets.

2.3 Related works

Model-based testing has been reported as having been successfully applied in several domains. This includes aerospace [1], automotive [2, 10], medical [16], communication protocols [3], and information systems [12]. These studies typically show improvements and benefits in applying the MBT approach in the different domains. However, they also highlight the typical situation where the model is built

by an expert in the MBT tools and their notation, with the help of information provided by the domain experts. Providing means for a domain expert without the need of deep MBT tool expertise is a topic where domain-specific modeling can be applied.

In MBT, the test model is typically created to describe a large set of potential SUT behavior. The number of potential test combinations that can be generated from such models is huge or even unbounded. To address this issue, Grieskamp et al. [3] have applied model slicing, where the expert defines constraints in the notation of the model program on how the test generator will generate test cases from the model. They use the term scenario-based test generation viewing the slices to define certain test generation scenarios. Our approach is similar while also providing a higher-abstraction level DSM language for the domain expert. Additionally, we also provide means to use this to manually create specific test scripts from the test models.

Specialized approaches for using MBT in specific domains have also been proposed. Takala et al. [13] have built a tool for model-based testing of Android smart-phone applications. This tool is intended to build test models for Android applications in terms of their common user-interface elements. This provides a specific DSM for MBT in for the Android application domain. We provide a generic approach that can be used as a basis for more specific approaches such as these.

Another example of applying DSM with MBT is presented by Kloos and Eschbach for the domain of railway control systems [9]. A specific language is presented for this domain and this can be used to create test models. This language requires detailed knowledge on domain aspects and formalisms such as Mealy machines and process calculi, which can be challenging for domain experts. We use a common programming language (Java) as the underlying notation to ease the language development, and provide a domain-specific abstraction on top of it to enable the domain expert to work with the model as well.

A related approach is presented by Katara and Kervinen [7] who use low-level keywords (e.g., press key X) and on top of those, higher-level domain-specific action words (e.g., take picture), and transitions between them, to describe the test model for MBT. Our approach is similar in using textual domain-specific language in the context of MBT. However, we automatically produce these domain languages from the test model and allow their use to guide test generation and effectively generate specialized versions of the model.

Finally, our previous work on creating graphical modelling DSM languages for MBT is also relevant [6]. In the previous work specific modeling languages are built for specific domains and complete model programs are generated, which are challenging to create, maintain and evolve in a graphical code generator not built for the MBT tools notation. In this paper, we present an approach where the DSM is automatically built as part of the provided test model. An interesting extension would be to add a graphical DSM layer on top of our automatically generated textual DSM language.

3 The OSMOTester Approach

Our approach is implemented and available in a tool called OSMOTester [4]. Here we refer to our approach as the OSMOTester approach according to this implementation. This approach is based on two layers in line with the definition of domain specific modeling we presented in section 2: one for language and generator (model program) development, and another for the solution development (DSM guided test generation). In this section we first present a high-level overview of the approach, followed by the support for developing the language and the test generator, and finally the support for the solution development. This approach works both for online (direct test execution in generation) and offline (generate scripts and execute later) MBT approaches.

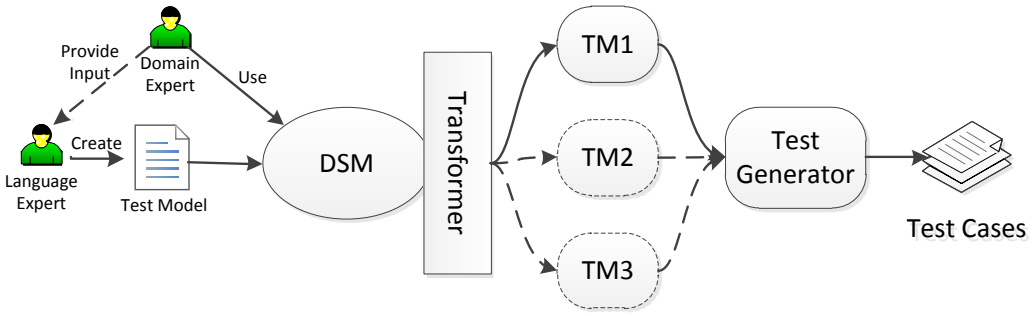


Figure 1: DSM elements in our approach

3.1 High-Level Overview

From the test automation perspective, our approach in terms of DSM is illustrated in Figure 1. Together with the domain expert, the language expert defines the generic overall test model. From this model, OSMOTester automatically forms a higher-level DSM language that the domain expert can use to guide the test generation from this model. In DSM terminology, a transformation is applied by OSMOTester based on the constraints defined by the domain expert to produce constrained variants of the test model (TM1, TM2, TM3 in the figure). The OSMOTester test generator then generates test cases from these test model variants (or scenarios/slices as in [3]).

3.2 Language and Generator Development

The modeling notation supported by OSMOTester is the standard Java programming language with specific annotations and modeling objects to support test generation from the models. With the term model object we refer to Java classes containing transition methods and data-flow variable usage. The basics of the modeling notation and the composition of test models from this notation have been described in our previous work [5]. It is based on our experiences in test modeling and use of other tools (e.g., ModelJUnit [14] is a close relative). We recall here the basics of this modeling notation including the latest evolutions.

We view the models for model-based testing as composed of two main elements: control-flow and data-flow. We start here by presenting the control-flow aspects followed by the data-flow aspects.

3.2.1 Control-Flow Modeling

With control-flow we refer to the order in which the different test steps are executed. The test step refers to executing a transition method on the model program. As is common for this type of approach, the possible transitions that can be taken at any time are defined by their associated guard statements. A transition can only be taken when the associated guard statements allow it to be taken.

To support for control-flow modeling OSMOTester defines two basic annotations, @Guard and @Transition. When a method in a model program is associated with a @Guard annotation, these methods must return a Boolean value of true to allow any associated transitions to be taken by the test generator. A guard method is associated to transitions by their naming, which is given as @Guard(”name”) and

Listing 1: Model object example.

```
1  /** The global model state, shared across test models. */
2  private final ModelState state;
3  /** The scripter for creating/executing the test cases. */
4  private final CalendarScripter scripter;
5
6  @Transition("AddEvent")
7  public void addEvent() {
8    String uid = state.randomUID();
9    Date start = state.randomStartTime();
10   Date end = calculateEndTime(start);
11   ModelEvent event = state.createEvent(uid, start, end);
12   scripter.addEvent(event);
13 }
14
15 @Guard("RemoveOrganizerEvent")
16 public boolean guardRemoveOrganizerEvent() {
17   return state.hasEvents();
18 }
19
20 @Transition("RemoveOrganizerEvent")
21 public void removeOrganizerEvent() {
22   ModelEvent event = state.getAndRemoveOrganizerEvent();
23   scripter.removeEvent(event.getUid(), event);
24 }
```

@Transition("name"). A guard can be associated to several transitions by using an array of names, or by leaving the name out completely which associates it to all transitions in the test model(s).

A transition is the central concept of the control-flow modeling as all other elements related to control-flow are executed together with associated transitions. As it is out experience that transitions in test generation represent a form of test step in a test model, we also support using the annotation @TestStep instead of @Transition. Other control-flow related elements include the @Pre and @Post annotations, which cause any tagged methods to be executed before (@Pre) or after (@Post) the associated transitions. Association is similar as for @Guard annotations. For example, we have used @Post annotations to provide generic test oracles that should be executed after each test step (to evaluate model state vs SUT), and both @Pre and @Post annotations to provide logging information on the model state before and after test steps.

As described in [5], these elements can be embedded in different Java classes and then composed in a modular fashion by the OSMOTester tool. This allows one to specify different concepts, partial models, and their compositions in separate modules and configurations. Listing 1 shows an example of a model object using these elements.

3.3 Data-Flow Modeling

As an extension to our previous work, we also provide a set of objects to support data-flow modeling. The main modeling objects for data-flow currently are:

- Value range: defining a numerical range of values for a variable

Listing 2: Data-flow example

```
@Variable
/** Used to generate start times between January 2000 and December 2010. */
private ValueRange<Long> startTime;

public ModelState() {
  Calendar start = Calendar.getInstance();
  start.setTime(new Date(0));
  start.set(2000, 0, 1, 0, 0, 0);
  Calendar end = Calendar.getInstance();
  end.setTime(new Date(0));
  end.set(2010, 11, 31, 23, 59, 59);
  long startMillis = start.getTimeInMillis();
  long endMillis = end.getTimeInMillis();
  startTime = new ValueRange<Long>(startMillis, endMillis);
}

public Date randomStartTime() {
  return new Date(startTime.next());
}
```

- Value set: defining a set of values for a variable
- Readable words: defining character combinations that consist of commonly displayed human-readable characters.

In order to use these data-flow objects, one initializes them at model-creation time and uses them to provide matching values where needed as input in the test steps. As each of the provided objects keeps a history of the values it has provided, it is possible to configure them with different algorithms for input generation. This includes random values, least covered values, or boundary values (for a value range). Each data-flow object can be configured separately with their own configuration of values and algorithms.

Listing 2 gives an example of defining a range of possible values for a start date of a calendar event using a value range object. New values for this variable are generated using the next() method, which is part of an interface shared by all data-flow objects.

Beyond these specific object types, also primitive values (e.g., integers, Strings, Booleans) and custom objects of any type can be recorded by OSMOTester. While their generation and updates may not be handled by OSMOTester itself, it will still record values of all @Variable tagged variables in the model objects regardless of their type. All values are recorded by their object instance, and can be compared to their String representation if used as definitions in coverage algorithms or similar components.

3.4 Composing the Models and Generating Tests

Once the different elements of the control-flow and data-flow models have been specified, they still need to be composed together and we need to be able to invoke a test generator to generate tests based on these models. OSMOTester provides the test generator component that basically traverses the given model program steps according to their guards and chosen generation algorithms. As the generator traverses

the transitions, it also explores the data-flow space by generating data from any encountered data-flow object.

The binding of the model objects together in practice consists of creating the test generator (Java) object and using its methods to add the (Java) model objects to it. For space reasons, we do not show this as an example here but one is available in [5]. The OSMOTester generator starts by parsing all the given model objects and associating all model elements to each other. It also stores references to all @Variable tagged variables to capture any values they produce. This allows for creation of more advanced algorithms and feature to support test generation without requiring specific action by the user.

It is possible to define a set of specific algorithms and constraints to guide the test generation also at this level. Test and suite end conditions provided with OSMOTester include:

- Length: end after generating a given number of steps or tests
- Probability: end after any step or test with a given probability
- Requirements coverage: end when the given requirements have been covered (identified by specific object calls in the test models)
- Step coverage: end when the defined set of transitions has been covered
- Data coverage: end when the defined set of values for given data variables has been covered
- And/Or compositions: allow composing several end conditions together with logical operators

Similarly, different algorithms for traversing the given models can be defined based on the different elements of the model objects. OSMOTester includes the following algorithms:

- Random: picks a random transition from the ones available
- Balancing: randomly picks an available transition but favors less covered ones
- Weighted: randomly picks an available transition but gives a higher probability to ones with higher weight.

Weight can be defined by the modeler as in @Transition(”name”, 5), where 5 is the weight.

3.5 Domain Specific Scripting Languages

In the previous subsection we described the first part of our DSM support, the language development framework. In this subsection we describe the second part of DSM in our MBT context: support for solution domain modeling. In our previous work we have created graphical domain-specific languages for manual test creation [11] and for test modeling for model-based test generation [6]. There approaches were based on using a specific domain-specific modeling tool to build the models separately from the test models.

Here the test models themselves are used as a basis for the domain language and the MBT tool as the test generator. The support for using such models has been integrated into the OSMOTester tool itself, requiring no external tools or practically no added effort on top of building the test models themselves as described in the previous subsection.

The basis for this domain language is formed by the transitions defined in the test model as described in section 3.2.1. More specifically, the names given to the transitions in the annotations as in @Transition(”name”) form the vocabulary of the domain specific language. In our experience, as people work to build such models and give names to their model elements, they naturally tend to use names of familiar domain and test concepts. For example, OSMOTester comes with a calendar application example that has the following transitions:

- Add event: Adds a new event to the calendar for an existing user.

Listing 3: DSM script example.

```
setting, value
model factory, osmo.tester.examples.gui.TestModelFactory
algorithm, random

step, times
add event, >=2
add overlapping event, >= 3
add task, == 1
add overlapping task, <=2

variable, coverage
event count, 5
```

- Link event to user: Links an existing event to another user, making the new user a participant in the event organized by the first user.
- Remove organizer event: Removes an event completely by deleting it from the organizer and as a result from all linked participants.
- Remove participant event: Removes a participant from an event.
- Add task: Adds a task for an existing user. A task is like an event with no duration or participants.
- Remove task: Removes a chosen task from associated user.
- Add task overlapping event: Adds a task that overlaps a chosen event in time for the same user.
- Add overlapping event: Adds an event overlapping another event in time for the same user.

These are transitions (test steps) of the calendar example that describe its nominal (expected) behavior. Additionally, the calendar test model also defines a set of transitions for testing the error handling behavior of the calendar. These are the following transitions:

- Remove a task that does not exist: Tries to remove a task which does not exist (invalid data).
- Remove an event that does not exist: Tries to remove an event which does not exist (invalid data).

OSMOTester supports two different types of applications of these elements as a domain-specific testing language. In this subsection we will further demonstrate their use in terms of the calendar example.

3.5.1 Abstract Domain-Specific Scripting

We can define constraints over the model programs to guide the OSMOTester test generator. The model defines the overall possible flows of test generation, while the scripting language allows for guiding it towards specific goals. This support is based on the observed needs in industrial application, where the domain experts wish to guide test generation to explore specific areas of interest at different times.

For example, considering the calendar example, we may wish to generate a set of test cases for having several overlapping events and some tasks, with several events active at a time (i.e., not removed before new ones are added). In this case, we can use the generic model program specified previously, and just define the specific requirements for the test cases in the scripting language. A script supporting this definition is shown in Listing 3.

This shows three different tables illustrating different aspects than can be defined for configuring the test generator. First, we provide a settings table specifying that the model objects for test generation are

provided by a class called osmo.tester.example.gui.TestModelFactory. We also define that we wish to use the random algorithm for test generation.

Second, we define a step coverage requirement table specifying that we want the AddEvent step (transition) to occur in each generated test case a minimum of 2 times. We also define that we want the step AddOverlappingEvent to occur a minimum of 3 times. The AddTask step needs to be present exactly once in each generated test case, and the AddOverlappingTask at most 2 times.

The third and final table in this example specifies that the variable EventCount should have the value of 5 at some point in each generated test case. As the model program uses the EventCount variable to express the total number of both organizer events and participant events, this ensures that their total sum as active events during a test case reaches 5 at some point. At the same time, the previously defined requirements for different types of transitions ensure that different types of events are also generated in each test case (organizer and participant).

A table missing in this example is one that allows the user also to override what data values are generated for specific input values. This table is similar to the variable coverage table, defining variable names and their possible values to be generated. If there is no definition for a variable, the generic model program definitions are used instead.

These examples also illustrate the expertise required to build suitable models than can be used in a diverse way as a basis for test generation. For example, in the calendar example, the current model program does not let us specify how many users we wish to have included in the generated tests. In fact, the example provided with the OSMOTester distribution generates a number of users randomly between 3 and 5. If this becomes important to control, a specific transition for adding transitions could be defined and we could then define exact number of these transitions required. Alternatively, a data variable could be defined that is used by the model program to generate a matching number of users and thus allows the user to control the number through the scripting language.

In addition to manual script writing, OSMOTester also provides a graphical user interface (GUI) to create these scripts. This is illustrated in Figure 2. The useful part of this GUI is that given the set of model objects that compose the model-program it can automatically identify all possible transitions and data variables. Thus it can show the user the options available for building the DSM scripts and also the operators available. An example of these is shown in Figure 2 for the calendar example.

Considering the end result, generating a number of tests using test generation algorithms to execute a model program produces a lot of variation but this may not always be the type of variation the domain expert wants at that time. Many MBT tools use static analysis techniques such as symbolic execution and constraint solving to generate specific test cases traversing the model in desired paths. As we take a more dynamic approach of executing the model as a normal program, we also provide support for optimizing the test set in this way.We support optimizing for transition, transition pair, requirements (specific definitions in model objects), variable (number of values) and variable value (specific values) coverage via a greedy optimizer. The optimizer takes a weight value for each of these coverage criteria and provides an optimized subset of specified size from the given overall set.

In our experience test generation from the models is generally fast, and this provides a practical method for providing an optimized test suite. For example, generating 50000 test cases from the calendar example takes about 11 seconds, and optimizing this to find an optimized set of 50 tests with the algorithm takes about 17 seconds on an Intel CoreI5 2,67Mhz system utilizing a single core, with the Java virtual machine maximum heap size of 880MB (formed by Java default settings, not modified or monitored as memory use did not become an issue). As the test object OSMOTester creates for each generated test case gives the model program a possibility to store the test script with it, the user can then pick the optimized set of scripts for use.

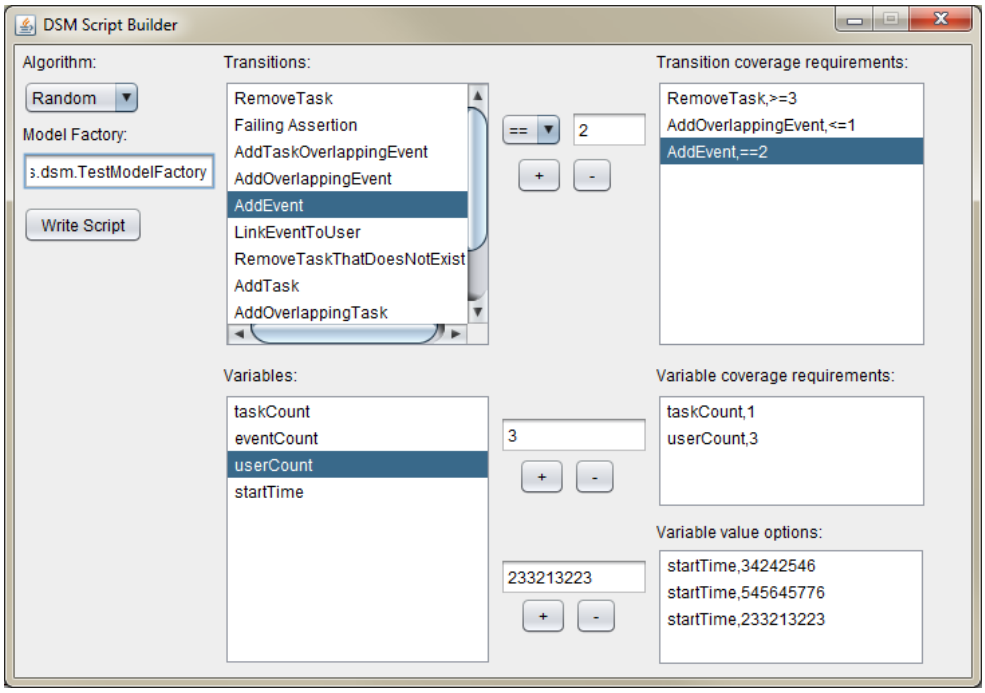


Figure 2: OSMO DSM GUI

3.5.2 Manual Domain-Specific Scripting

Model-based testing is commonly considered as a means for automated test generation from test models. The tests are generated by using some automated algorithm to traverse the test model as illustrated in previous sections. While it is a great goal to strive for creating all test cases automatically from such models, in practice it is our experience that many people still wish to see a set of specific manually crafted test cases that they can control and verify they cover a given set of paths, requirements and other elements of interest. The approach we presented in section 3.5.1 helps address many of these requirements through guiding the test generation through user-provided constraints. However, additionally even more explicit options for manual guidance are needed to fully address this need.

In this section we present how OSMOTester enables the user to manually generate exact test cases from the same test models. Similar to the DSM scripting language, this can also be manually scripted or guided via a GUI shown in Figure 3. This GUI consists of four main elements. The test log in the upper left corner shows the test steps that have been taken. The sequence number is reset when a new test case is started. The metrics set in the upper right corner shows how many times the different transitions have been taken in the current test suite.

The next transition choice in the lower left corner shows which transitions can be taken at a given time. By clicking on a transition in this list, the user can guide the OSMOTester generator to take that transition. Choosing one actually executes that step of the model program and updates the set of available transitions. As the GUI only displays the valid transitions for the current state, the user can only create valid test cases with it.

The lower right corner contains the set of controls used to further guide the test generation. The write script button writes the generated test script to a file. It should be noted that this is not a test script for the

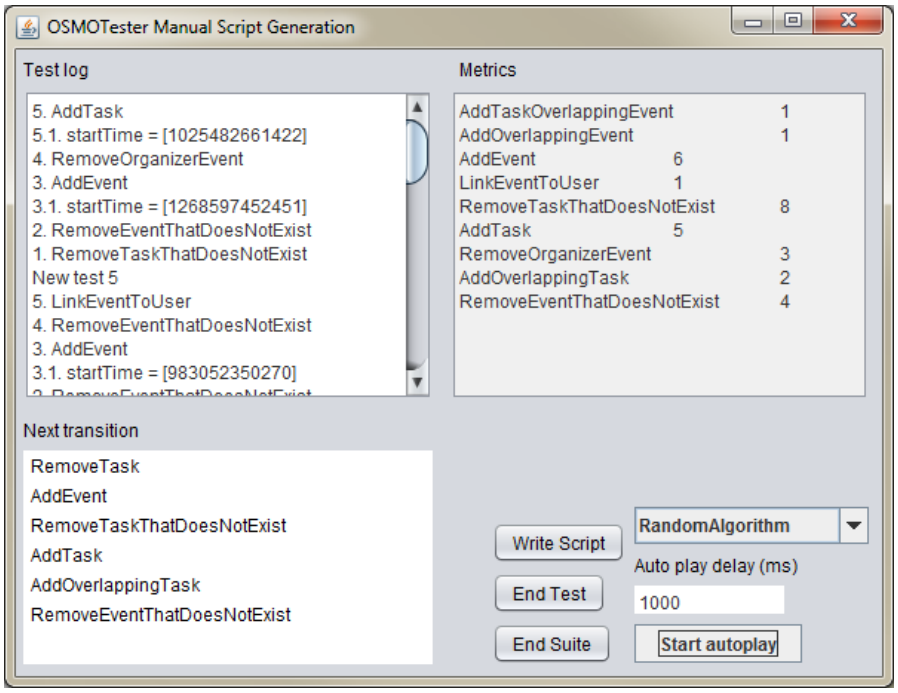


Figure 3: Manual GUI

SUT but a script for OSMOTester itself, defining which transitions to take and which values to provide for data variables. The end test button starts a new test case for the suite and end suite finishes the test generation. The algorithm, delay and start autoplay controls are all related to the autoplay feature. It enables the user to start and stop automated model traversal using a chosen algorithm, and to continue manual generation when preferred, always updating the GUI.

As an end result, OSMOTester will generate a test script matching the set of test cases and their transitions (test steps) as generated in this tool. Listing 4 shows an example portion of this script matching the end of test 4 and start of test 5. Notice that this also includes the values of model variables used in those steps. That is, the user can script both the sequence of transitions and variable values in those steps.

Beyond the transitions, the GUI also allows the user to specify the data values of relevant model variables. Figure4 shows an example of three types of dataflow variables. The (a) option shows a request for the user to provide a value for a numeric range (shown in red if an invalid number is give), (b) shows a choice from a set of values, and (c) shows a request for a word that can be any set of readable characters. The name of the variable is shown in the window title. Option OK inputs the given manual choice for this step in the model program, Skip uses the model program logic to generate the value once, and Auto sets the model program to generate all values for this variable in the future automatically.

This approach has several benefits. First of all, it enables the user to easily create specific test cases from the same model as is used for test generation. Thus test scripting can be done from a single source with less duplicated effort. It also eases test maintenance as updates to the model will also be directly updated to the test cases themselves. Thus changes required to test cases to adapt to changes in the test target can be handled in one place and are automatically updated to all test cases, both manually created and automatically generated.

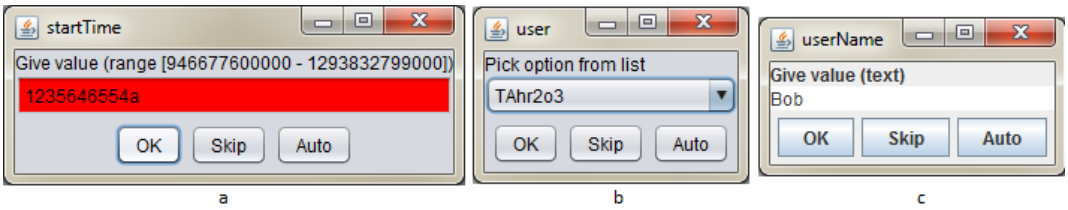


Figure 4: Dataflow GUI (a) Value Range (b) Value Set (c) Readable words

Listing 4: Manual test script example

```
1 action, name, value
2 ...
3 step, link event to user,
4 new test,,
5 step, add event,
6 variable, start time, 1268597452451
7 step, remove event that does not exist,
8 step, remove task that does not exist
```

4 Discussion

As we have presented, our approach is based on using existing programming languages and tool as a basis for modeling. As the approach is only based on the basic programming language constructs, it does not require specific tool support for modeling such as a customized visualization and editing component. Instead, the user can choose any integrated development environment (IDE) they are familiar with and use that as they find best (within the Java programming language constraints). This also makes it much easier for us to build a modeling language and framework as we do not need to worry about features such as refactoring, syntax highlighting, debugging and code navigation. These are practically provided for free by the available tools (IDE).

In our experience, people working with test automation are often also experienced programmers. For them, working with this type of a modeling approach is easy as they can work using tools and notation familiar to them. This also keeps them informed exactly about how their tests are built, instead of hiding if behind the custom notations of tools. In practice our experience is also that the approach gives a lot of power that is not available in customized notations as popular general purpose programming languages typically have more resources put into their development and as well as available libraries.

However, as not all people are experts in programming or wish to write their test case using a programming language notation, higher levels of abstraction are also important. For example, a domain expert may be interested in testing their system using specific configurations of the test models with specific properties. Even if in theory it may be possible to show that a MBT tool will generate a set of impressive test cases, it is equally or even more important to provide confidence for different stakeholders that the important features and their important properties are covered in the provided test cases. By providing means to generate overall test cases from the model, constrain the test generation by using a DSM language, or to manually define specific test cases we enable these different goals to be achieved.

Many existing tools make extensive use of static analysis techniques such as symbolic execution and constraint solving [14]. These enable features such as generating test data to cover specific paths

of the test model. In our case, we have opted not to use such techniques, beyond those provided by the IDE's, but to focus on supporting and effectively exploiting the runtime execution aspects. In this case, any programming language features can be used in the model programs without worrying about the scalability of the analysis techniques. At the same time, as the model program is executed it can also be easily used to provide features such as the manual test generation GUI we have presented here. As the model program is actually executed, the user can at all times be given exact information about the current state of the model and what actions are possible in that state.

While the approach in many ways gives the user power, it also puts a lot of responsibility to them. As the models are not extensively checked by the tool, they may require some more analysis effort by the user (which is not necessarily a bad thing as it increases the understanding of the system). For example, one has to realize that requiring 10 event removals will never terminate if only 5 event creations are allowed. The domain expert needs to have such understanding, but it is also possible to configure end conditions and create models that report such failures, or provide thresholds for breaking and report failure to achieve required constraints, if needed.

For optimization we apply test selection by allowing one to generate a large set and provide optimization algorithms to select a subset with specific properties. This works when generating offline test scripts to be executed after generation. However, if done in online mode where the tests are executed as generated this is not effective but rather manual tuning of the model becomes more important. In these scenarios it could be beneficial also to have more advanced analysis features from the static analysis domain as well, such as the model checking capabilities in Spec Explorer that also uses a form of model programs [3].

OSMOTester is currently being used by several companies in the embedded and software industry to support test automation. It has been our experience in case studies with industrial partners that the user naturally names the elements in the built model using familiar terms in the target domain. It has also been our experience that while a language expert (a role we commonly take) may help them build a model, they can make the most of it if given a simple way to guide the test generation for different aspects in their model. Yet in many cases they are not interested in learning all the details of the model internal elements as domain experts. Examples of real requirements include setting up a model to produce a certain number of users, varying the generated tests with a chosen subset of possible parameters, and using a specific set of values in specific set of generated tests. The approach we have presented here has helped address those needs.

5 Conclusions

Model-based testing is a powerful test generation technique and tools for this have come a long way and are increasingly adopted in the industry. However, their generic nature, required skills and exposed formalisms can hinder their potential in industrial context with domain experts. In this paper, we presented an approach to guide model-based test generation using domain-specific concepts through support integrated into a MBT tool. This approach starts from using a framework over a common programming language (Java) to build the test model program. The naming conventions used in this framework automatically turn this test model into a domain-specific language that can be used to guide the test generation from this model. The user can then either generate test from the generic test model, constrain it to generate varied test cases for specific scenario(s) or manually create specific test cases from the model.

This allows not only automated generation of a large set of test cases from the test model but also addressing more specific needs by domain experts without needing a language (test model) expert to help

with model customization. It also allows one to build more confidence in covering specific elements of the system under test as required by test requirements. As the user can create test cases from the model using manual guidance they can have confidence the requirements are covered and how they are covered. This also helps in the general test maintenance issues as updating the model will also automatically update the manually created test scripts that are based on this model. In the future we plan to explore further means to ease the adoption and use of MBT with concepts such as specification mining input.

References

[1] M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker & R. Kasuda (2002): *Mars Polar Lander Fault Identification Using Model-Based Testing*. In: *8th IEEE Int'l. Conf. on Engineering of Complex Computer Systems (ICECCS02)*, pp. 163–169, doi:10.1109/ICECCS.2002.1181509.

[2] E. Bringmann & A. Krämer (2008): *Model-Based Testing of Automotive Systems*. In: *IEEE Int'l. Conf. on Software Testing, Verification and Validation (ICST2008)*, pp. 485–493 doi:10.1109/ICST.2008.45.

[3] W. Grieskamp, N. Kicillof, K. Stobie & V. Braberman (2011): *Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology*. *Journal of Software Testing, Verification and Reliability* 21(1), pp. 55–71, doi:10.1002/stvr.427.

[4] T. Kanstrén (2011): *OSMOTester*. Available at `http://code.google.com/p/osmo`.

[5] T. Kanstrén, O-P. Puolitaival & J. Perälä (2011): *An Approach to Modularization in Model-Based Testing*. In: *3rd Int'l. Conf. on Advances in System Testing and Validation Lifecycle (VALID2011)*.

[6] T. Kanstrén, O-P. Puolitaival, V-M. Rytky, A. Saarela & J. S. Keränen (2012): *Experiences in Setting up Domain-Specific Model-Based Testing*. In: *13th IEEE Int'l. Conf. on Industrial Technology (ICIT2012)*.

[7] M. Katara & A. Kervinen (2006): *Making Model-Based Testing more Agile: A Use Case Driven Approach*. In: *Haifa Verification Conference (HVC2006)*, pp. 219–234, doi:10.1007/978-3-540-70889-6_17.

[8] S. Kelly & J-P. Tolvanen (2008): *Domain Specific Modeling: Enabling Full Code Generation*. Wiley.

[9] J. Kloos & R. Eschbach (2010): *A Systematic Approach to Construct Compositional Behaviour Models for Network-structured Safety-critical Systems*. *Electronic Notes in Theoretical Computer Science* 263, pp. 145–160, doi:10.1016/j.entcs.2010.05.009.

[10] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch & T. Stauner (2005): *One Evaluation of Model-Based testing and its Automation*. In: *27th Int'l. Conf. on Software Engineering (ICSE2005)*, pp. 392–401, doi:10.1145/1062455.1062529.

[11] O-P. Puolitaival, T. Kanstrén, V-M. Rytky & A. Saarela (2011): *Utilizing Domain-Specific Modelling for Software Testing*. In: *3rd Int'l. Conf. on Advances in System Testing and Validation Lifecycle (VALID2011)*.

[12] P. Santos-Neto, R. Resende & C. Pádua (2008): *An Evaluation of a Model-Based Testing Method for Information Systems*. In: *ACM Symposium on Applied Computing*, pp. 770–776, doi:10.1145/1363686.1363865.

[13] T. Takala, M. Katara & J. Harty (2012): *Experiences of System-Level Model-Based GUI Testing of Android Applications*. In: *4th IEEE Int'l. Conf. on Software Testing, Verification and Validation (ICST2011)*, doi:10.1109/ICST.2011.11.

[14] M. Utting & B. Legeard (2007): *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann.

[15] M. Utting, A. Pretschner & B. Legeard (2011): *A Taxonomy of Model-Based Testing Approaches*. *Journal of Software Testing, Verification and Reliability*, doi:10.1002/stvr.456.

[16] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella & B. Hasling (2008): *Applying Model-Based Testing to Healthcare Products: Preliminary Experiences*. In: *30th Int'l. Conf. on Software Engineering (ICSE2008)*, pp. 669–671, doi:10.1145/1368088.1368183.

Talking quiescence: a rigorous theory that supports parallel composition, action hiding and determinisation

Gerjan Stokkink, Mark Timmer, and Mariëlle Stoelinga
Formal Methods and Tools, Faculty of EEMCS
University of Twente, The Netherlands
{w.g.j.stokkink, timmer, marielle}@cs.utwente.nl

The notion of quiescence — the absence of outputs — is vital in both behavioural modelling and testing theory. Although the need for quiescence was already recognised in the 90s, it has only been treated as a second-class citizen thus far. This paper moves quiescence into the foreground and introduces the notion of quiescent transition systems (QTSs): an extension of regular input-output transition systems (IOTSs) in which quiescence is represented explicitly, via quiescent transitions. Four carefully crafted rules on the use of quiescent transitions ensure that our QTSs naturally capture quiescent behaviour.

We present the building blocks for a comprehensive theory on QTSs supporting parallel composition, action hiding and determinisation. In particular, we prove that these operations preserve all the aforementioned rules. Additionally, we provide a way to transform existing IOTSs into QTSs, allowing even IOTSs as input that already contain some quiescent transitions. As an important application, we show how our QTS framework simplifies the fundamental model-based testing theory formalised around ioco.

1 Introduction

Quiescence is a fundamental concept in modelling system behaviour. It explicitly represents the fact that, in certain system states, no output is provided. The absence of outputs is often essential: an ATM, for instance, should deliver the requested amount of money only once, not twice (see Figure 1). This means that the ATM's state just after paying out money (s_0 in Figure 1) should be quiescent: it should not produce any output until further input is given. On the other hand, the state before paying out (s_3 in Figure 1) should clearly not be quiescent. Hence, quiescence can also sometimes be considered as erroneous behaviour.

Thus, the notion of quiescence is essential in testing: if a system under test (SUT) does not provide any output, then the test evaluation algorithm must decide whether to produce a pass verdict (allowing quiescence at this point) or a fail verdict (forbidding quiescence at this point).

Origins. The notion of quiescence was first introduced by Vaandrager in [14] to obtain a natural extension of the notion of a terminal or blocking state: if a system is input-enabled (i.e., always ready to receive inputs), then no states are blocking, since each state has outgoing input transitions. However, quiescence can still be used to denote the fact that a state would be blocking when considering only the output actions. Quiescence is explored further in [6, 7].

Tretmans introduced the notion of *repetitive quiescence* [11, 12], which emerged from the need to continue testing, even in a quiescent state: in the ATM example above, we need to test further behaviour that arises from the (quiescent) state after providing money. To accommodate these needs, Tretmans

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 73–87, doi:10.4204/EPTCS.80.6

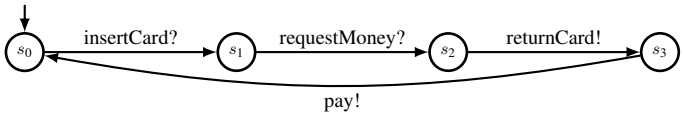


Figure 1: A very basic ATM.

introduced the *suspension automaton* as an auxiliary concept. More recent uses of quiescence include [1], applying it in the context of machine learning.

Example 1.1. Consider the automaton given in Figure 1. The states s_0 and s_1 are quiescent, since they do not have any outgoing output transitions. To obtain the suspension automaton corresponding to such a system, Tretmans adds self-loops, labelled with the quiescence label δ, to each quiescent state. □

Limitations of current treatments. While the papers above all convincingly argued the need for quiescence, none of them presents a comprehensive theory of quiescence. Firstly, quiescence is not treated as a first-class citizen: although the suspension automaton is used during testing, it is not defined as an entity in itself. Therefore, quiescence cannot be used to specify systems, and neither is it clear what properties a suspension automaton satisfies or should satisfy. Since conformance relations such as `ioco` are defined based on 'suspension traces', which are the traces of a suspension automaton, it seems much more appealing to directly start from these suspension automata and base the whole theory on them.

Secondly, basic operators like parallel composition and hiding were only defined for input-output transition systems, but have not been studied for suspension automata at all. Therefore, it was still an open question to what extent these operators could be lifted to the setting of quiescence.

Our approach. The current paper remediates the shortcomings of previous work and presents a comprehensive theory for quiescence, by introducing *quiescent transition systems* (QTSs). These are input-output transition systems in which quiescence can be represented explicitly by δ-transitions, and form a fully-formalised alternative to Tretmans' suspension automata. Whereas suspension automata are always constructed by adding δ-transitions to existing LTSs and subsequently determinising [13], QTSs are defined in a precise manner as a stand-alone entity, can be built from scratch and need not necessarily be deterministic.

As a first step, we handle QTSs that are input-enabled (never reject an input) and most importantly convergent (free of infinite sequences of internal transitions), since the interplay between quiescence and infinite sequences of internal transitions is delicate. Hence, we first focus on the basics. Relaxing these restrictions is an important direction for future work.

Starting point in our theory is the observation that, when treating quiescence as a first-class citizen, restrictions need to be put in place. For instance, it should never be the case that a δ-transition is followed by an output, as this would contradict the meaning of quiescence. As another example, as argued elaborately in Section 3, we do not allow a δ-transition to enable additional behaviour; after all, it would not make much sense if our observation of the absence of outputs impacts the system. In this paper we present and discuss four such rules, that restrict the domain of all possible QTSs to a sensible subclass.

We define three well-known automata-theoretical operations on QTSs: parallel composition, hiding and determinisation. These operations are very important, as they allow a modular approach to system specification. Additionally, we explain how to obtain a QTS from an IOTS by a process called deltafication. We define this process in a liberal way, supporting also the construction of a QTS from an IOTS

that already has some δ-transitions in place. We show that our four requirements on QTSs, which are a key contribution of this paper, are preserved by all of these operations.

This novel theory of QTSs simplifies the theory of model-based testing. Hence, we conclude this paper by showing how QTSs can be used to define the conformance relation `ioco`, and aid in test case generation and evaluation.

Overview of the paper. First, we present some preliminaries on input-output transition systems in Section 2. Then, Section 3 introduces the QTS model and its operations, as well as a variety of important (closure) properties. Section 4 explains how to construct QTSs based on IOTSs, and Section 5 discusses the application of QTSs to test theory. Finally, conclusions and future work are presented in Section 6.

Due to space limitations, we refer to [8] for detailed proofs of all our lemmas, propositions and theorems.

2 Background

2.1 Preliminaries

Given a set L, we denote by L^* the set of all sequences over L. Given a sequence $\sigma = a_1 a_2 \dots a_n$, we define the length of σ, denoted $|\sigma|$, as n. The empty sequence is denoted by ϵ.

Given two sequences $\rho = a_1 a_2 \dots a_n \in L^*$ and $\upsilon = b_1 b_2 \dots b_k \in L^*$, we define the concatenation of ρ and υ, denoted $\rho + \upsilon$ or $\rho\upsilon$, as $a_1 a_2 \dots a_n b_1 b_2 \dots b_k$. The sequence ρ is a *prefix* of υ, denoted $\rho \sqsubseteq \upsilon$, if there is a $\rho' \in L^*$ such that $\rho\rho' = \upsilon$; if $\rho' \neq \epsilon$, then ρ is a *proper prefix* of υ, denoted $\rho \sqsubset \upsilon$.

Given a set $S \subseteq L^*$, a sequence $\sigma \in S$ is called *maximal with respect to* $\sqsubseteq$ if there does not exist a sequence $\rho \in S$ such that $\sigma \sqsubset \rho$. Clearly, such a maximal sequence always exists.

We use $\wp(L)$ to denote the *power set* of L, i.e., $\wp(L)$ is the set of all subsets of L, including the empty set and L itself.

2.2 Input-Output Transition Systems

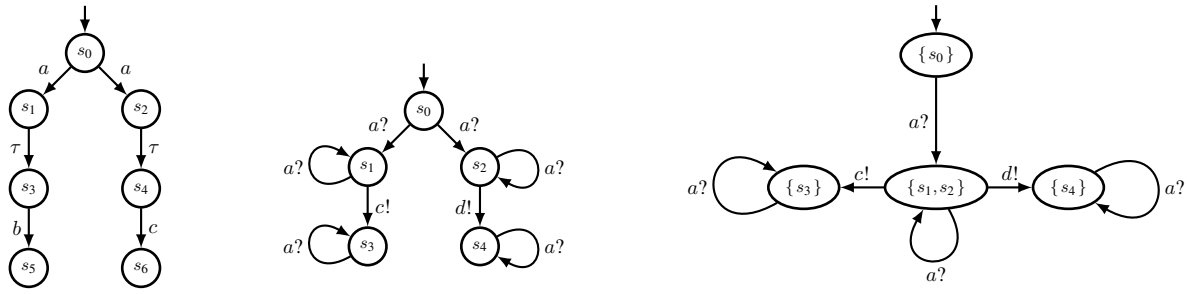
Before we introduce Input-Output Transition Systems, we first describe the modelling formalism they are based on: Labelled Transition Systems.

Definition 2.1 (Labelled Transition Systems)**.** A *Labelled Transition System* (LTS) is a quadruple $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$, such that:

- S is a (possibly uncountable) set of states;
- $S^0 \subseteq S$ is a non-empty set of initial states;
- L is a set of labels, each representing a different action. We take $\tau \notin L$ to stand for an internal (unobservable) action and define $L^\tau = L \cup \{\tau\}$;
- $\rightarrow \subseteq S \times L^\tau \times S$ is the transition relation. We use $s \xrightarrow{a} s'$ to denote $(s, a, s') \in \rightarrow$, write $s \xrightarrow{a}$ if there is an $s' \in S$ such that $s \xrightarrow{a} s'$, and $s \not\xrightarrow{a}$ if this is not the case. If $s \xrightarrow{a}$, we say that the action a is *enabled* in state s.

We use $S_\mathcal{A}$, $S^0_\mathcal{A}$, $L_\mathcal{A}$ and $\rightarrow_\mathcal{A}$ to denote the components of an LTS $\mathcal{A}$. These subscripts are left out when it is clear from the context which LTS is referred to.

Example 2.2. Figure 2(a) shows an LTS $\mathcal{A}$. □



(a) $\mathcal{A}$ (b) $\mathcal{B}$ (c) $det(\mathcal{B})$

Figure 2: Visual representation of the LTS $\mathcal{A}$ and the IOTSs $\mathcal{B}$ and $det(\mathcal{B})$. We represent states by circles, and transitions by arrows; each arrow in turn is labelled with the associated action for that particular transition. The initial state is marked by an arrow without a source state. Frow now on, we typically will not label individual states.

Often, in particular in the context of testing, it is desirable to be able to distinguish between actions that are initiated by the environment (inputs), and actions that are initiated by the system itself (outputs). To this end, we introduce Input-Output Transition Systems, which are an extension of regular LTSs.

Definition 2.3 (Input-Output Transition Systems)**.** An *Input-Output Transition System* (IOTS) is a quintuple $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow \rangle$, where L^{I} is a set of input labels and L^{O} a set of output labels such that $L^{\mathrm{I}} \cap L^{\mathrm{O}} = \emptyset$. We define $L = L^{\mathrm{I}} \cup L^{\mathrm{O}}$ and $L^{\tau} = L \cup \{\tau\}$, where $\tau \notin L$. S, S^0 and $\rightarrow$ are as defined for LTSs. Additionally, IOTSs must be *input-enabled*, i.e., $s \xrightarrow{a}$ for all $s \in S, a \in L^{\mathrm{I}}$.

Remark 2.4. Throughout this article we sometimes suffix a question mark (?) to the input labels and an exclamation mark (!) to the output labels, to help differentiating the two types. These are, however, not part of the label.

Note that IOTSs are similar to I/O automata [5, 4], except that the latter allow multiple internal actions, rather than τ only. All our results can easily be phrased in the I/O automata framework.

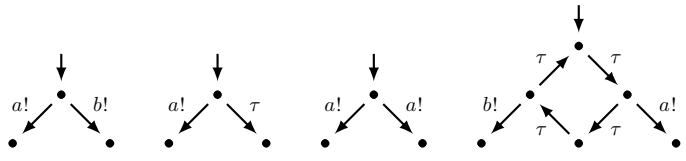
By requiring IOTSs to be input-enabled, any input initiated by the environment is never refused by the system. For deterministic systems (see Definition 2.7), this restriction can easily be lifted by adding a sink state which has self-loops for all possible actions, and adding transitions for the missing inputs to that sink state (so-called *demonic completion* [4, 15]). For nondeterministic systems, a solution is provided in [3].

Example 2.5. Figure 2(b) shows an IOTS $\mathcal{B}$. Note that since $L^{\mathrm{I}} = \{a\}$ and $s \xrightarrow{a}$ for every $s \in S$, $\mathcal{B}$ is input-enabled. □

We introduce the standard language-theoretic concepts for IOTSs.

Definition 2.6 (Notations)**.** Let $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow \rangle$ be an IOTS, then:

- A *path* in $\mathcal{A}$ is a (possibly infinite) sequence $\pi = s_0 a_1 s_1 \ldots s_n$ such that for all $1 \leq i \leq n$ we have $s_{i-1} \xrightarrow{a_i} s_i$ with $a_i \in L^{\tau}$. The set of all paths in $\mathcal{A}$ is denoted *paths*$(\mathcal{A})$.
- The path operators *first* and *last* yield the first and last state of a finite path, respectively, e.g., for $\pi = s_0 a_1 s_1 a_2 s_2$ we have *first*$(\pi) = s_0$ and *last*$(\pi) = s_2$. A path π is called *initial* if *first*$(\pi) \in S^0$.
- The path operator *trace* yields the sequence of actions that is obtained by erasing all states and τ-actions from a given path, e.g., for $\pi = s_0 a_1 s_1 \tau s_2 a_2 s_3$ we have *trace*$(\pi) = a_1 a_2$; we call such a



(a) $\mathcal{A}$ (b) $\mathcal{B}$ (c) $\mathcal{C}$ (d) $\mathcal{D}$

Figure 3: One deterministic ($\mathcal{A}$) and three nondeterministic ($\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$) IOTSs. The IOTS $\mathcal{D}$ is divergent.

sequence of actions a *trace* of $\mathcal{A}$. The *length* of a trace $\sigma = a_1 a_2 \dots a_n$, denoted $|\sigma|$, is the length of the corresponding sequence, i.e., $|\sigma| = |a_1 a_2 \dots a_n| = n$.

- Given an action a and a set of actions P, we denote by $a \restriction P$ the *projection* of a on P, i.e., $a \restriction P = a$ if $a \in P$, and $a \restriction P = \epsilon$ otherwise. The projection of a trace $\sigma = a\sigma'$ on a set of actions P follows naturally from this: $\sigma \restriction P = a\sigma' \restriction P = a \restriction P + \sigma' \restriction P$. Finally, the projection of a set of traces T on a set of actions P is defined as $T \restriction P = \{ \sigma \restriction P \mid \sigma \in T \}$.
- If there is a finite path π in $\mathcal{A}$ such that $\mathit{first}(\pi) = s$, $\mathit{last}(\pi) = s'$ and $\mathit{trace}(\pi) = \sigma$, we write $s \stackrel{\sigma}{\Longrightarrow} s'$; if there exists an $s' \in S$ such that $s \stackrel{\sigma}{\Longrightarrow} s'$, we write $s \stackrel{\sigma}{\Longrightarrow}$, and $s \stackrel{\sigma}{\not\Longrightarrow}$ if this is not the case.
- For a finite trace σ and state $s \in S$, we denote by $\mathit{reach}(s, \sigma)$ the set of states in $\mathcal{A}$ (possibly empty) that can be reached from s via σ, i.e., $\mathit{reach}(s, \sigma) = \{ s' \in S \mid s \stackrel{\sigma}{\Longrightarrow} s' \}$. Similarly, for a given finite trace σ and a set of states $S' \subseteq S$, we denote by $\mathit{reach}(S', \sigma)$ the set of states in $\mathcal{A}$ that can be reached from any of the states in S' via σ, i.e., $\mathit{reach}(S', \sigma) = \{ s \in S \mid \exists s' \in S' \,.\, s' \stackrel{\sigma}{\Longrightarrow} s \}$.
- For a finite trace σ and state $s \in S$, $\mathit{out}(s, \sigma)$ is the set of output actions that are enabled in any of the states reachable from s by σ, i.e., $\mathit{out}(s, \sigma) = \{ a \in L^{\mathrm{O}} \mid \exists s' \in \mathit{reach}(s, \sigma) \,.\, s' \stackrel{a}{\Longrightarrow} \}$. We use the shorthand $\mathit{out}(s)$ for the case $\mathit{out}(s, \epsilon)$, i.e., the set of output actions that are enabled in s itself.
- For every $s \in S$ we denote by $\mathit{traces}(s)$ the set of all traces of $\mathcal{A}$ that correspond to paths that start in s, i.e., $\mathit{traces}(s) = \{ \mathit{trace}(\pi) \mid \pi \in \mathit{paths}(\mathcal{A}) \wedge \mathit{first}(\pi) = s \}$. We denote by $\mathit{traces}(\mathcal{A}) = \bigcup_{s \in S^0} \mathit{traces}(s)$ the set of all traces that correspond to initial paths in $\mathcal{A}$. Two IOTSs $\mathcal{B}$ and $\mathcal{C}$ are *trace equivalent* if $\mathit{traces}(\mathcal{B}) = \mathit{traces}(\mathcal{C})$.

A fundamental concept in automata theory is determinism.

Definition 2.7 (Determinism)**.** An IOTS $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow \rangle$ is *deterministic* if for all $s, s', s'' \in S, a \in L$ we have that $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ imply $a \neq \tau$ and $s' = s''$. Otherwise, $\mathcal{A}$ is *nondeterministic*.

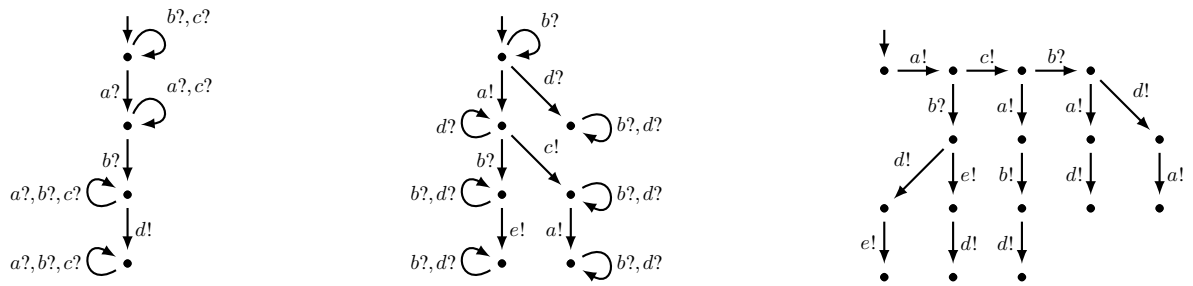
Example 2.8. Figure 3 shows some deterministic and nondeterministic IOTSs. □

Lastly, we introduce the notions of convergence and divergence.

Definition 2.9 (Divergence)**.** Given an IOTS $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow \rangle$, a state $s \in S$ of $\mathcal{A}$ is *divergent* if there is an infinite path $s_0 a_1 s_1 a_2 s_2 \dots$ with $s_0 = s$ and $s_i \in S$, that contains only τ transitions, i.e., $a_i = \tau$ for all i. An IOTS is called divergent if it contains at least one such state, otherwise it is *convergent*.

For the purposes of this paper, we require all IOTSs to be convergent.

Example 2.10. Figure 3(d) shows the divergent IOTS $\mathcal{D}$. Clearly, it is possible for $\mathcal{D}$ to perform an infinite sequence of τ-transitions by continuously looping through the innermost four states. □



(a) $\mathcal{A}$ (b) $\mathcal{B}$ (c) $\mathcal{A} \parallel \mathcal{B}$

Figure 4: The IOTSs $\mathcal{A}$ and $\mathcal{B}$, and their parallel composition $\mathcal{A} \parallel \mathcal{B}$. Note that we have left out some of the $b?$-labelled self-loops from the visualisation of $\mathcal{A} \parallel \mathcal{B}$ to reduce clutter.

2.3 Operations on IOTSs

In this section, we introduce several standard operations on IOTSs. First, every nondeterministic IOTS can be transformed into a deterministic IOTS [9]; the latter is called the determinisation of the original IOTS and is trace equivalent to it [2]. Using this operator, modelling effort is saved since no attention needs to be paid to making the specification deterministic.

Definition 2.11 (Determinisation)**.** The *determinisation* of an IOTS $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow_{\mathcal{A}} \rangle$ is the IOTS $\mathit{det}(\mathcal{A}) = \langle T, \{ S^0 \}, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow_{\mathrm{d}} \rangle$ such that $T = \wp(S) \setminus \emptyset$ and $\rightarrow_{\mathrm{d}} = \{ (U, a, V) \in T \times L \times T \mid V = \mathit{reach}_{\mathcal{A}}(U, a) \wedge V \neq \emptyset \}$.

Example 2.12. Consider the nondeterministic IOTS $\mathcal{B}$ shown in Figure 2(b). Its corresponding determinisation $\mathit{det}(\mathcal{B})$ is shown in Figure 2(c). □

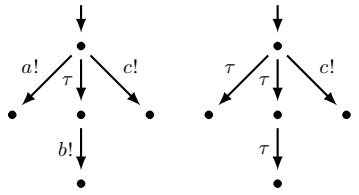
Second, we define the parallel composition operator. This operator is fundamental in modelling frameworks for component-based design. It allows one to build complex system models from smaller ones, thus breaking up the specification of a system into manageable pieces. Parallel composed IOTSs synchronise on shared inputs and complementary input-output pairs [4].

Definition 2.13 (Parallel composition of IOTSs)**.** Given are two IOTSs $\mathcal{A} = \langle S_{\mathcal{A}}, S^0_{\mathcal{A}}, L^{\mathrm{I}}_{\mathcal{A}}, L^{\mathrm{O}}_{\mathcal{A}}, \rightarrow_{\mathcal{A}} \rangle$ and $\mathcal{B} = \langle S_{\mathcal{B}}, S^0_{\mathcal{B}}, L^{\mathrm{I}}_{\mathcal{B}}, L^{\mathrm{O}}_{\mathcal{B}}, \rightarrow_{\mathcal{B}} \rangle$ such that $L^{\mathrm{O}}_{\mathcal{A}} \cap L^{\mathrm{O}}_{\mathcal{B}} = \emptyset$. The *parallel composition* of $\mathcal{A}$ and $\mathcal{B}$ is the IOTS $\mathcal{A} \parallel \mathcal{B} = \langle S_{\mathcal{A}\parallel\mathcal{B}}, S^0_{\mathcal{A}\parallel\mathcal{B}}, L^{\mathrm{I}}_{\mathcal{A}\parallel\mathcal{B}}, L^{\mathrm{O}}_{\mathcal{A}\parallel\mathcal{B}}, \rightarrow_{\mathcal{A}\parallel\mathcal{B}} \rangle$, where $S_{\mathcal{A}\parallel\mathcal{B}} = S_{\mathcal{A}} \times S_{\mathcal{B}}$, $S^0_{\mathcal{A}\parallel\mathcal{B}} = S^0_{\mathcal{A}} \times S^0_{\mathcal{B}}$, $L^{\mathrm{I}}_{\mathcal{A}\parallel\mathcal{B}} = (L^{\mathrm{I}}_{\mathcal{A}} \cup L^{\mathrm{I}}_{\mathcal{B}}) \setminus (L^{\mathrm{O}}_{\mathcal{A}} \cup L^{\mathrm{O}}_{\mathcal{B}})$, and $L^{\mathrm{O}}_{\mathcal{A}\parallel\mathcal{B}} = L^{\mathrm{O}}_{\mathcal{A}} \cup L^{\mathrm{O}}_{\mathcal{B}}$. The transition relation $\rightarrow_{\mathcal{A}\parallel\mathcal{B}}$ is defined as follows:

$$
\begin{aligned}
\rightarrow_{\mathcal{A}\parallel\mathcal{B}} \;=\; & \{ ((s,t), a?, (s',t')) \mid s \xrightarrow{a?}_{\mathcal{A}} s' \wedge t \xrightarrow{a?}_{\mathcal{B}} t' \} \\
\cup\; & \{ ((s,t), a!, (s',t')) \mid s \xrightarrow{a?}_{\mathcal{A}} s' \wedge t \xrightarrow{a!}_{\mathcal{B}} t' \} \\
\cup\; & \{ ((s,t), a!, (s',t')) \mid s \xrightarrow{a!}_{\mathcal{A}} s' \wedge t \xrightarrow{a?}_{\mathcal{B}} t' \} \\
\cup\; & \{ ((s,t), a, (s',t)) \mid s \xrightarrow{a}_{\mathcal{A}} s' \wedge t \in S_{\mathcal{B}} \wedge a \in L^{\tau}_{\mathcal{A}} \setminus L_{\mathcal{B}} \} \\
\cup\; & \{ ((s,t), a, (s,t')) \mid t \xrightarrow{a}_{\mathcal{B}} t' \wedge s \in S_{\mathcal{A}} \wedge a \in L^{\tau}_{\mathcal{B}} \setminus L_{\mathcal{A}} \}
\end{aligned}
$$

Thus, $L_{\mathcal{A}\parallel\mathcal{B}} = L^{\mathrm{I}}_{\mathcal{A}\parallel\mathcal{B}} \cup L^{\mathrm{O}}_{\mathcal{A}\parallel\mathcal{B}} = L_{\mathcal{A}} \cup L_{\mathcal{B}}$.

Example 2.14. Figure 4 shows two IOTSs $\mathcal{A}$ and $\mathcal{B}$, and their parallel composition $\mathcal{A} \parallel \mathcal{B}$. We have $L^{\mathrm{I}}_{\mathcal{A}} = \{a, b, c\}$, $L^{\mathrm{O}}_{\mathcal{A}} = \{d\}$, $L^{\mathrm{I}}_{\mathcal{B}} = \{b, d\}$, and $L^{\mathrm{O}}_{\mathcal{B}} = \{a, c, e\}$. Note that indeed $L^{\mathrm{O}}_{\mathcal{A}} \cap L^{\mathrm{O}}_{\mathcal{B}} = \emptyset$, as required; therefore, by Definition 2.13, $L^{\mathrm{I}}_{\mathcal{A}\parallel\mathcal{B}} = \{b\}$ and $L^{\mathrm{O}}_{\mathcal{A}\parallel\mathcal{B}} = \{a, c, d, e\}$. □



(a) $\mathcal{A}$ (b) $\mathit{hide}(\mathcal{A}, \{a, b\})$

Figure 5: The IOTSs $\mathcal{A}$ and $\mathit{hide}(\mathcal{A}, \{a, b\})$.

Finally, it is often useful to hide certain actions of a given IOTS, thereby essentially renaming the corresponding labels to τ. For example, when parallel composing two IOTSs, some actions are only used for synchronisation; after composition, they are not needed anymore.

Definition 2.15 (Action hiding in IOTSs)**.** Let $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow_{\mathcal{A}} \rangle$ be an IOTS and $H \subseteq L^{\mathrm{O}}$ a set of output labels, then one can *hide* H in $\mathcal{A}$ to get the IOTS $\mathit{hide}(\mathcal{A}, H) = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}} \setminus H, \rightarrow_{\mathrm{h}} \rangle$, where $\rightarrow_{\mathrm{h}} = \{ (s, a, s') \in \rightarrow_{\mathcal{A}} |\ a \notin H \} \cup \{ (s, \tau, s') \in S \times \{\tau\} \times S \mid \exists a \in H \,.\, (s, a, s') \in \rightarrow_{\mathcal{A}} \}$.

Thus, we only allow output actions to be hidden. Furthermore, we do not allow action hiding to lead to divergent IOTSs, i.e., the hiding of outputs may not lead to the creation of τ-loops.

Example 2.16. Figure 5 shows the IOTSs $\mathcal{A}$ with $L^{\mathrm{O}}_{\mathcal{A}} = \{a, b, c\}$ and $\mathcal{B} = \mathit{hide}(\mathcal{A}, \{a, b\})$. □

From now on, we typically won't show all input-labelled self-loops in visualisations of IOTSs, to reduce clutter. Thus, we assume that every IOTS is input-enabled (unless mentioned otherwise).

2.4 Properties of IOTSs

IOTSs possess several interesting properties, that will also be of use when working with QTSs later on. We provide three results, showing that (1) hiding of actions corresponds to projection of traces, (2) parallel composition does not introduce new traces when projecting on the alphabet of either one of the components, and (3) parallel composition of components that synchronise on all actions yields the intersection of the traces of the components.

Proposition 2.17. *Given an IOTS $\mathcal{A}$ and a set of labels $H \subseteq L^{\mathrm{O}}_{\mathcal{A}}$, we have $\mathit{traces}(\mathit{hide}(\mathcal{A}, H)) = \mathit{traces}(\mathcal{A}) \restriction (L_{\mathcal{A}} \setminus H)$.*

Proposition 2.18. *Given two IOTSs $\mathcal{A}$ and $\mathcal{B}$, we have $\mathit{traces}(\mathcal{A} \parallel \mathcal{B}) \restriction L_{\mathcal{A}} \subseteq \mathit{traces}(\mathcal{A})$ and $\mathit{traces}(\mathcal{A} \parallel \mathcal{B}) \restriction L_{\mathcal{B}} \subseteq \mathit{traces}(\mathcal{B})$.*

Proposition 2.19. *Given two IOTSs $\mathcal{A}, \mathcal{B}$ with $L_{\mathcal{A}} = L_{\mathcal{B}}$, we have $\mathit{traces}(\mathcal{A} \parallel \mathcal{B}) = \mathit{traces}(\mathcal{A}) \cap \mathit{traces}(\mathcal{B})$.*

3 Quiescent Transition Systems

3.1 Basic notions and requirements

IOTSs can be used to model the inputs and outputs of a system, but cannot explicitly express the observation of the absence of outputs, also called the observation of quiescence [14, 11, 7]. To fill this void, we introduce Quiescent Transition Systems. These automata can be used to model all possible observations for a particular system, including quiescence, and can thus be thought of as 'observation automata'.

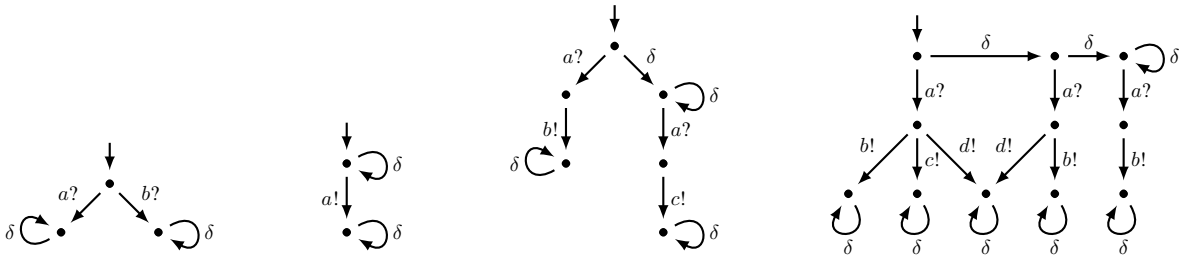


Figure 6: The QTSs $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ that do not satisfy rule R1, rule R2, rule R3 and rule R4, respectively.

They are based on Tretmans' suspension automata [11], in the sense that a δ-transition represents the observation of quiescence. A basic variant of QTSs was already used in [10] in a testing framework. However, restrictions for QTSs to prohibit counterintuitive behaviour, as well as characteristics and closure properties of such models, have never been studied before.

Definition 3.1 (Quiescence)**.** Let $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow \rangle$ be an IOTS. A state $s \in S$ is called *quiescent* if $\nexists a \in L^{\mathrm{O}} \cup \{\tau\} \,.\, s \xrightarrow{a}$, i.e., no outputs or internal transitions can be executed in state s.

A system in a quiescent state will be idle until a new input is supplied. Note that a state s that can still perform a τ-step is not considered quiescent, even if there is no output $a! \in L^{\mathrm{O}}$ such that $s \stackrel{a!}{\Longrightarrow}$. After all, since quiescence signifies that a system is idle indefinitely, it would not make sense if there are still internal steps possible. Moreover, from a more technical point of view, this ensures that QTSs are closed under hiding and that hiding and deltafication (see Section 4) are commutative.

Definition 3.2 (Quiescent Transition Systems)**.** A *Quiescent Transition System* (QTS) is an IOTS $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}} \cup \{\delta\}, \rightarrow \rangle$, where $\delta \notin L^{\mathrm{I}} \cup L^{\mathrm{O}}$ is a special output label that is used to denote the observation of quiescence. We define $L = L^{\mathrm{I}} \cup L^{\mathrm{O}}$, $L^{\delta}_{\tau} = L^{\mathrm{I}} \cup L^{\mathrm{O}} \cup \{\delta, \tau\}$ and let $\rightarrow \subseteq S \times L^{\delta}_{\tau} \times S$ be the transition relation. Like regular IOTSs, QTSs must be input-enabled, i.e., $s \xrightarrow{a}$ for all $s \in S, a \in L^{\mathrm{I}}$. Furthermore, we also require the following rules to hold for all states $s, s', s'' \in S$:

Rule R1 (Quiescence should be observable)**:** if s is quiescent, then $s \xrightarrow{\delta}$**.**

This rule requires that each quiescent state has an outgoing δ-transition. Consider the QTS $\mathcal{A}$ in Figure 6(a). This QTS does not satisfy this rule, as the topmost state cannot produce any outputs, but neither can execute an outgoing δ-transition.

Rule R2 (No outputs after quiescence)**:** if $s \xrightarrow{\delta} s'$, then s' is quiescent**.**

This rule ensures that the system is idle after a δ-transition, i.e., it cannot provide an output (except for δ itself) or execute an internal transition, before another input is provided. In Figure 6(b) the QTS $\mathcal{B}$ is shown which does not satisfy this rule. From the top-most state it is possible to first observe quiescence (the δ-transition) and after that the $a!$ output, without an intermediate input. Since there is no particular observation duration associated with quiescence, but quiescence rather means that the system idles indefinitely, this is clearly counterintuitive and therefore disallowed.

Rule R3 (Quiescence does not enable new behaviour)**:** if $s \xrightarrow{\delta} s'$, then $\mathit{traces}(s') \subseteq \mathit{traces}(s)$**.**

Given a state s' of a QTS that is reached from another state s by a δ-transition (i.e., observation of quiescence), this rule demands that any trace that can be executed starting from state s' can

also be executed in state s, i.e., the observation of quiescence may not introduce any new possible observations. This rule was added to prevent situations like the one depicted in Figure 6(c). For QTS $\mathcal{C}$ it is possible to observe the output $c!$ (after the input $a?$) after first observing quiescence, but if quiescence is not observed (because, for instance, the input $a?$ was directly given) the output $b!$ will be observed after the input $a?$ instead. Thus, the prior observation of quiescence allows new behaviour to be observed later on, which is counterintuitive. This rule therefore ensures that all behaviour that can be observed after observing quiescence can also be observed before.

Rule R4 (Continued quiescence preserves behaviour)**:** if $s \xrightarrow{\delta} s'$ and $s' \xrightarrow{\delta} s''$, then $\mathit{traces}(s') = \mathit{traces}(s'')$**.**

A QTS $\mathcal{D}$ that violates this rule is shown in Figure 6(d). From the initial state an observation of quiescence can be made, which then leads to a new state where the trace ac can no longer be observed. From the latter state another observation of quiescence can be made, which leads to another state where the trace ad can no longer be observed. Rule R3 allows this, but as there is no particular time interval associated with the observation of quiescence, this does not make sense. We therefore have the additional requirement that any observations possible after two (or more) consecutive observations of quiescence should also be possible after a single observation of quiescence, and vice versa.

Just as for IOTSs, we require QTSs to be convergent. The reason for this is that divergent systems have states that can execute internal transitions infinitely often and never output anything. Considering such a state quiescent would be nonintuitive, as it is not idle (and might even be able to provide an output action, even though it does not show it). Not considering it quiescent would also be nonintuitive, because of the possibility that no visible behaviour is observed.

Note that the converse of rule R1 is not required, e.g., we do not forbid that a state has both a δ-transition and an output action enabled. This situation can arise during the determinisation of a QTS, as we will see in Section 4. However, the δ-transition should still end up in a quiescent state, as required by rule R2. Also note that a trace of a QTS can contain a sequence of δ-actions. Although this might seem odd, it corresponds to the practical testing scenario of observing a time-out rather than an output more than once in a row.

Since computing trace inclusion is expensive [1], an easier way to ensure that a QTS complies to rule R3 is to make sure the following alternative rule R3′ holds for all states $s, s', s'' \in S$.

Rule R3′: if $s \xrightarrow{\delta} s'$ and $\exists a? \in L_{\mathrm{I}}$ such that $s' \xrightarrow{a?}$ s'' then also $s \xrightarrow{a?} s''$**.**

Clearly, any QTS that satisfies rule R3′ also satisfies rule R3.

Similarly, conformance to rule R4 for a QTS can be achieved by making sure that the following alternative rule R4′ holds for all states $s, s' \in S$ of the QTS.

Rule R4′: if $s \xrightarrow{\delta} s'$ then $s' \xrightarrow{\delta} s'$, and if also $s' \xrightarrow{\delta} s''$ then $s'' = s'$**.**

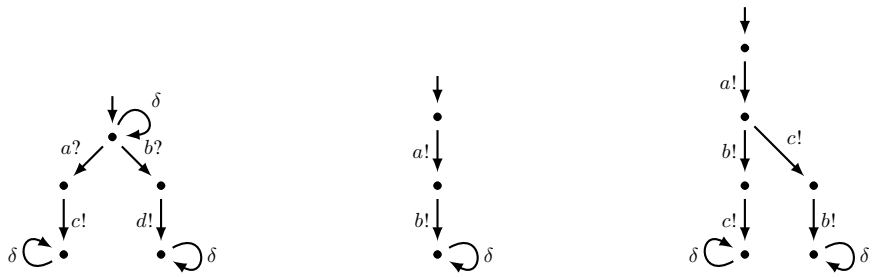
Clearly, any QTS that satisfies rule R4′ also satisfies rule R4.

When comparing the structure of two QTSs $\mathcal{A}$ and $\mathcal{B}$, the notion of isomorphisms can be useful.

Definition 3.3 (Isomorphic QTSs)**.** Two QTSs $\mathcal{A} = \langle S_{\mathcal{A}}, S^0_{\mathcal{A}}, L^{\mathrm{I}}_{\mathcal{A}}, L^{\mathrm{O}}_{\mathcal{A}} \cup \{\delta\}, \rightarrow_{\mathcal{A}} \rangle$ and $\mathcal{B} = \langle S_{\mathcal{B}}, S^0_{\mathcal{B}}, L^{\mathrm{I}}_{\mathcal{B}}, L^{\mathrm{O}}_{\mathcal{B}} \cup \{\delta\}, \rightarrow_{\mathcal{B}} \rangle$ are called *isomorphic*, denoted $\mathcal{A} \cong \mathcal{B}$, if there exists a bijection $h\colon S_{\mathcal{A}} \to S_{\mathcal{B}}$ (called an isomorphism) such that the following holds:

1. for all $s_0 \in S^0_{\mathcal{A}}$ there exists a $t_0 \in S^0_{\mathcal{B}}$ such that $h(s_0) = t_0$, and vice versa;
2. $s \xrightarrow{a}_{\mathcal{A}} s'$ if and only if $h(s) \xrightarrow{a}_{\mathcal{B}} h(s')$, for all $s, s' \in S_{\mathcal{A}}$ and $a \in L_{\mathcal{A}} \cup \{\delta, \tau\}$.

Thus, two isomorphic QTSs are structurally equivalent.



(a) $\mathcal{A}$ (b) $\mathcal{B}$ (c) $\mathcal{A} \parallel \mathcal{B}$

Figure 7: The QTSs $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{A} \parallel \mathcal{B}$.

3.2 Operations on QTSs

Since QTSs are a specialisation of IOTSs, all operations that are applicable to IOTSs (such as determinisation, parallel composition and hiding of actions) are also applicable to QTSs. Determinisation for QTSs is exactly the same as for IOTSs, but there are some minor differences for parallel composition and action hiding.

Definition 3.4 (Parallel composition of QTSs)**.** Let $\mathcal{A} = \langle S_{\mathcal{A}}, S^0_{\mathcal{A}}, L^{\mathrm{I}}_{\mathcal{A}}, L^{\mathrm{O}}_{\mathcal{A}} \cup \{\delta\}, \rightarrow_{\mathcal{A}} \rangle$ and $\mathcal{B} = \langle S_{\mathcal{B}}, S^0_{\mathcal{B}}, L^{\mathrm{I}}_{\mathcal{B}}, L^{\mathrm{O}}_{\mathcal{B}} \cup \{\delta\}, \rightarrow_{\mathcal{B}} \rangle$ be two QTSs such that $L^{\mathrm{O}}_{\mathcal{A}} \cap L^{\mathrm{O}}_{\mathcal{B}} = \emptyset$. The *parallel composition* of $\mathcal{A}$ and $\mathcal{B}$ is then the QTS $\mathcal{A} \parallel \mathcal{B} = \langle S_{\mathcal{A}\parallel\mathcal{B}}, S^0_{\mathcal{A}\parallel\mathcal{B}}, L^{\mathrm{I}}_{\mathcal{A}\parallel\mathcal{B}}, L^{\mathrm{O}}_{\mathcal{A}\parallel\mathcal{B}} \cup \{\delta\}, \rightarrow_{\mathcal{A}\parallel\mathcal{B}} \rangle$, where $S_{\mathcal{A}\parallel\mathcal{B}} = S_{\mathcal{A}} \times S_{\mathcal{B}}$, $S^0_{\mathcal{A}\parallel\mathcal{B}} = S^0_{\mathcal{A}} \times S^0_{\mathcal{B}}$, $L^{\mathrm{I}}_{\mathcal{A}\parallel\mathcal{B}} = (L^{\mathrm{I}}_{\mathcal{A}} \cup L^{\mathrm{I}}_{\mathcal{B}}) \setminus (L^{\mathrm{O}}_{\mathcal{A}} \cup L^{\mathrm{O}}_{\mathcal{B}})$, and $L^{\mathrm{O}}_{\mathcal{A}\parallel\mathcal{B}} = L^{\mathrm{O}}_{\mathcal{A}} \cup L^{\mathrm{O}}_{\mathcal{B}}$. $\rightarrow_{\mathcal{A}\parallel\mathcal{B}}$ is defined as follows:

$$
\begin{aligned}
\rightarrow_{\mathcal{A}\parallel\mathcal{B}} \;=\; & \{\, ((s,t), a?, (s',t')) \mid s \xrightarrow{a?}_{\mathcal{A}} s' \wedge t \xrightarrow{a?}_{\mathcal{B}} t' \,\} \\
& \cup \{\, ((s,t), a!, (s',t')) \mid s \xrightarrow{a?}_{\mathcal{A}} s' \wedge t \xrightarrow{a!}_{\mathcal{B}} t' \,\} \\
& \cup \{\, ((s,t), a!, (s',t')) \mid s \xrightarrow{a!}_{\mathcal{A}} s' \wedge t \xrightarrow{a?}_{\mathcal{B}} t' \,\} \\
& \cup \{\, ((s,t), \delta, (s',t')) \mid (s,\delta,s') \in \rightarrow_{\mathcal{A}} \wedge (t,\delta,t') \in \rightarrow_{\mathcal{B}} \,\} \\
& \cup \{\, ((s,t), a, (s',t)) \mid s \xrightarrow{a}_{\mathcal{A}} s' \wedge t \in S_{\mathcal{B}} \wedge a \in L^{\tau}_{\mathcal{A}} \setminus L_{\mathcal{B}} \,\} \\
& \cup \{\, ((s,t), a, (s,t')) \mid t \xrightarrow{a}_{\mathcal{B}} t' \wedge s \in S_{\mathcal{A}} \wedge a \in L^{\tau}_{\mathcal{B}} \setminus L_{\mathcal{A}} \,\}
\end{aligned}
$$

Thus, when compared to the parallel composition of regular IOTSs, we have the additional requirement that parallel composed QTSs must synchronise on the δ-action, as the observation of quiescence can be made simultaneously for multiple QTSs. Again, we find that $L_{\mathcal{A}\parallel\mathcal{B}} = L^{\mathrm{I}}_{\mathcal{A}\parallel\mathcal{B}} \cup L^{\mathrm{O}}_{\mathcal{A}\parallel\mathcal{B}} = L_{\mathcal{A}} \cup L_{\mathcal{B}}$.

Example 3.5. See Figure 7(a) for the visual representation of a QTS $\mathcal{A}$ which satifies all the requirements for QTSs listed in Definition 3.2. Figure 7(b) shows another QTS $\mathcal{B}$ and Figure 7(c) shows the parallel composition of the QTSs $\mathcal{A}$ and $\mathcal{B}$. □

Definition 3.6 (Action hiding in QTSs)**.** Let $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}} \cup \{\delta\}, \rightarrow_{\mathcal{A}} \rangle$ be a QTS and $H \subseteq L^{\mathrm{O}}$ a set of labels, then one can *hide* H in $\mathcal{A}$ to obtain the IOTS *hide*$(\mathcal{A}, H) = \langle S, S^0, L^{\mathrm{I}}, (L^{\mathrm{O}} \setminus H) \cup \{\delta\}, \rightarrow_{\mathrm{h}} \rangle$, where $\rightarrow_{\mathrm{h}} = \{\, (s,a,s') \in \rightarrow_{\mathcal{A}} \mid a \notin H \,\} \cup \{\, (s,\tau,s') \in S \times \{\tau\} \times S \mid \exists a \in H \,.\, (s,a,s') \in \rightarrow_{\mathcal{A}} \,\}$.

We do not allow the special output label δ to be hidden, as this label doesn't represent a specific output but rather (the observation of) a lack of outputs. Furthermore, as for IOTSs, we do not allow action hiding to lead to divergent QTSs, i.e., hiding may not lead to the creation of τ-loops.

3.3 Properties of QTSs

In this section, we present several interesting properties of QTSs. First of all, it turns out that our model is closed under all operations defined thus far: determinisation, action hiding and parallel composition. Therefore, these operations are indeed well-defined for QTSs.

Theorem 3.7. *QTSs are closed under determinisation, action hiding and parallel composition. Hence, given two QTSs $\mathcal{A}$, $\mathcal{B}$ and a set of labels $H \subseteq L^{\mathrm{O}}_{\mathcal{A}}$, also $det(\mathcal{A})$, $hide(\mathcal{A}, H)$ and $\mathcal{A} \parallel \mathcal{B}$ are QTSs.*

We also provide two results concerning the traces of parallel compositions of QTSs, generalising the corresponding properties of IOTSs as given in Section 2.4. First, parallel composition does not introduce new traces when projecting on the alphabet of either one of the components. That is, when disregarding the actions of component $\mathcal{B}$ in the traces of $\mathcal{A} \parallel \mathcal{B}$, the resulting set of traces is a subset of the traces of $\mathcal{A}$. It then quite easily follows that, when two parallel QTSs have the same alphabet (and hence synchronise on all actions), we obtain a subset of the intersection of their individual traces.

Proposition 3.8. *Given two QTSs $\mathcal{A}$ and $\mathcal{B}$, we have $traces(\mathcal{A} \parallel \mathcal{B}) \upharpoonright (L_{\mathcal{A}} \cup \{\delta\}) \subseteq traces(\mathcal{A})$ and $traces(\mathcal{A} \parallel \mathcal{B}) \upharpoonright (L_{\mathcal{B}} \cup \{\delta\}) \subseteq traces(\mathcal{B})$.*

Proposition 3.9. *Given two QTSs $\mathcal{A}$, $\mathcal{B}$ with $L_{\mathcal{A}} = L_{\mathcal{B}}$, we have $traces(\mathcal{A} \parallel \mathcal{B}) = traces(\mathcal{A}) \cap traces(\mathcal{B})$.*

4 From IOTS to QTS: deltafication

Usually, the specification and implementation of a system (under development) are given as IOTSs, rather than QTSs. During testing, however, we typically observe the outputs of the system generated in response to inputs from the environment; thus, it is useful to be able to refer to the absence of outputs (i.e., quiescence) explicitly. Hence, we need a way to convert an IOTS to a QTS that captures all possible observations of it, including quiescence; this conversion is called deltafication and is described in [11, 12, 13]. First, however, we need to introduce an additional condition C1 for IOTSs, for every $s, s' \in S$:

Condition C1: if $s \xrightarrow{\delta} s'$, then for all $\sigma \in traces(s')$:

$$\exists t' \in reach(s', \sigma)\ .\ t' \text{ is quiescent } \wedge\ t' \not\xrightarrow{\delta} \ \Rightarrow \forall t \in reach(s, \sigma)\ .\ t \text{ is quiescent } \wedge\ t \not\xrightarrow{\delta}$$

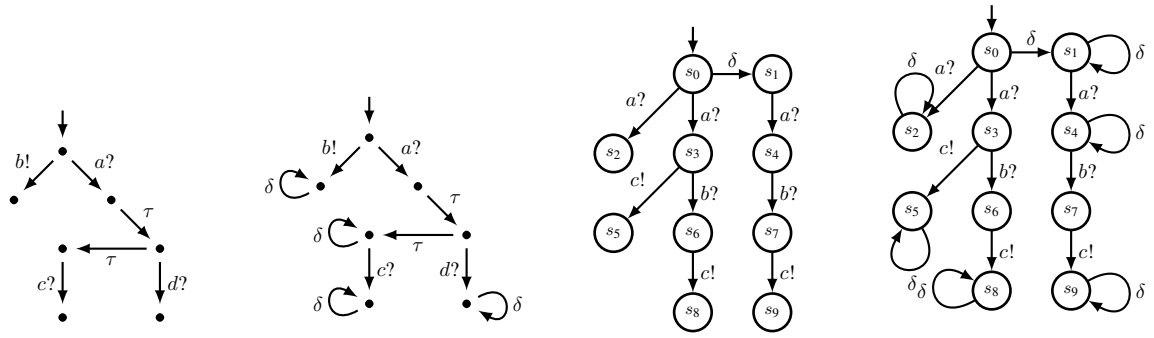
Condition C1 requires that if any trace $\sigma \in traces(s')$, when executed from s', can lead to a state that is quiescent and cannot execute a δ-transition, then it must always lead to a state that is quiescent and cannot execute a δ-transition when executed from s. This condition is weaker than R1, and allows us to determine the deltafication of systems that already contain some δ-transitions without requiring a δ-transition from every quiescent state. Note that any IOTS without δ-transitions vacuously satisfies C1.

Definition 4.1 (Deltafication)**.** Given an IOTS $\mathcal{A} = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow_{\mathcal{A}} \rangle$ that for all $s, s' \in S$ satisfies deltafication condition C1, and rules R2, R3 and R4 (see Definition 3.2), we define the *deltafication* of $\mathcal{A}$ as the QTS $\delta(\mathcal{A}) = \langle S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}} \cup \{\delta\}, \rightarrow_{\delta} \rangle$ where $\rightarrow_{\delta} = \rightarrow_{\mathcal{A}} \cup \{ (s, \delta, s) \in S \times \{\delta\} \times S \mid s \text{ is quiescent } \wedge\ s \not\xrightarrow{\delta}_{\mathcal{A}} \}$.

Example 4.2. An IOTS $\mathcal{A}$ and its deltafication $\delta(\mathcal{A})$ are shown in Figure 8(a) and 8(b), respectively. □

Remark 4.3. To see why condition C1 is necessary, consider the IOTS $\mathcal{B}$ and its deltafication $\delta(\mathcal{B})$ shown in Figure 8(c) and Figure 8(d), respectively; the states have been labelled for convenience. $\mathcal{B}$ does not satisfy condition C1, since $s_0 \xrightarrow{\delta} s_1$, $s_4 \in reach(s_1, a)$ and s_4 is quiescent and $s_4 \not\xrightarrow{\delta}$, but $s_3 \in reach(s_0, a)$ and s_3 is not quiescent. As a consequence, the deltafication $\delta(\mathcal{B})$ is not a valid QTS: for $\delta(\mathcal{B})$ we have $a\delta bc \in traces(s_1)$, but $a\delta bc \notin traces(s_0)$, thereby violating rule R3.

A more liberal version of C1, where the second quantification is changed to an existential one, would not be strong enough to prevent this: it would not forbid this example, as $s_2 \in reach(s_0, a)$ is quiescent and cannot do a δ-transition.



(a) $\mathcal{A}$ (b) $\delta(\mathcal{A})$ (c) $\mathcal{B}$ (d) $\delta(\mathcal{B})$

Figure 8: Deltafications of the IOTSs $\mathcal{A}$ and $\mathcal{B}$.

4.1 Validity of deltafication

Now, we present several interesting properties regarding the deltafication of IOTSs and QTSs. First, we show that deltafication indeed yields a valid QTS, and that it is idempotent.

Lemma 4.4. *Given an IOTS $\mathcal{A}$ that satisfies condition C1 and rules R2, R3 and R4, $\delta(\mathcal{A})$ is a QTS.*

Proposition 4.5. *Deltafication is idempotent, i.e., given an IOTS $\mathcal{A}$ that satisfies condition C1 and rules R2, R3 and R4, we have $\delta(\delta(\mathcal{A})) = \delta(\mathcal{A})$.*

Any IOTS $\mathcal{A}$ with $\delta \notin L_{\mathcal{A}}$ vacuously satisfies condition C1 and rules R2, R3 and R4. Therefore, the following theorem follows directly from Lemma 4.4.

Theorem 4.6. *Given an IOTS $\mathcal{A}$ such that $\delta \notin L_{\mathcal{A}}$, $\delta(\mathcal{A})$ is a QTS.*

By Definition 3.2, QTSs are IOTSs that satisfy rules R1, R2, R3 and R4. Since every state s in a QTS enables at least one output action or δ (due to rule R1), it never occurs that s is quiescent and does not enable a δ-transition, and hence every QTS satisfies condition C1 vacuously.

By Lemma 4.4, this immediately implies the following theorem.

Theorem 4.7. *QTSs are closed under deltafication, i.e., given a QTS $\mathcal{A}$, $\delta(\mathcal{A})$ is also a QTS.*

4.2 Commutativity results

In this section we investigate the commutativity of deltafication with determinisation, action hiding and parallel composition. We will show that parallel composition can safely be swapped with deltafication, but that determinisation has to precede deltafication to get sensible results. Also, we show that action hiding does not commute with deltafication.

Proposition 4.8. *Deltafication and determinisation do not commute, i.e., given an IOTS $\mathcal{A}$ that satisfies condition C1 and rules R2, R3 and R4, it is not necessarily the case that $det(\delta(\mathcal{A})) \cong \delta(det(\mathcal{A}))$.*

Proof. Observe the IOTS $\mathcal{A}$, its determinisation $det(\mathcal{A})$ and deltafication $\delta(\mathcal{A})$ in Figure 9(a,b,c). Clearly, the deltafication of the determinisation of $\mathcal{A}$ (i.e., $\delta(det(\mathcal{A}))$), shown in Figure 9(d), results in an incorrect observation automaton, as it does not model the fact that in the nondeterministic QTS $\delta(\mathcal{A})$ quiescence may be observed after an initial $a?$ input, as required by rule R1.

Contrary to the deltafication of the determinisation of $\mathcal{A}$, the determinisation of the deltafication of $\mathcal{A}$ (i.e., $det(\delta(\mathcal{A}))$), which is shown in Figure 9(e), does preserve the fact that quiescence may be observed

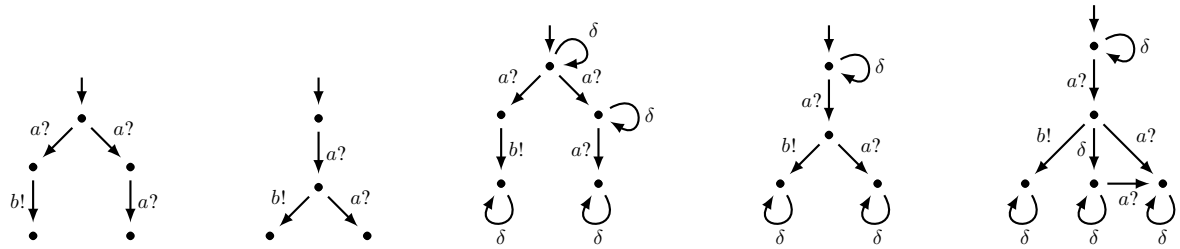


Figure 9: The determinisation and deltafication of IOTS $\mathcal{A}$ do not commute.

after an initial $a?$ input. This shouldn't come as a surprise, since for any IOTS $\mathcal{A}$ the determinisation $det(\mathcal{A})$ is trace equivalent to the original automaton, as was observed earlier. □

Thus, when transforming a nondeterministic IOTS $\mathcal{A}$ to a deterministic QTS, one should take care to first derive $\delta(\mathcal{A})$ and afterwards determinise to obtain $det(\delta(\mathcal{A}))$.

The following results show that deltafication does commute with both action hiding and parallel composition. For action hiding this is trivial. After all, hiding only renames output actions to τ and deltafication only adds δ-loops to states that have no outgoing output transitions, no outgoing τ-transitions and no outgoing δ-transition. Hence, they work on disjoint sets of states; commutativity is therefore immediate.

Theorem 4.9. *Deltafication and action hiding commute, i.e., given an IOTS $\mathcal{A}$ that satisfies condition C1 and rules R2, R3 and R4, and a set of labels $H \subseteq L^{\mathrm{O}}_{\mathcal{A}}$, we have $\delta(hide(\mathcal{A}, H)) \cong hide(\delta(\mathcal{A}), H)$.*

Theorem 4.10. *Deltafication and parallel composition commute, i.e., given two IOTSs $\mathcal{A}$ and $\mathcal{B}$ with $L^{\mathrm{O}}_{\mathcal{A}} \cap L^{\mathrm{O}}_{\mathcal{B}} = \emptyset$ that satisfy condition C1 and rules R2, R3 and R4, we have $\delta(\mathcal{A} \parallel \mathcal{B}) \cong \delta(\mathcal{A}) \parallel \delta(\mathcal{B})$.*

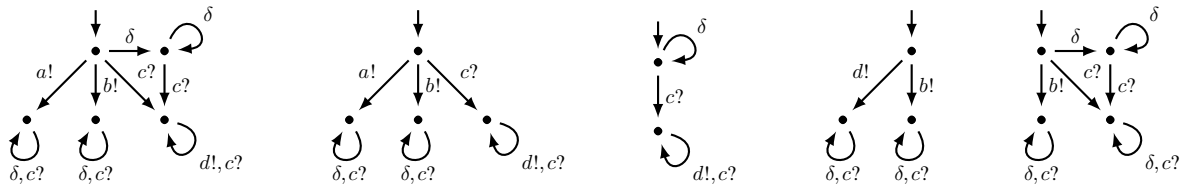
These results are vital, as they allow great modelling flexibility. After all, hiding and parallel composition are often already applied to the IOTSs that describe a specification and its implementation. We now showed that this yields the same QTSs as in case these operations are applied after deltafication.

5 Application to testing

Our main motivation for introducing and studying the QTS model was to enable a clean theoretical framework for model-based testing. In this section, we illustrate how the model can be incorporated in the `ioco` (input-output conformance) testing theory [13].

5.1 A conformance relation based on QTSs

To interpret the results of testing, we need to know which implementations are considered correct. For this, we use a conformance relation, such as `ioco`, that relates specifications to implementations if and only if the latter is 'correct' with respect to the former. For `ioco`, this is the case if the implementation never provides an unexpected output when it is only fed inputs that are allowed according to the specification. In this setting, an unexpected absence of outputs of the implementation is also considered to be unexpected output. This can be formalised very nicely using QTSs, as they already model the expected absence of outputs by explicit δ-transitions.



(a) $\mathcal{A}_{\text{spec}}$ (b) $\mathcal{A}_{\text{impl}_1}$ (c) $\mathcal{A}_{\text{impl}_2}$ (d) $\mathcal{A}_{\text{impl}_3}$ (e) $\mathcal{A}_{\text{impl}_4}$

Figure 10: A specification with two correct and two erroneous implementations.

Definition 5.1. Let $\mathcal{A}_{\text{impl}}, \mathcal{A}_{\text{spec}}$ be QTSs over the same alphabet $L^{\text{O}} \cup L^{\text{I}} \cup \{\delta\}$. Then

$$\mathcal{A}_{\text{impl}} \sqsubseteq_{\texttt{ioco}} \mathcal{A}_{\text{spec}} \text{ if and only if } \forall \sigma \in \mathit{traces}(\mathcal{A}_{\text{spec}}) \,.\, \mathit{out}_{\mathcal{A}_{\text{impl}}}(\sigma) \subseteq \mathit{out}_{\mathcal{A}_{\text{spec}}}(\sigma),$$

where $\mathit{out}_{\mathcal{A}}(\sigma) = \{a! \in L^{\text{O}} \cup \{\delta\} \mid \sigma a! \in \mathit{traces}(\mathcal{A})\}$.

Since we require all QTSs to be input-enabled, it is easy to see that `ioco`-conformance precisely corresponds to traditional trace inclusion over QTSs.

Example 5.2. Consider the specification $\mathcal{A}_{\text{spec}}$ given in Figure 10. It allows the initial state to either be quiescent, output an $a!$ or output a $b!$. We present four implementations. The first two implementations are `ioco`-correct with respect to $\mathcal{A}_{\text{spec}}$: although they omit some of the traces of the specification, they never provide an unexpected output after a trace that is in the specification. The third implementation is erroneous since it can provide a $d!$ output from the initial state, while the specification does not allow this. The fourth implementation is erroneous since it is unexpectedly quiescent after the trace $c?$. □

Note that QTSs allowed us in this example to explicitly model the fact that both quiescence and some output actions are considered correct behaviour of a system. Also, note that the unexpected quiescence of the fourth implementation is clearly marked by a δ-transition in the QTS.

5.2 Testing using QTSs

Using the notion of `ioco`-correspondence, it is quite easy to derive test cases for QTSs. Basically, at each point in time we choose to either try to provide an input, observe the behaviour of the system or stop testing. As long as the trace we obtain in this way (including the δ-actions) is also a trace of the specification, the implementation is correct. Due to the explicit presence of quiescence in the QTS model of the specification, it is easy to see that this straightforward way of testing precisely corresponds to checking `ioco`-conformance.

6 Conclusions and Future Work

We introduced the notion of quiescent transition systems (QTSs), explicitly modelling the absence of outputs as a first-class citizen. We provided four restrictions for QTSs, to eliminate counterintuitive behaviours. Also, we defined the common automaton operations — parallel composition, determinisation and action hiding — directly on QTSs, and showed that all of our restrictions are indeed preserved by the operations. We presented a way to obtain a QTS from a traditional input-output transition system (IOTS), even allowing the situation in which the IOTS already partially models quiescence. Finally,

we illustrated how our novel theory of QTSs can be used to greatly simplify the theory of model-based testing, defining the conformance relation `ioco` in terms of QTSs.

So far, we only allowed input-enabled and convergent QTSs; i.e., systems that cannot perform an endless series of unobservable transitions. Future work will focus on extending our framework to divergent systems that are not necessarily input-enabled. Also, we plan on linking QTSs to timed automata, to explicitly represent δ-transitions as finite timeouts, bridging the gap between formal and practical testing.

Acknowledgements This research has been partially funded by NWO under grants 612.063.817 (SYRUP) and Dn 63-257 (ROCKS).

References

[1] F. Aarts & F. W. Vaandrager (2010): *Learning I/O Automata*. In: *Proc. of the 21th Int. Conf. on Concurrency Theory (CONCUR)*, *LNCS* 6269, Springer, pp. 71–85, doi:10.1007/978-3-642-15375-4_6.

[2] C. Baier & J.-P. Katoen (2008): *Principles of Model Checking*. The MIT Press.

[3] H. C. Bohnenkamp & M. I. A. Stoelinga (2008): *Quantitative testing*. In: *Proc. of the 8th ACM & IEEE Int. Conf. on Embedded software (EMSOFT)*, ACM, pp. 227–236, doi:10.1145/1450058.1450089.

[4] R. De Nicola & R. Segala (1995): *A process algebraic view of input/output automata*. *Theoretical Computer Science* 138, pp. 391–423, doi:10.1016/0304-3975(95)92307-J.

[5] N. A. Lynch & M. R. Tuttle (1987): *Hierarchical Correctness Proofs for Distributed Algorithms*. In: *Proc. of the 6th Annual ACM Symp. on Principles of Distributed Computing (PODC)*, pp. 137–151, doi:10.1145/41840.41852.

[6] R. Segala (1993): *Quiescence, Fairness, Testing, and the Notion of Implementation*. In: *Proc. of 4th Int. Conf. on Concurrency Theory (CONCUR)*, *LNCS* 715, Springer, pp. 324–338, doi:10.1007/3-540-57208-2_23.

[7] R. Segala (1997): *Quiescence, Fairness, Testing, and the Notion of Implementation*. *Information and Computation* 138(2), pp. 194 – 210, doi:10.1006/inco.1997.2652.

[8] W. G. J. Stokkink, M. Timmer & M. I. A. Stoelinga (2012): *Talking quiescence: a rigorous theory that supports parallel composition, action hiding and determinisation (extended version)*. Technical Report TR-CTIT-12-05, CTIT, University of Twente.

[9] T. A. Sudkamp (2006): *Languages and machines*. Pearson Addison Wesley.

[10] M. Timmer, H. Brinksma & M. I. A. Stoelinga (2011): *Model-Based Testing*. In: *Software and Systems Safety: Specification and Verification*, *NATO Science for Peace and Security Series D: Information and Communication Security* 30, IOS Press, Amsterdam, pp. 1–32, doi:10.3233/978-1-60750-711-6-1.

[11] G. J. Tretmans (1996): *Test Generation with Inputs, Outputs, and Quiescence*. In: *Proceedings of the 2nd Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, *LNCS* 1055, Springer, pp. 127–146, doi:10.1007/3-540-61042-1_42.

[12] G. J. Tretmans (1996): *Test Generation with Inputs, Outputs and Repetitive Quiescence*. *Software - Concepts and Tools* 17(3), pp. 103–120.

[13] G. J. Tretmans (2008): *Model Based Testing with Labelled Transition Systems*. In: *Formal Methods and Testing*, *LNCS* 4949, Springer, pp. 1–38, doi:10.1007/978-3-540-78917-8_1.

[14] F. W. Vaandrager (1991): *On the Relationship Between Process Algebra and Input/Output Automata (Extended Abstract)*. In: *Proc. of 6th Annual Symposium on Logic in Computer Science (LICS)*, IEEE, pp. 387–398, doi:10.1109/LICS.1991.151662.

[15] H. M. van der Bijl, A. Rensink & G. J. Tretmans (2004): *Compositional Testing with ioco*. In: *Formal Approaches to Software Testing (FATES)*, *LNCS* 2931, Springer Verlag, Berlin, pp. 86–100, doi:10.1007/978-3-540-24617-6_7.

Towards Symbolic Model-Based Mutation Testing: Combining Reachability and Refinement Checking

Bernhard K. Aichernig Elisabeth Jöbstl
Institute for Software Technology
Graz University of Technology
Graz, Austria
aichernig@ist.tugraz.at joebstl@ist.tugraz.at

Model-based mutation testing uses altered test models to derive test cases that are able to reveal whether a modelled fault has been implemented. This requires conformance checking between the original and the mutated model. This paper presents an approach for symbolic conformance checking of action systems, which are well-suited to specify reactive systems. We also consider non-determinism in our models. Hence, we do not check for equivalence, but for refinement. We encode the transition relation as well as the conformance relation as a constraint satisfaction problem and use a constraint solver in our reachability and refinement checking algorithms. Explicit conformance checking techniques often face state space explosion. First experimental evaluations show that our approach has potential to outperform explicit conformance checkers.

1 Introduction

In most cases, full verification of a piece of software is not feasible. Possible reasons are the increasing complexity of software systems, the lack of highly-educated staff or monetary restrictions. In order to ensure quality and validate system requirements, testing is a viable alternative if it is systematic and automated. Model-based testing fulfills these criteria. The test engineer creates a formal model that describes the expected behaviour of the system under test (SUT). Test cases are then (automatically) derived from this test model by applying different algorithms and test specifications.

One big question is where to get the test specifications from. Our approach is fault-centred, i.e., mutation-based. Classical mutation testing is a method to assess and increase the quality of an existing test suite. The source code of the original program is syntactically altered by applying patterns of typical programming errors, so-called *mutation operators* [14, 15]. The test cases are then executed on the generated *mutants*. If not at least one test case is able to kill a mutant, the test suite has to be improved. Mutation testing relies on two assumptions that have been empirically confirmed: (1) The *competent programmer hypothesis* states that programmers are skilled and do not completely wrong. It assumes that they only make small mistakes. (2) The *coupling effect* states that test cases which are able to detect simple faults (like faults introduced by mutations) are also able to reveal more complex errors.

We employ the mutation concept on the test model instead of the source code and generate test cases that are able to kill the mutated models (*model-based mutation testing*). The generated tests are then run on the SUT and will detect whether a modelled fault has been implemented. So far, much more effort has been spent on the definition of mutation operators and classical mutation testing and not so much work has been done on test case generation from mutations [17].

What we have not mentioned so far: It is possible that a mutant does not show any different behaviour from the original program, although it has been syntactically changed. In this case, the mutant is equivalent to the original and no test case exists that can distinguish the two programs. In general, it is not

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 88–102, doi:10.4204/EPTCS.80.7

decidable whether two programs are equivalent. Hence, mutation testing and its wider application are constrained by the *equivalent mutants problem* [17]. For test case generation, we also have to tackle this problem. Only if the original and the mutated model are not equivalent, we can generate a distinguishing test case. In our case, we do not check for total equivalence, but for refinement. The models we use are *action systems*, which were originally introduced by Back [8]. Action systems are well-suited for modelling reactive systems and allow non-determinism.

Within the European project MOGENTES [1], our group already developed a test case generation tool named *Ulysses*. It is basically an *ioco* checker for action systems and performs an explicit forward search of the state spaces. *ioco* is the input-output conformance relation by Tretmans [22]. Ulysses does not only work for discrete systems, but also supports hybrid action systems via qualitative reasoning techniques [11]. Experiments have shown that the performance of explicit enumeration of the state space involves high memory consumption and runtimes when being applied on complex models. In this paper, we present an alternative approach to determine (non-)refinement between two action systems.

As already shown in [6, 19], constraint satisfaction problems can be used to encode conformance relations and generate test cases. Each of this works dealt with transformational systems, i.e., systems that are started and take some input, process the input by doing some computations and then return an output and stop again. As already mentioned, action systems are well-suited to model reactive systems, i.e., systems that are continuously interacting with their environment. This kind of systems bring up a new aspect: reachability. Hence, the main contribution of this paper is a symbolic approach for refinement checking of reactive systems via constraint solving techniques that avoids state space explosion. We use the predicative semantics of action systems to encode (1) the transition relation and (2) the conformance relation as a constraint satisfaction problem. The constraint system representing the transition relation is used for a reachability analysis like it is known from model checking. For each reached state, we test whether it fulfills the constraint system that represents the conformance relation, which is refinement.

The rest of this paper is structured as follows. The next section presents our running example, a car alarm system. Section 3 gives an overview of the syntax and semantics of action systems and introduces the conformance relation we use. Section 4 explains our approach for finding differences between two action systems. Afterwards, Section 5 presents some experimental data on the application of our implementation on the car alarm system. Subsequently, Section 6 deals with restrictions and mentions some of our plans for future work. Finally, Section 7 discusses related work and concludes the paper.

2 Running Example

In order to demonstrate the basic concepts of our approach, we use a simplified version of a car alarm system (CAS). The example is taken from Ford's automotive demonstrator within the MOGENTES project. The following requirements were specified and served as the basis for our model:

R1 - Arming. The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment, and all doors are closed.

R2 - Alarm. The alarm sounds for 30 seconds if an unauthorized person opens the door, the luggage compartment, or the bonnet. The hazard flasher lights will flash for five minutes.

R3 - Deactivation. The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

[1] `http://www.mogentes.eu`

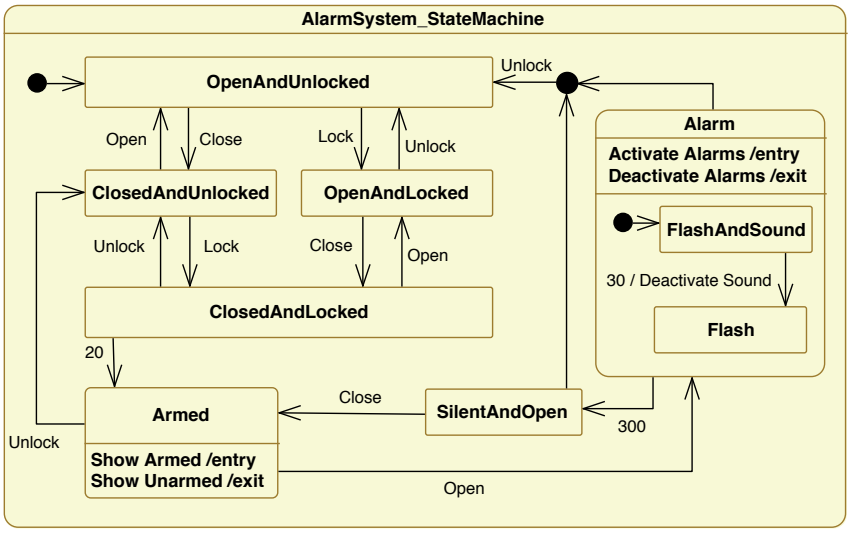


Figure 1: UML state machine of the car alarm system

Figure 1 shows a UML state machine of our CAS. From the state *OpenAndUnlocked* one can traverse to *ClosedAndLocked* by closing all doors and locking the car. As specified in requirement R1, the alarm system is armed after 20 seconds in *ClosedAndLocked*. Upon entry of the *Armed* state, the model calls the method *AlarmArmed.SetOn*. Upon leaving the state, which can be done by either unlocking the car or opening a door, *AlarmArmed.SetOff* is called. Similarly, when entering the *Alarm* state, the optical and acoustic alarms are enabled. When leaving the alarm state, either via a timeout or via unlocking the car, both acoustic and optical alarm are turned off. Note that the order of these two events is not specified, neither for enabling nor for disabling the alarms. Hence the system is not deterministic. When leaving the alarm state after a timeout (cf. requirement R2) the system returns to an armed state only in case it receives a close signal. Turning off the acoustic alarm after 30 seconds, as specified in requirement R2, is reflected in the time-triggered transition leading to the *Flash* sub-state of the *Alarm* state.

3 Preliminaries

3.1 Action Systems

Action systems [8] are a kind of guarded-command language for modelling concurrent reactive systems. They have a formal semantics with refinement laws and are compositional [9]. Many extensions exist, but the main idea is that a system state is updated by guarded actions that may be enabled or not. If no action is enabled, the action system terminates. If several actions are enabled, one is chosen non-deterministically. Hence, concurrency is modelled in an interleaving semantics. The formal method *B* has recently adopted the action-system style in the form of *Event-B* [2].

Example 3.1. Our action systems are written in Prolog syntax. Listing 1 shows code snippets from the action system model of the CAS as described in Section 2. The first two lines contain user-defined types. All types are basically integers, but their ranges can be restricted. In Line 1, a type with name *enum_State* is defined. Its domain begins with 0 and ends with 7. Line 4 declares a variable with name *aState* which is of type *enum_State*. Line 6 defines the list of variables that make up the state of the action system.

Listing 1: Code snippet from the action system model for the car alarm system

```
1  type(enum_State, X) :- X in 0..7.
2  type(int, X) :- X in 0..270.
3  ...
4  var([aState], enum_State).
5  ...
6  state_def([aState, fromAlarm, fromArmed, ..., flashOn, soundOn]).
7
8  init([6, 0, 0, 0, 0, 0]).
9
10 as :-
11     actions (
12     'after'(Wait_time)::(true) =>
13     (
14        ((Wait_time #= 20 #/\ aState #= 3) =>
15             (aState := 2; fromClosedAndLocked_OR_fromSilentAndOpen := 1))
16       []
17        ((Wait_time #= 30 #/\ aState #= 1 #/\ fromArmed #= 4) =>
18             (aState := 0; fromAlarm := 4; fromArmed := 0))
19       []
20        ((Wait_time #= 270 #/\ aState #= 0 #/\ fromAlarm #= 2) =>
21             (aState := 7; fromAlarm := 1; fromArmed := 0))
22     ),
23     'Lock'::(true) =>
24     (
25        ((aState #= 6 #/\ fromAlarm #= 0) => (aState := 5))
26       []
27        ((aState #= 4 #/\ fromArmed #\= 1) => (aState := 3; fromArmed := 0))
28     ),
29 ...
30     ),
31   dood (
32       'Lock'
33   [] [X:int]:'after'(X)
34   [] ...
35   ).
```

The *init* predicate in Line 8 defines the initial values for the state. At Line 10, the actual action system begins. It consists of an *actions* block (Lines 11 to 30) and an *do-od* block (Lines 31 to 35).

The *actions* block defines named actions. Each action consists of a name, a guard and a body (*name* :: *guard* => *body*) (cf. Lines 23 to 28). Actions may also have parameters, like action *after* in Line 12. The operator [] denotes non-deterministic choice. We use it in our example together with guards to distinguish between different cases in which an action may fire. Consider for example Lines 14 and 15. The action *after(20)* may fire if the action system is in a state where variable *aState* equals 3, which corresponds to state "ClosedAndLocked" in the CAS state chart (Figure 1). The action system then assigns variable *aState* value 2 and variable *fromClosedAndLocked OR fromSilentAndOpen* value 1, which corresponds to the state "Armed" in the state chart. The do-od block connects previously defined actions via non-deterministic choice. Basically, the execution of an action system is a continuous iteration over the do-od block. Here, there is always at least one action enabled. Hence, the car alarm system never terminates, but continuously waits for stimuli.

$$
\begin{array}{lll}
M ::= D\ \texttt{as :- actions}(\overline{A}),\ \texttt{dood}(P). & & P ::= E \mid E\ [\!]\ P \\
D ::= \overline{\texttt{type}(t,X)\ \texttt{:-}\ X\ \texttt{in}\ n_1..n_2.}\ \overline{\texttt{var}([\overline{v}],t).}\ \texttt{state\_def}([\overline{v}]).\ \texttt{init}([\overline{c}]). & & E ::= l \mid [\overline{X:t}]l(\overline{X}) \\
A ::= L :: g => B & & L ::= l \mid l(\overline{X}) \\
B ::= v := e \mid g => B \mid B;B \mid B\ [\!]\ B & & e ::= v \mid c \mid e+e \mid \ldots
\end{array}
$$

Figure 2: Syntax of a subset of action systems

$$
\begin{array}{llll}
l :: g => B & =_{df} & g \wedge B \wedge tr' = tr^\frown[l] & l(\overline{X}) :: g => B \quad =_{df} \quad \exists\, \overline{X} : g \wedge B \wedge tr' = tr^\frown[l(\overline{X})] \\
x := e & =_{df} & x' = e \wedge y' = y \wedge \ldots \wedge z' = z & g => B \quad =_{df} \quad g \wedge B \\
B(\overline{v},\overline{v}');B(\overline{v},\overline{v}') & =_{df} & \exists\, \overline{v_0} : B(\overline{v},\overline{v_0}) \wedge B(\overline{v_0},\overline{v}') & B\ [\!]\ B \quad =_{df} \quad B \vee B
\end{array}
$$

Figure 3: Predicative semantics of actions

Syntax. In the literature many versions of Back's original action-system notation [8] exist. The syntax used in this work is presented in Figure 2. Our syntax contains some elements of Prolog, because the tool is implemented in SICStus Prolog. Here, an action system model M comprises the basic definitions D, a set of action definitions $\overline{A}$ and the do-od block P. In the basic definitions we define the types t, declare variables v of type t, define the system state-space as variable vector $\overline{v}$ and finally provide the initial state as vector of constants $\overline{c}$. An action A is a labelled guarded command with label L, guard g and body B. Actions may have a list of parameters $\overline{X}$. The body of an action may assign an expression e to a variable v or it may be composed of (nested) guarded commands itself. Composition may be sequential or non-deterministic choice. The do-od block P provides the event-based view on the action system. Here, the actions are composed by their action labels l. Currently, we only support non-deterministic choice in the do-od block, but in future sequential and prioritized composition will be added.

Semantics. The formal semantics of action systems is usually defined in terms of weakest preconditions. However, for our constraint-based approach, we found a relational predicative semantics being more suitable. We follow the style of He and Hoare's Unifying Theories of Programming [16]. Figure 3 presents the formal semantics of the actions of our modelling language. The state-changes of actions are defined via predicates relating the pre-state of variables $\overline{v}$ and their post-state $\overline{v}'$. Furthermore, the labels form a visible trace of events tr that is updated to tr' whenever an action runs through. Hence, a guarded action's transition relation is defined as the conjunction of its guard g, the body of the action B and the adding of the action label l to the previously observed trace. In case of parameters $\overline{X}$, these are added as local variables to the predicate. An assignment updates one variable x with the value of an expression e and leaves the rest unchanged. Sequential composition is standard: there must exist an intermediate state $\overline{v_0}$ that can be reached from the first body predicate and from which the second body predicate can lead to its final state. Finally, non-deterministic choice is defined as disjunction. The semantics of the do-od block is as follows: while actions are enabled in the current state, one of the enabled actions is chosen non-deterministically and executed. An action is enabled in a state if it can run through, i.e. if a post-state exists such that the semantic predicate can be satisfied. The action system terminates if no action is enabled. The labelling of actions is non-standard and has been added in order to support an event-view for testing.

3.2 Conformance

Once the modelling language with a precise semantics is fixed, we can define what it means that a SUT conforms to a given reference model, i.e. if the observations of a SUT confirm the theory induced by a formal model. This relation between a model and the SUT is called the conformance relation.

In model-based mutation testing, the conformance relation plays an additional role. It defines if a syntactic change in a mutant represents an observable fault, i.e. if a mutant is equivalent or not. However, for non-deterministic models an equivalence relation is no suitable conformance relation. An abstract non-deterministic model may do more than its concrete counterpart. Hence, useful conformance relations are order-relations rather than equivalence relations, the order going from abstract to more concrete models. In this work, we have chosen UTP's refinement relation as a conformance relation. UTP defines refinement via implication, i.e. more concrete implementations I imply more abstract models M.

Definition 3.1. ***(Refinement)***

$$M \sqsubseteq I \;=_{df}\; \forall x, x' y, y', \cdots \in \alpha \;:\; I \Rightarrow M \qquad \textit{for all } M, I \textit{ with alphabet } \alpha.$$

The alphabet α is the set of variables denoting observations.

In [4] we have developed a mutation testing theory based on this notion of refinement. The key idea is to find test cases whenever a mutated model M^M does not refine an original model M^O, i.e. if $M^O \not\sqsubseteq M^M$. Hence, we are interested in counter-examples to refinement. From Definition 3.1 follows that such counter-examples exist if and only if implication does not hold:

$$\exists x, x', y, y', \cdots \in \alpha \;:\; M^M \wedge \neg M^O$$

This formula expresses that there are observations in the mutant M^M that are not allowed by the original model M^O. We call a state, i.e. a valuation of all variables, *unsafe* if such an observation can be made.

Definition 3.2. ***(Unsafe State)*** *A pre-state u is called unsafe if it shows wrong (not conforming) behaviour in a mutated model M^M with respect to an original model M^O. Formally, we have:*

$$u \in \{s \mid \exists\, s' : M^M(s, s') \wedge \neg M^O(s, s')\}$$

We see that an unsafe state can lead to an incorrect next state. In model-based mutation testing, we are interested in generating test cases that cover such unsafe states. Hence, our fault-based testing criteria are based on the notion of unsafe states. How to search for unsafe states in action systems efficiently is discussed in the next section.

4 Searching Unsafe States

Figure 4 gives an overview of our approach to find an unsafe state. The inputs are the original action system model AS^O and a mutated version AS^M. Each action system consists of a set of actions AS^O_i and AS^M_j respectively, which are connected via non-deterministic choice. The first step is a preprocessing activity to check for refinement quickly. It is depicted on the left-hand side of Figure 4 as box *find mutated action*. If there does not exist an unsafe state at this point, we cannot find any mutated action that yields non-conformance. Hence, we already know that the action systems are equivalent. If we find an unsafe state in this phase, we cannot be sure that it is reachable from the initial state of the action system. But we know which action has been mutated and are able to construct a *non-refinement constraint*, which describes the set of all unsafe states. The next step performs a reachability analysis and uses the non-refinement constraint to test each reached state whether it is an unsafe state. In the following, we give more details.

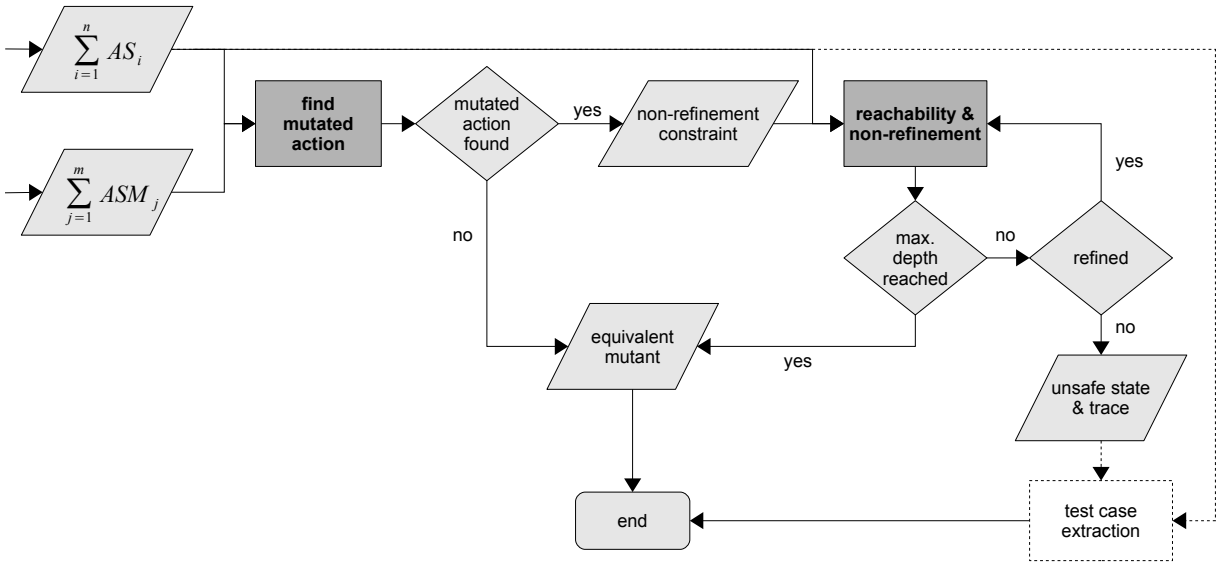


Figure 4: Process for finding an unsafe state

4.1 Non-Refinement of Action Systems

In the previous section, we have introduced non-refinement as a general criterion for identifying unsafe states. Now, we are going to concentrate on the special case of action systems.

The observations in our action system language are the event-traces and the system states before $(\overline{v}, tr)$ and after one execution $(\overline{v}', tr')$ of the do-od block. Then, a mutated action system AS^M refines its original version AS^O if and only if all observations possible in the mutant are allowed by the original. Hence, our notion of refinement is based on both, event traces and states. However, in an action system not all states are reachable from the initial state. Therefore, reachability has to be taken into account.

We reduce the general refinement problem of action systems to a step-wise simulation problem only considering the execution of the do-od block from reachable states:

Definition 4.1. ***(Refinement of Action Systems)*** *Let AS^O and AS^M be two action systems with corresponding do-od blocks P^O and P^M. Furthermore, we assume a function "reachable" that returns the set of reachable states for a given trace in an action system. Then*

$$AS^O \sqsubseteq AS^M =_{df} \forall \overline{v}, \overline{v}', tr, tr' : ((\overline{v} \in reachable(AS^O, tr) \wedge P^M) \Rightarrow P^O) \quad .$$

This definition is different to Back's original refinement definition based on state traces[9]. Here, also the possible event traces are taken into account. Hence, also the action labels have to be refined.

Negating this refinement definition and considering the fact that the do-od block is a non-deterministic choice of actions A_i leads to the non-refinement condition for two action systems:

$$\exists \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reachable(AS^O, tr) \wedge (A_1^M \vee \cdots \vee A_n^M) \wedge \neg A_1^O \wedge \cdots \wedge \neg A_m^O)$$

By applying the distributive law, we bring the disjunction outwards and obtain a set of constraints for detecting non-refinement.

Algorithm 1 *findMutatedAction*$(AS^O, AS^M) : (AS_i^M, CS\_nonrefine)$

1: $CS\_AS^O := trans(AS^O)$
2: **for all** $A_i^M \in AS^M$ **do**
3: $\quad CS\_AS_i^M := trans(A_i^M)$
4: $\quad CS\_nonrefine := CS\_AS_i^M \wedge \neg CS\_AS^O$
5: $\quad$ **if** $sat(CS\_nonrefine)$ **then**
6: $\quad\quad$ **return** $(A_i^M, CS\_nonrefine)$ *// mutated action found*
7: $\quad$ **end if**
8: **end for**
9: **return** $(nil, false)$ *// equiv*

Theorem 4.1. ***(Non-refinement)*** *A mutated action system AS^M does not refine its original AS^O, iff any action A_i^M of the mutant shows trace or state-behaviour that is not possible in the original action system:*

$$AS^O \not\sqsubseteq AS^M \quad \textit{iff} \quad \bigvee_{i=1}^{n} \exists \overline{v}, \overline{v}', tr, tr' \; : \; (\overline{v} \in reachable(AS^O, tr) \wedge A_i^M \wedge \neg A_1^O \wedge \cdots \wedge \neg A_m^O)$$

In the following, we discuss how this property is applied in our refinement checking process.

4.2 Finding a Mutated Action

The non-refinement condition presented in Theorem 4.1 is a disjunction of constraints of which each deals with one action A_i^M of the mutated action system AS^M. Hence, it is sufficient to satisfy one of these sub-constraints in order to find non-conformance. We use this for our implementation as we perform the non-refinement check action by action. Here, we first concentrate on finding a possibly unreachable unsafe state. Reachability is dealt with separately (see Section 4.3).

Algorithm 1 gives details on the action-wise non-refinement check, which is depicted on the left-hand side of Figure 4 (box *find mutated action*). We transform the whole do-od block of the original into a constraint system according to our predicative semantics of action systems (Line 1). We then translate one action of the mutated action system into a constraint system (Line 3). The non-refinement constraint *CS_nonrefine* is the conjunction of the constraint system representing the mutated action ($CS\_AS_i^M$) and the negated constraint system representing the original action system ($\neg CS\_AS^O$, cf. Line 4). Note that sequential composition involves existential quantification, which becomes universal quantification due to negation. Existential quantification is implicit in constraint systems. Universal quantification would lead to quantified constraint satisfaction problems (QCSPs) that are not supported by common constraint solvers. Fortunately, we can resolve this problem by a normal form that requires that non-deterministic choice is always the outermost operator and not allowed in nested expressions. In this way, the left-hand side of a sequential composition is always deterministic and existential quantification can be eliminated. Our car alarm system example (cf. Listing 1) already satisfies this normal form. Otherwise, each action system can be automatically rewritten to this normal form. This has not yet been implemented.

The non-refinement constraint for the just translated action is then given to a constraint solver to check whether it is satisfiable by any $\overline{v}, \overline{v}', tr, tr'$(Line 5), i.e., whether there exists an unsafe state $\overline{v}$ for AS^M and AS^O. If yes, we found the mutated action and return it together with the according non-refinement constraint *CS_nonrefine*. Otherwise, the next action A_i^M is investigated (loop in Line 2). If no action leads to a satisfiable non-refinement constraint, then AS^M refines AS^O (Line 9). Algorithm 1

is sound for first order mutants (one syntactical change per mutant). It aborts after finding the first action that leads to an unsafe state. Note that we do not know yet whether an unsafe state is actually reachable. For higher-order mutants (more than one syntactical change per mutant) it could happen that our algorithm finds a mutated action for which no unsafe state is reachable. In this case, it is necessary to go back and search for another mutated action until an unsafe state is actually reachable or all actions are processed.

Identifying the mutated action is important for our performance for two reasons: (1) Solving the non-refinement constraint *CS_nonrefine* for one action is by far faster than solving a non-refinement constraint encoding all actions of the mutated action system at once. Experiments showed that the latter is impractical with the currently used constraint solver. (2) By knowing which action has been mutated, we know which non-conformance constraint has to be fulfilled by an unsafe state. This saves constraint solver calls during the reachability analysis, which is presented in the following.

4.3 Reaching an Unsafe State

Now we know whether there exists any unsafe state. If this is the case, we also know which action has been mutated and we have determined a non-refinement constraint that describes the set of all possible unsafe states. But we do not know yet, whether an unsafe state is actually reachable from a given initial state. It is possible that an unsafe state exists theoretically and has been found in the previous step, but that no unsafe state is reachable from the initial state of the system. In this case, the mutated action system conforms to the original, i.e., the mutant refines the specification. To find out whether an unsafe state is actually reachable, we perform a state space exploration of the original action system *AS*. During this reachability analysis, each encountered state is examined if it is an unsafe state. This test is realized via a constraint solver that checks whether the reached state fulfills our non-refinement constraint (see right-hand side of Figure 4).

The pseudo-code in Algorithm 2 gives more details on our combined reachability and non-refinement check. The algorithm requires the following inputs: (1) the original action system AS^O, (2) the constraint system *CS_nonrefine* representing the non-refinement constraint obtained from Algorithm 1, (3) an integer *max* restricting the search depth, and (4) the initial state *init* of the action system AS^O. The algorithm returns a pair consisting of the found unsafe state and the trace leading there.

At first (Lines 1 to 3), we check whether the initial state is already an unsafe state. This is, we call the constraint solver with the non-refinement constraint and set the input state to be the initial state of *AS*. If the solver finds an action *a* leading to a post-state *s* then we detected non-conformance. We found either a state that can be reached from *init* only in the mutant but not in the original or an action that is enabled at state *init* only in the mutant but not in the original. In this case, *init* is returned as unsafe state together with the empty trace. Otherwise, we perform a breadth-first search (Lines 4 to 19) starting at *init*. The queue *ToExplore* holds the states that have been reached so far and still have to be further explored. It contains pairs consisting of the state and the shortest trace leading to this state. The set *Visited* holds all states that have been reached so far and is maintained to avoid the re-exploration of states. To ensure termination, the state space is only explored up to a user-defined depth *max* (Line 9).

The function *succStateAndAction*(s_0) (Line 10) returns the set of all successors of state s_0. Each successor is a pair consisting of the successor state s_1 and the action a_1 leading from s_0 to s_1. The successors are calculated via the predicative semantics of our action systems (cf. Section 3.1). Thereby, we gain a constraint system representing the transition relation of our action system. It describes one iteration of the do-od block. The interesting variables in the constraint system are the input state variables, the action variable, and the post-state variables. The input state variables are set to be equal to the variables in s_0.

Algorithm 2 $reachNonRefine(AS^O, CS\_nonrefine, max, init) : (unsafe, trace)$

```
1:  if ∃a,s : CS_nonrefine(init,a,s) then
2:     return (init,[])
3:  end if
4:  Visited := {init}
5:  ToExplore := enqueue((init,[]),[])
6:  while ToExplore ≠ [] do
7:     (s0,tr_s0) := head(ToExplore)
8:     ToExplore := dequeue(ToExplore)
9:     if length(tr_s0) < max then
10:       for all (s1,a1) ∈ succStateAndAction(s0) : s1 ∉ Visited do
11:          tr_s1 := add(tr_s0,a1)
12:          if ∃a2,s2 : CS_nonrefine(s1,a2,s2) then
13:             return (s1,tr_s1)      // unsafe state
14:          end if
15:          Visited := add(s1,Visited)
16:          ToExplore := enqueue((s1,tr_s1),ToExplore)
17:       end for
18:    end if
19: end while
20: return (nil,[])      // equiv
```

We then use a constraint solver to set the action variable a_1 and the variables that make up the post-state s_1. By calling the constraint solver multiple times with an extended constraint system (with the added restriction that the next solution has to be different fromt the previous ones), we get all transitions that are possible from s_0.

Each state s_1 that is reached in this way and has not yet been processed ($s_1 \notin Visited$) is checked for being an unsafe state (Line 12). This works analogously to Line 1. If an unsafe state is found it is returned together with the trace leading there (Line 13). Otherwise, the state is included in the set of visited states (Line 15) and enqueued for further exploration (Line 16). If no unsafe state is found up to depth *max*, the mutant refines the original action system and we return the pair $(nil, [])$ as a result (Line 20).

4.4 Test Case Extraction

We implemented our technique in SICStus Prolog[2] (version 4.1.2). SICStus comes with an integrated constraint solver *clpfd* (Constraint Logic Programming over Finite Domains) [13], which we used. Our implementation results either in the verdict *equiv*, which means that the mutated action system conforms to the original, or in an unsafe state and a sequence of actions leading to this state. In the latter case it is possible to generate a test case. The trace resulting from our approach is not yet a test case, although it reaches the unsafe state. We still need to add verdicts (pass, fail, and inconclusive) where necessary. Additionally, the trace has to be at least one step longer in order to check that only correct behaviour occurs after the unsafe state. A test case generated in this way is able to reveal whether the model mutant has been implemented. This test case extraction step has not yet been implemented and remains future

[2] `http://www.sics.se/sicstus/`

work. It is indicated by the dotted parts at the right bottom of Figure 4. For an explicit *ioco* checking technique, we have suggested different test case extraction strategies in [3].

5 Empirical Results

For an empirical evaluation of our prototypical implementation, we have modelled the car alarm system (CAS) described in Section 2 as an action system. Some code snippets of the model have already been presented in Listing 1. Additionally, we have manually created first order mutants (one mutation per mutant) for the original CAS model. We applied the following three mutation operators:

- *guard true*: Setting all possible guards to true resulted in 34 mutants.
- *comparison operator inversion*: The action system contains two comparison operators: equality (`#=`) and inequality (`#\=`). Inverting all possible equality operators (resulting in inequality) yielded 52 mutants. Substituting inequality by equality operators resulted in 4 mutants.
- *increment integer constant*: Incrementation of all integer constants by 1 resulted in 116 mutants. Note that at the upper bound of a domain, we took the smallest possible value in order to avoid domain violations.

From these mutation operators, we obtained a total of 206 mutated action systems. Additionally, we also included the original action system as an equivalent mutant. Unfortunately, the currently used constraint solver was not able to handle 12 of the 207 mutants within a reasonable amount of time during refinement checking without reachability (see Section 4.2). We will try another constraint solver and see if the performance increases. For now we had to exclude the 12 mutants from our experiments.

We ran our experiments on a machine with a dual-core processor (2.8 GHz) and 8 GB RAM with a 64-bit operating system. Table 1 gives information about the execution times of our *refinement checker* prototype for the remaining 195 mutations. All values are given in seconds unless otherwise noted. We conducted our experiments for four different versions of the CAS: (1) **CAS_1**: the CAS as presented in Section 2 with parameter values 20, 30, and 270 for the action *after*, (2) **CAS_10**: the CAS with parameter values multiplied by 10 (200, 300, and 2700), (3) **CAS_100**: the CAS with parameters multiplied by 100, and (4) **CAS_1000**: the CAS with parameters multiplied by 1000. These extended parameter ranges shall test the capabilities of our symbolic approach. The column *find mutated action* shows that checking whether there possibly exists an unsafe state and which action has been mutated (see Section 4.2) is quite fast. The reachability and non-refinement check (column *reach & non-refine*, see Section 4.3) needs the bigger part of the overall execution time (column *total*). The four versions of the CAS differ only in the parameter values and the domains for the parameters. Our approach takes almost the same amount of time for all four versions: approximately $1\frac{3}{4}$ minutes to process all 195 mutants, on average half a second per mutant, a minimum time per mutant of 0.03 seconds, and a maximum of about 3 seconds for one mutant.

To have at least a weak reference point for our performance, we have also utilized our explicit *ioco* checker *Ulysses* [3, 11] to generate tests for the CAS. We have to admit that this comparison is not totally fair, since Ulysses works quite differently: First of all, Ulysses uses a different conformance relation named *ioco* (input-output conformance for labelled transition systems, see [22]). We ran Ulysses in two settings. First, on the CAS with distinguished input and output actions. The input actions were Close, Open, Lock, and Unlock. The remaining actions were classified as outputs. Second, we classified all actions of the CAS as outputs. This setting is closer to our notion of conformance, since in refinement we do not distinguish between input and output actions. Nevertheless, the conformance relations are still

CAS version		refinement checker			Ulysses	
		find mutated action	reach & non-refine	total	in/out	out
CAS_1	total	16	90	106	98	65
	average	0.08	0.46	0.54	0.50	0.34
	min.	0.01	0.02	0.03	0.05	0.05
	max.	0.30	2.80	3.10	6.30	5.33
CAS_10	total	15	86	101	8.8 h	7.9 h
	average	0.08	0.44	0.52	2.7 min	2.4 min
	min.	0.01	0.02	0.03	0.45	0.36
	max.	0.27	2.80	3.07	2.6 h	2.6 h
CAS_100	total	16	90	106	-	-
	average	0.08	0.46	0.54	-	-
	min.	0.01	0.02	0.03	-	-
	max.	0.27	2.77	3.04	-	-
CAS_1000	total	15	85	100	-	-
	average	0.08	0.44	0.52	-	-
	min.	0.01	0.02	0.03	-	-
	max.	0.27	2.69	2.96	-	-

Table 1: Execution times for our refinement checking tool and the *ioco* checker Ulysses applied on four versions of the car alarm system. All values are given in seconds unless otherwise noted.

not identical. In refinement, we only check that an implementation does not show unspecified behaviour. Hence, an implementation can always do less than specified. In *ioco*, abscence of (output) behaviour has to be explicitly permitted by the specification model. Another difference between Ulysses and our approach are the final results. Ulysses generates adaptive test cases, not only a trace leading to an unsafe state as our tool does (cf. Section 4.4).

Despite these inconsistencies, the comparison with Ulysses still demonstrates one thing very clearly: the problems with explicit state space exploration. Ulysses explicitly enumerates all symbolic values (like parameters in the CAS example). Table 1 also gives the execution times for Ulysses on the CAS with our two settings: (1) distinction between inputs and outputs (column *in/out*) and (2) every action is an output (column *out*). For the original CAS version (**CAS_1**), Ulysses is faster than our constraint-based approach, particularly if every action is an output. In this case, test case generation with Ulysses took only one minute for all 195 mutants. But when it comes to **CAS_10** with larger parameter values (200, 300, and 2700 instead of 20, 30, and 270) Ulysses runs into massive problems. The execution time drastically increases to almost 9 hours (in/out) and about 8 hours (out). On average, each mutant takes 2.7 to 2.4 minutes. One mutant even caused a runtime of 2.6 hours. We observed a memory usage of up to 6 GB RAM. We suspect that a significant amount of the execution time is spent on swapping. For the CAS versions **CAS_100** and **CAS_1000**, we did not run Ulysses as the runtimes would be even higher.

Already for the original CAS (**CAS_1**), Ulysses needs 5 to 6 seconds for some mutants that altered the *after* action that has one parameter: the time to wait with a range from 0 to 270. Our approach took only 0.1 seconds to find the unsafe state and the corresponding trace. Hence, Ulysses shows very good performance for systems with small domains. When it comes to larger ranges of integers, Ulysses comes to its limits quite soon. In this cases, our approach represents a viable alternative.

6 Restrictions and Further Optimizations

Although our approach shows great promise for solving the problems with large variable domains, it is far from being perfect. In the following, we discuss restrictions and possible optimizations of the overall approach as well as of our current implementation: More elaborate *conformance relations* are possible. In [23] we presented a predicative semantics for ioco. Alternating simulation is also an option.

As already discussed in Section 4.4, our approach currently results in an unsafe state and a trace leading there. The generation of *adaptive test cases* remains future work. Our action systems are ignorant of *time*. In the CAS the waiting time was modelled as a simple parameter. For more elaborate models with clocks a tick-action modelling the progress of time is needed. For a full timed-automata model, the actions could be extended with deadlines similar to [10].

One obvious improvement for our *implementation* is the use of more efficient data structures. Currently, we use lists in most cases as they are the most common data structure in Prolog. For example, the set of visited states in Algorithm 2 is currently represented by a list. The use of ordered sets in combination with hash values would be reasonable. As already mentioned, we implemented our approach in SICStus Prolog. It comes with a built-in constraint solver (*clpfd* - Constraint Logic Programming over Finite Domains [13]), which we use so far. Our next steps will include a comparison with other constraint solvers, e.g., Minion[3]. Additionally, we already supervise an ongoing diploma thesis on the use of different SMT solvers like Yices[4] or Z3[5].

7 Conclusion

This paper deals with model-based mutation testing. Like in classical model-based testing, we have a test model describing the expected behaviour of a system under test. This model is mutated by applying syntactical changes. We then generate test cases that are able to reveal whether a software system has implemented the modelled faults. We have chosen action systems as a formalism for system modelling. In this paper, we presented our syntax and a predicative semantics for action systems. We also explained refinement in the context of action systems. Most importantly, we have developed and implemented an approach for refinement checking of action systems as a first step for test case generation from mutated action systems. Throughout the whole paper, a car alarm system served as a running example, which was not only used for illustration but also served as a case study for our experiments.

We employ constraint satisfaction techniques that have already been used previously [6, 19] to encode conformance relations and generate test cases. Nevertheless, prior works dealt with systems that take an input and deliver some output. This paper deals with refinement checking of reactive systems. The thereby introduced continuous interaction with the environment brings up a new aspect: reachability. Hence, the main contribution of this paper is a symbolic approach for refinement checking of reactive systems via constraint solving techniques that avoids state space explosion, which is often a problem with explicit techniques.

Our approach to detect non-refinement in action systems is basically a combination of reachability and refinement checking. We use the predicative semantics of action systems to encode (1) the transition relation and (2) the conformance relation as a constraint satisfaction problem. During reachability analysis, the constraint system representing the transition relation is used for finding successor states. The

[3] `http://minion.sourceforge.net`

[4] `http://yices.csl.sri.com/`

[5] `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

constraint system encoding refinement enables us to test each reached state whether it is an unsafe state, i.e., whether this state is directly followed by observations in the mutant that must not occur at this state according to the original model.

Experimental results with an action system modelling a car alarm system have demonstrated the potential of our approach compared to explicit conformance checking techniques. We conducted experiments with four different versions of the car alarm system that only differ in the integer ranges of the parameters. The smallest model deals with parameters from 0 to 270, the largest model contains integer parameters from 0 to 270000. Our implementation provides constant runtime for all four models. For 195 mutated models, we only need about $1^3/_4$ minutes regardless of the parameter ranges. The explicit conformance checker that we also applied on two model versions was faster (1 to $1^1/_2$ minutes) for the smallest model, but already the next larger model caused an execution time of about 8 hours.

There is existing literature on model-based mutation testing. One of the first models to be mutated were predicate-calculus specifications [12] and formal Z specifications [21]. Later on, model checkers were available to check temporal formulae expressing equivalence between original and mutated models. In case of non-equivalence, this leads to counterexamples that serve as test cases [7]. This is very similar to our approach, but in contrast to this state-based equivalence test, we check for refinement allowing non-deterministic models. Another conformance relation capable to deal with non-determinism is the input-output conformance (*ioco*) of Tretmans [22]. The first use of an ioco checker for mutation testing was on LOTOS specifications [5]. The tool *Ulysses* that was already mentioned in Section 5 applies ioco checking for mutation-based test case generation on qualitative action systems [11]. A further conformance relation supporting non-determinism is FDR (Failures-Divergence Refinement) for the CSP process algebra [1]. The corresponding FDR model checker/refinement checker has been used in [20] to set up a whole testing theory in terms of CSP. This work allows test case generation via test purposes, but not by model mutation.

Our own past work has shown that typically there is no silver bullet in automatic test case generation that is able to deal with every system efficiently [18]. As we only used one exemplary model for evaluating our approach so far, it is too early to say whether the performance of our approach may be generalized. Future work will include more experiments with different types of systems to find this out.

Acknowledgment. Research herein was funded by the Austrian Research Promotion Agency (FFG), program line "Trust in IT Systems", project number 829583, TRUst via Failed FALsification of Complex Dependable Systems Using Automated Test Case Generation through Model Mutation (TRUFAL).

References

[1] A. W. Roscoe (1994): *Model-checking CSP*, chapter 21. Prentice-Hall. Available at `http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/50.ps`.

[2] Jean-Raymond Abrial (2010): *Modelling in Event-B: System and software design.* Cambridge University Press.

[3] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl & Willibald Krenn (2011): *Efficient Mutation Killers in Action*. In: *IEEE 4th Int. Conf. on Software Testing, Verification and Validation, ICST 2011*, IEEE Computer Society, pp. 120–129. Available at `http://dx.doi.org/10.1109/ICST.2011.57`.

[4] Bernhard K. Aichernig & Jifeng He (2009): *Mutation testing in UTP*. *Formal Aspects of Computing* 21(1-2), pp. 33–64, doi:10.1007/s00165-008-0083-6.

[5] Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer & Franz Wotawa (2007): *Protocol Conformance Testing a SIP Registrar: An Industrial Application of Formal Methods*. In: *5th IEEE Int. Conf. on*

Software Engineering and Formal Methods, SEFM 2007, IEEE Computer Society, pp. 215–224. Available at http://doi.ieeecomputersociety.org/10.1109/SEFM.2007.31.

[6] Bernhard K. Aichernig & Percy Antonio Pari Salas (2005): *Test Case Generation by OCL Mutation and Constraint Solving*. In: *5th Int. Conf. on Quality Software, QSIC 2005*, IEEE Computer Society, pp. 64–71. Available at http://doi.ieeecomputersociety.org/10.1109/QSIC.2005.63.

[7] Paul Ammann, Paul E. Black & William Majurski (1998): *Using Model Checking to Generate Tests from Specifications*. In: *2nd IEEE Int. Conf. on Formal Engineering Methods, ICFEM 1998*, IEEE Computer Society, pp. 46–54. Available at http://computer.org/proceedings/icfem/9198/91980046abs.htm.

[8] Ralph-Johan Back & Reino Kurki-Suonio (1983): *Decentralization of Process Nets with Centralized Control*. In: *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, ACM, pp. 131–142.

[9] Ralph-Johan Back & Kaisa Sere (1991): *Stepwise Refinement of Action Systems*. *Structured Programming* 12, pp. 17–30.

[10] Sébastien Bornot, Joseph Sifakis & Stavros Tripakis (1998): *Modeling Urgency in Timed Systems*. In: *Compositionality: The Significant Difference (COMPOS'97)*, *LNCS* 1536, Springer, pp. 103–129.

[11] Harald Brandl, Martin Weiglhofer & Bernhard K. Aichernig (2010): *Automated Conformance Verification of Hybrid Systems*. In: *10th Int. Conf. on Quality Software, QSIC 2010*, IEEE Computer Society, pp. 3–12. Available at http://dx.doi.org/10.1109/QSIC.2010.53.

[12] Timothy A. Budd & Ajet S. Gopal (1985): *Program testing by specification mutation*. *Computer languages* 10(1), pp. 63–73. Available at http://dx.doi.org/10.1016/0096-0551(85)90011-6.

[13] Mats Carlsson, Greger Ottosson & Björn Carlson (1997): *An Open-Ended Finite Domain Constraint Solver*. In: *9th Int. Symp. on Programming Languages: Implementations, Logics, and Programs*, PLILP '97, Springer, pp. 191–206. Available at http://dl.acm.org/citation.cfm?id=646452.692956.

[14] R. DeMillo, R. Lipton & F. Sayward (1978): *Hints on test data selection: Help for the practicing programmer*. *IEEE Computer Society* 11(4), pp. 34–41.

[15] Richard G. Hamlet (1977): *Testing programs with the aid of a compiler*. *IEEE Transactions on Software Engineering* 3(4), pp. 279–290.

[16] C.A.R. Hoare & Jifeng He (1998): *Unifying Theories of Programming*. Prentice-Hall International.

[17] Yue Jia & Mark Harman (2011): *An Analysis and Survey of the Development of Mutation Testing*. *IEEE Transactions on Software Engineering* 37(5), pp. 649–678. Available at http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62.

[18] Elisabeth Jöbstl, Martin Weiglhofer, Bernhard K. Aichernig & Franz Wotawa (2010): *When BDDs Fail: Conformance Testing with Symbolic Execution and SMT Solving*. In: *3rd Int. Conf. on Software Testing, Verification and Validation, ICST 2010*, IEEE Computer Society, pp. 479–488. Available at http://doi.ieeecomputersociety.org/10.1109/ICST.2010.48.

[19] Willibald Krenn & Bernhard K. Aichernig (2009): *Test Case Generation by Contract Mutation in Spec#*. *Electronic Notes in Theoretical Computer Science* 253(2), pp. 71 – 86, doi:10.1016/j.entcs.2009.09.052.

[20] Sidney Nogueira, Augusto Sampaio & Alexandre Mota (2008): *Guided Test Generation from CSP Models*. In: *5th Int. Colloquium on Theoretical Aspects of Computing, ICTAC 2008*, *LNCS* 5160, Springer, pp. 258–273. Available at http://dx.doi.org/10.1007/978-3-540-85762-4_18.

[21] Philip Alan Stocks (1993): *Applying formal methods to software testing*. Ph.D. thesis, Department of computer science, University of Queensland.

[22] Jan Tretmans (1996): *Test Generation with Inputs, Outputs and Repetitive Quiescence*. *Software - Concepts and Tools* 17(3), pp. 103–120.

[23] Martin Weiglhofer & Bernhard Aichernig (2010): *Unifying Input Output Conformance*. In: *Unifying Theories of Programming*, *LNCS* 5713, Springer, pp. 181–201. Available at http://dx.doi.org/10.1007/978-3-642-14521-6_11.

Rule-based Test Generation with Mind Maps

Dimitry Polivaev
Giesecke & Devrient GmbH
Munich, Germany
dimitry.polivaev@gi-de.com

This paper introduces basic concepts of rule based test generation with mind maps, and reports experiences learned from industrial application of this technique in the domain of smart card testing by Giesecke &Devrient GmbH over the last years. It describes the formalization of test selection criteria used by our test generator, our test generation architecture and test generation framework.

1 Introduction

Testing is very important for smart card development because failure costs can be very high. Smart card manufacturer Giesecke & Devrient GmbH spends significant effort on implementation of tests and the development of new testing techniques. This paper presents some of the latest methodological results.

Our test cases should be short and easy to understand. They should execute fast and run independently from other test cases, which simplifies debugging in case of test failure. In the past all test cases were separately specified and implemented by test engineers. For testing of complex systems thousands of such test cases containing setup, focus, verification and post processing were implemented. Manual development of redundant code of the test scripts resulted in high costs of maintenance and change. We studied existing test generation techniques and decided to develop our own approach.

Known test generation techniques often concentrate on only one modeling abstraction. For instance the classification tree method [3] provides exploration of test case space based on systematic partitioning of test input and output data. It uses classification tree as a model of a test space. It does not make use of SUT models e.g. for automatic data partitioning or selection of relevant data combinations.

There are also classical model based approaches where test cases and test coverage are generated from the SUT models. This technique is supported by different test generation tools like Conformiq Designer [2], LEIRIOS Test Designer [6] or Smartesting CertifyIt [7]. They do not consider test strategies to be the responsibility of test engineers. The test strategies are implemented in the tools. To influence the test generation the SUT model should be changed, other ways to introduce e.g. additional variations of test data or different preconditions leading to the same model state are not supported. Hence the models become more complex, they include test specific elements and can not be reused for different kinds of tests.

Also, models usually need to be complete, i.e. they need to contain all actions which can be used by test cases. As a consequence, for complex systems where test preconditions need to refer to different subsystems, the latter must be modeled as well because there is no other way to add the required actions to the generated test cases. And finally, test case generation using model state exploration techniques have high requirements on computer power and limitations on model state space.

Usage of different kinds of models allows software engineering principles to be applied to test development. Identification of different abstractions makes it possible to use replaceable components for controlling different aspects of the test generation. An interesting example of separating different aspects of test generation in different models is given by Microsoft Spec Explorer [5]. There are models of

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 103–114, doi:10.4204/EPTCS.80.8

SUT and also so-called slices, representing test scenarios considered by the generator. However it still focuses on model state exploration making the SUT model the central element of test generation with the limitations described above.

Some other tools model test sequences. This technique is represented by e.g. .getmore [8]. Here a test sequence model is mixed with a SUT model, making it hard to reuse them separately. They also do not care about the specification of different data inputs within the same test sequence or definition of model independent test coverage goals.

In our approach there are three kinds of models. The most important one is a **test strategy.** It models a test case space and is responsible for the amount and variance of the generated test cases. Test strategies are placed in the focus of the test development. Test engineers explicitly define them, taking advantage of their knowledge, intuition and experience. Then there is a **model of system under test (SUT)** which simulates SUT data and behavior and, last nut not least, a **test goal model** which defines the test coverage. All of these model types operate with different abstractions and need different notations.

This paper explains a formalization of test strategy and test goals used by our test generator. It also describes test generator architecture and a test generation framework. The framework allows generation of test suites from modular reusable components. In addition to the three kinds of models listed above there are components called solvers and writers responsible for the output of the test scripts.

Structure of this paper. This paper describes models and other components used in our test generator. Section 2 introduces an example SUT used in all following sections to demonstrate the introduced concepts. Property based test strategies developed as rule sets are explained in section 3. Test goals used for test selection based on test coverage measurement are introduced in section 4. Section 5 describes the other test generation components and demonstrates our test generation architecture. Our experiences from the card development projects using this test generation technique are then reported. The paper ends with a small conclusion section.

2 Example: calculation of phone call costs

Let us consider the following specification as an example of a system under test. It specifies software for the calculation of phone call costs. The interface consists of two functions:

- setCheapCallOptionActive(isActive)
- calculateCallPrice(country, phoneNumber, day, callBeginTime, callDuration)

The first function is used to activate or deactivate a special tariff.

Function calculateCallPrice satisfies the following requirements:

If an invalid country or phone number is used, the function returns 0, otherwise the used tariff is given by a price table: The destination can be obtained from the country as follows:

Destination	Standard Tariff	CheapCall Tariff	At night, on the weekend	Time unit (seconds)
National	0.10$	0.07$	0.03$	1
International_1	1.00$	0.50$	0.80$	20
International_2	2.00$	1.20$	1.80$	30

Table 1: Call costs pro time unit dependent on call time and destination.

- if country is empty or 'National' the destination is set to National ,
- if country is 'Greenland', 'Blueland' or 'Neverland' the destination is set to International 1,
- if country is 'Yellowland' or 'Redland' the destination is set to International 2

All other country values are invalid.
Tariff "At night" applies if the call begins between 8 pm and 6 am.
Tariff "On the weekend" applies if the call begins on Saturday or on Sunday.
Call duration given in seconds is rounded up to units specified in column "Time unit".
Maximal call duration is limited to 24 hours.

3 Exploration of test case space with test strategies

Tests are developed as a part of the quality assurance process. They should check that software implementation satisfies the user requirements. Each tested requirement should be covered by a sufficient set of test cases checking all relevant aspects. There is an infinite number of possible test cases many of which should be implemented. Hence test development requires good systematics to choose which ones to actually implement.

We call a model, describing how test case space is systematically explored, a **test strategy**. It defines the number and variance of test cases in a generated test suite. For example, the test strategy could define which combinations of valid and invalid input data should be used. As we explain later, test strategies usually not only guide input data coverage, they also describe other test aspects e.g. test intention (e.g. a good case or a bad case), expected results or a test case name.

The base concept of test strategy definitions is a test case property.

3.1 Test case properties

The **test case property** is a key-value pair. Each property describes a test case characteristic such as test name, variations of test actions or their input data, other test case related data like expected results, test coverage and so on. Test properties can be used to declare what features are tested by a particular test case or to classify tests in a test suite in some another way. The keys and values can be represented by objects of arbitrary type. In our implementation the keys are strings and the values are some hierarchical data structures.

Some of the test case property values e.g. some of the input parameters can be assigned independently, but there may also be dependencies between property values. For example, in addition to properties corresponding to input parameters of the method "calculateCallPrice", the test strategy could define classifying properties

- i*sCallValid* with possible values *true* and *false*,
- *failureReason* with possible values *invalidCountry* and *invalidNumber*, which makes sense only if the call is not valid.
- *destination* with values *National*, *International_1* and *International_2*.

According to the requirements, if test case property *destination* is assigned value *International_1*, parameter *country* can be chosen only from values *Greenland*, *Blueland* or *Neverland.*

A test strategy specifies range and number of the generated test cases by describing combinations of the test case property values considering their dependencies. The test case space covered by a test strategy is completely defined by the strategy property space. When all required property values are set, a script with the test case can be written.

3.2 Rule-based exploration of property space

Test strategies can be represented by a collection of business rules building an ordered **rule set**. The rules specify variations of the test property values and dependencies between them.

Each rule defines values of one property called its **target property**. Because each test case definition depends on values of many properties, the test strategy is given by a set of rules. The values can be defined as expressions referencing other property values and arbitrary function calls. The rule can also add new rules to the rule set. The added rules have other target properties. They remain in the rule set, as long as iterations over the values of the target property of the rule where they are defined continue.

There are two kinds or rules: **iteration rules** and **default rules**.

The iteration rules define lists of values for their target properties. Their evaluation is triggered by assignment of a value to another property set by another iteration rule (so-called forward chaining).

The default rules specify values of properties that have not been assigned by the iteration rules. Such a rule can be evaluated when its target property value is requested by another rule or by the test generation algorithm (so-called backward chaining).

3.2.1 Iteration rules

Each iteration rule contains three parts: the **WHEN**-part describes when the rule is applied, the optional **IF**-part describes an additional condition and the **THEN**-part describes the action setting the list of property values and sometimes adding new rules to the set.

The WHEN-part can reference an arbitrary number of properties. If it does not reference any property, the rule is applied when test generation starts, otherwise it waits until all referenced properties become assigned by actions of other iteration rules.

If the rule is executed and its condition is satisfied, then its target property is sequentially assigned values from the value list set by the rule action. Each assignment triggers evaluation of the dependent iteration rules. Each test generated by the rule engine can be uniquely identified by the values of the properties iterating over more than one value.

The iteration value list can be attributed with an option ”shuffled” changing the order of the list elements every time the iteration resumes from the beginning of the list. It produces different random value combinations of parallel iterated properties. Hence if generation runs with different seeds it creates different test cases.

If a rule sets an empty list of property values, it means that no value can be found for the already set values of other properties. In this case rule engine tries to change a value of some property referenced in its WHEN part.

Some informal examples of the iteration rules.

Let us look at some examples of the iteration rules. They will be repeated in mind map notation in a later part of this article.

Iteration rule 1: **WHEN** (empty) **IF** (empty) **THEN** property *isCallValid* is sequentially assigned values *true*, *false*

Iteration rule 2: **WHEN** property *isCallValid* is assigned **IF** it has value *true* **THEN** property *destination* is sequentially assigned values *National, International_1* and *International_2*

Iteration rule 3: **WHEN** property *isCallValid* is assigned **IF** it has value *true* **THEN** property *callDuration* is sequentially assigned values *1* and *60*

Iteration rule 4: **WHEN** property *destination* is assigned **IF** it has value *International_1* **THEN** property *country* is sequentially assigned values *Greenland, Blueland, Neverland*

3.2.2 Default rules

Default rules contain only an optional IF - part with a condition and a THEN - part describing an action. They are evaluated whenever their target property value is requested from an IF- or THEN-part of another rule or from any other test generation component but has not been assigned yet. The action assigns some value to the rule's target property. Default rules are not designed for producing iterations, they always assign single values.

Informal example of a default rule

Default rule 1: **IF** *destination* = *International_2* **THEN** assign to *country* value *Redland*

3.2.3 Rule stack

Iteration rules with the same properties in the WHEN part and the same target property build a so-called rule stack. They are processed in the opposite order to their definition. If some rule with empty condition or with satisfied condition was found no other rules from the stack are executed. There are also rule stacks with default rules.

Rule sets must be self-consistent: if a property has already been assigned, no other iteration rule may try to reassign it.

Rules added to the stack later can override previously defined rules. This can be used to define strategy variations overloading some default strategy. For example there can be some common and product dependent subsets for testing of a product line.

3.2.4 Rule engine

There is a rule engine generating combinations of test case properties corresponding to single test cases. It starts with a given rule set and executes all iteration rules from rule stacks with an empty WHEN part, iterates over all values from the property lists and processes chained iteration rules as properties become assigned. Each combination of property values can become a test case.

The rule engine tracks all dependencies between properties. Property A is called dependent on property B if there is a rule with target property A which refers to property B from its WHEN, IF or THEN part. When a property iterates over different values, the next value is assigned only after all iterations over its dependent properties finishes.

The rule engine run terminates after all iterations specified by iteration rules are finished.

Whenever an unassigned property is requested, the rule engine executes a related default rule if it is available and applicable according to its IF-part.

3.2.5 Example

Iteration rules 1 to 4 from the above example generate following property combinations:

```
1:$isCallValid:TRUE/$destination:National/$callDuration:1
2_$isCallValid:TRUE/$destination:International_1/$country:Greenland/$callDuration:60
3:$isCallValid:TRUE/$destination:International_1/$country:Blueland/$callDuration:1
4:$isCallValid:TRUE/$destination:International_1/$country:Neverland/$callDuration:60
5:$isCallValid:TRUE/$destination:International_2/$callDuration:1
6:$isCallValid:FALSE
```

The rule engine simultaneously iterates over values of properties *country* and *callDuration*. If their lists had the option *shuffled* set, mutual combinations of their values would be randomized.

3.2.6 Mind map representation

The rule sets modeling test case space can be implemented in scripts or with mind maps.

In the mind maps relations between properties are represented by relative positions of the mind map nodes.

All examples given in figures 1,2 and 3 demonstrate how the above set of rules can be implemented in a mind map.

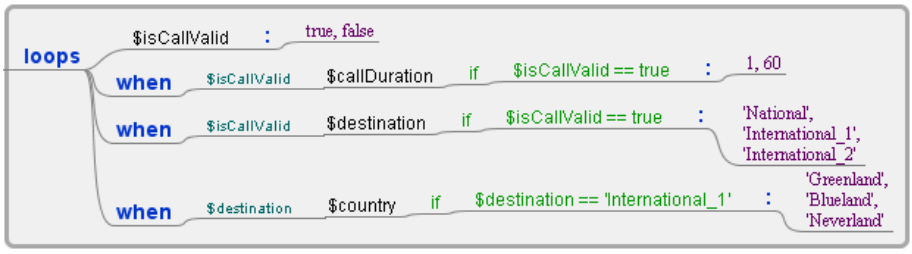


Figure 1: Exact copy of the above rules in the mind map notation.

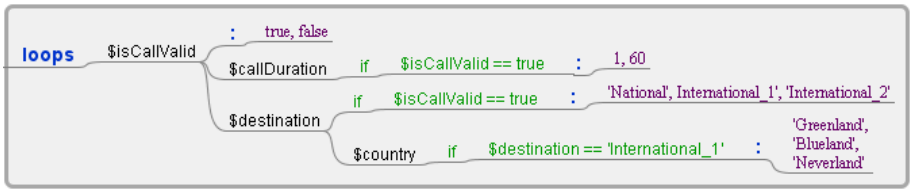


Figure 2: Here the WHEN relation is represented by a property definition outgoing from another property definition. The IF-parts are specified explicitly.

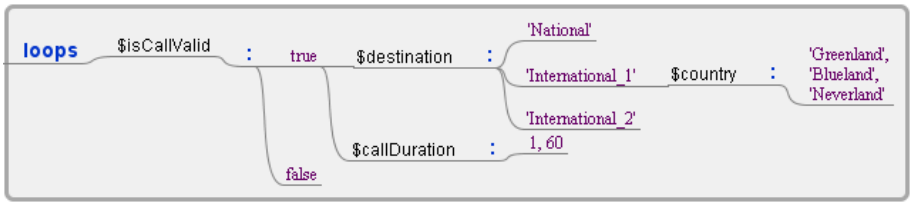


Figure 3: The map structure is used for representing both WHEN and IF parts. IF equality conditions are specified by property definitions outgoing from value definitions.

Use of mind maps for test strategy implementation simplifies development, reviewing and improvement of the strategy. Mind maps as a representation of generation rules offers good visualization, automatic context dependent node formatting, search and filtering of the rule sets.

4 Test coverage controlled by test goals

All possible combinations of input parameters cannot be tested within a reasonable amount of time. If test strategies generate too many test cases, the test cases contributing to the achievement of desired coverage criteria must be distinguished from the test cases, which can be discarded without negative effects on coverage. Therefore test selection and requirement coverage criteria should be defined in addition to the test strategy rules. Such criteria are called **test goals**.

Test goals can also be used to check completeness and correctness of test generation. If some goals are not achieved, it can indicate an error in test generation components or in the specification.

We differentiate between finite and infinite test goals. A **finite test goal** is basically a check list which can contain SUT model code coverage, the modified condition/decision coverage or the boundary condition coverage, values of the test input parameters, model predicted results or model interim data separately or in a combination. For instance, in our phone call example, a test goal can require that some calculated call prices are covered by the complete test suite or by its subset, limited to calls to some particular destination.

The finite goal is defined as a pair of the complete check list given as e.g. set of strings and a function mapping test case data to the set of values containing the check list values. A test is important for a finite goal, if the goal function called with the test related data returns some value contained in the check list for the first time since the generation was started.

Because sometimes calculation of the complete check list is difficult or even impossible as in the case of model path coverage, there are also so-called **infinite goals**. These consist only of their goal function and there is no predefined check list. The test is important for an infinite goal if the function returns a result not previously returned. The check list is automatically filled with all returned values.

The test goal functions can be defined as expressions using the test case property values. And every expression defined on test case property values can be used to define a new test case property itself. For instance the goal functions can use SUT model execution results, its intermediate states, collected code coverage and state coverage data. A set of all statements contained in the model can be obtained from the model automatically. It builds a check list for a goal based on the model code coverage. Check lists for state coverage or result coverage related goals can be separately created by test engineer. This way coverage of the test suite based on model code and state coverage is ensured.

Given a set of goals, test generation statistics are collected during the test generation. They describe how often any particular goal value was returned for the generated test cases.

Goals related to combinations of test case properties specified in the strategy rules can be defined in the mind maps as in our example. The example test goal given in figure 4, requires that all possible combinations of country and tariff are tested with a valid call.

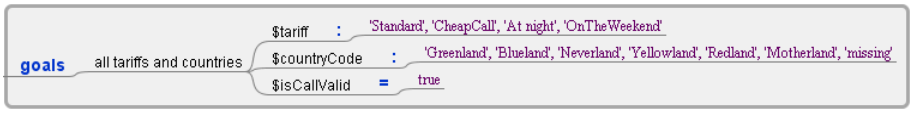


Figure 4: Example of property based test goal definition in a mind map

5 Test engineering

For the sake of maintainability and changeability of test suites they should be generated from modular organized reusable components. The complete test generation architecture is described in this section.

5.1 Test components

Previous sections explained **test strategies** and **test goals** used for test variant generation, selection and test coverage estimation. These, together with a so-called test script **writer,** call functions from the SUT related components **model** and **solver** which model the behavior of system under test and know, or are able to calculate how the SUT state can be manipulated or checked.

The **writer** is a component which creates executable test scripts from the test property values. For instance it outputs test header, test name, description, commands, comments etc. It is the only module which depends on the language and libraries used by the generated test scripts. It is called by a framework for each generated test case separately.

The **model** calculates expected results and reports data for statement coverage, path coverage, modified condition / decision coverage, internal states at given statements, given a function name, input parameters and data state before call. The collected coverage information can be used for test coverage measurement and for test goal definition. A model code coverage related test goal can be defined as a pair of the set of all statements contained in the model and a function returning a list of statements covered by the given test case. Similar goals can be defined for measuring of the modified condition / decision coverage and of boundary coverage.

The **solver** decides which SUT functions should be called and which parameter values should be used in order to bring the SUT into a required state and to check its state. For instance, test commands in preconditions, post processing or verification can be calculated in this component. In our example there is a function setCheapCallOptionActive(isActive). To use this function in a precondition the writer can obtain the complete precondition data from the solver.

The solver can be implemented using tools statically analyzing the model. However direct implementation is often easier and more efficient. It is particularly the case for complex systems, where test preconditions require calls to different subsystems not covered by the model.

If a suitable solver is available, the test strategy can directly define iterations over test output data related or model state related test case properties. It is useful for generating a test suite with test cases covering wished output data or SUT model states. Implementing such solvers can be difficult. Thus test strategies more often iterate over different preconditions and test input data. Then the output data, code and state coverage are controlled by the test goals and not by the strategy. This approach makes complex solvers superfluous but requires much more test case property combinations to be tried before all goals are achieved.

5.2 Component reusability

The proposed design simplifies reuse of the same components for different tasks because they can be replaced independently in order to create different test suites.

- Different strategies can be used for testing of different features of SUT.
- Different models can be used for testing of different products / product variations.

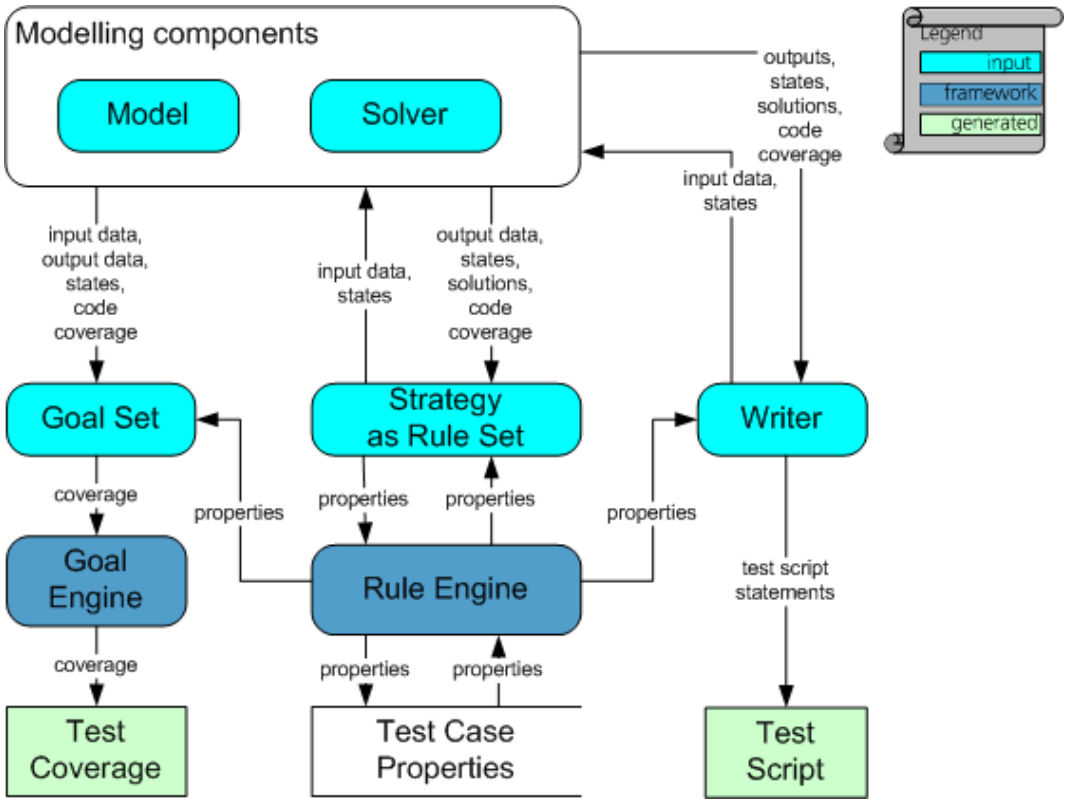


Figure 5: Test generation data flow

- Different solvers can be used to prepare an initial state of the test target, to verify its state after executing the focus or reset it at the end of the test case using different input data and command sequences.
- Different writers can be used for generating scripts for different script languages.
- Different test goals can be used for variation of test depth, e.g. for smoke tests, regression tests, exhaustive tests etc.

5.3 Test generation framework

The described ideas have been implemented as a test generation framework. The framework allows modular development of test suites similar to development of software. The test components are comparable to software components in a software development. The test generator produces test scripts and test coverage information like a compiler building executable software applications.

Our framework contains tools and libraries for the development of the test components and for the generation of executable test scripts in arbitrary textual languages.

For the development of test strategies and test goals as mind maps, the framework uses mind map editor Freeplane, which is available under GPL. It allows you to specify rules as mind maps, and supports mind map development through context dependent text formatting, map filtering and search. There is a converter transforming mind map based sets of rules and goals into the generation scripts.

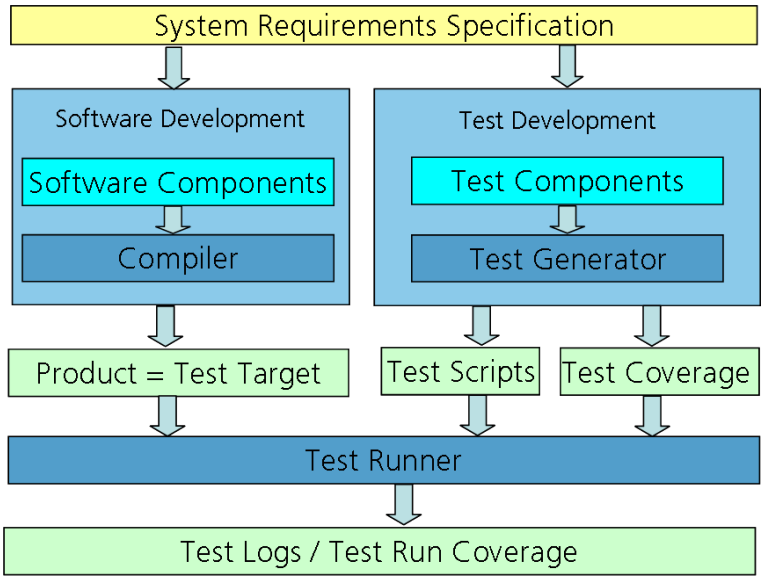


Figure 6: Test generation process compared to software development

The framework contains a rule engine evaluating the rules and a goal engine collecting information about test coverage achieved by generated test cases. Using information from the goal engine, our rule engine can also effectively reduce the explored property space, and thus the number of the generated property combinations, skipping iterations over properties not relevant for the achievement of the test goals. This procedure is similar to the one described in [1].

The framework contains a compiler for the programming language used for the development of writer and model components. The compiler generates code for automatic measurement of code, path and mc/dc coverage and boundary condition coverage needed for the coverage related goals. For the purpose of creating the test script files it supports test script templates embedded into the generation code. Such templates can access script variables and functions. The language has support for large integer arithmetic, complex data structures, it contains closures and statement blocks as method arguments. The test generator software translates scripts written in this language into java code, which is then compiled and executed. Java libraries can be called directly from the scripts if it is required.

6 Framework evaluation based on real life experience

The framework is currently used in projects testing the newest MasterCard, VISA smart cards and java cards. It demonstrated excellent scalability. In big projects it was used to generate tens of thousands of test cases but it was also efficient for generating small test suites containing a few hundred of test cases. For example, the test strategy definition of one typical project consists of 1135 iteration rules and 202 default rules, defined in 4007 nodes contained in 14 mind maps. The model implemented 1467 functional requirements. The test goal definitions were based on the complete code coverage, the modified condition/decision coverage and boundary coverage of the SUT model. They could be obtained automatically. The generator reduced the explored property space as described above, generated 90389 property value combinations, selected and created 5312 test cases.

Our overall positive experience has shown that the described methodology can be effectively used for test generation for different test targets. Test implementation effort has been reduced by more than 50% compared to methods not using the test generation. We achieved even greater reduction of test adaptation efforts on specification updates because they often caused only minor changes of the SUT model and the test generation could be repeated.

Test development using the proposed methodology fits well in existing processes. It is helpful to start with the manual implementation of some test cases which can later be used for creating the test templates embedded in a writer component. After they are available, development of all necessary components can be started. In a project team, they are implemented by different people at the same time, giving a natural way to divide the work. All components can be developed incrementally. It is possible to have a small number of test cases, with a high coverage of the requirements, available quickly. While the first found bugs are being fixed, the strategies can be extended. And there is no need to implement a full model of the SUT. Only aspects required to define a strategy and test goals or to calculate expected results should be considered.

The method achieves better test case systematics compared to manual test development we used before. It results from the use of formally defined test strategies and the monitoring of test coverage by test goals. Test case input data determined by the test strategy with random values and mutually random value combinations for different properties additionally increase test coverage and help to discover further bugs. Specification errors can be found earlier with the aid of model code coverage information and test goals statistics collected during the test generation. Detailed debugging information like the model code coverage is supplied. This also helps to reduce effort for test object debugging.

7 Conclusion

Our methodology identifies different kinds of test generation related models. It offers different abstractions and notations for their definition and suggests to implement them in different ways, and just as detailed as necessary to achieve the test goals.

It sees the test strategy as the driving force of the test generation and explains how to implement it using a special form of business rules systematically defined in mind maps. The maps look similar to classification trees known from the classification tree method but they are more powerful. Their strength is to consider complex dependencies between property values, visually specify their relations and dynamically change a rule set during the strategy run.

Information obtained from SUT models including model code coverage data and internal states of the model at arbitrary points of model execution can also be used by the strategy and by the other components. The test generation framework supplies a compiler for the development of SUT models and implements the measurement of model coverage, the rule engine and the goal engine.

Use of the framework in different real life projects demonstrated its high efficiency, excellent scalability and good performance resulting in a reduction of testing costs and improved maintainability and changeability of the created test suites.

References

[1] Chandrasekhar Boyapati, Sarfraz Khurshid & Darko Marinov (2002): *Korat: automated testing based on Java predicates*. *SIGSOFT Softw. Eng. Notes* 27, pp. 123–133. Available at `http://dx.doi.org/10.1145/566171.566191`.

[2] Conformiq: *Conformiq Designer*. http://www.conformiq.com/products/conformiq-designer/.

[3] M. Grochtmann & J. Wegener (1995): *Test Case Design Using Classification Trees and the Classification-Tree Editor CTE*. *Proceedings of the 8th International Software Quality Week(QW '95)* .

[4] Antti Huima (2007): *Implementing Conformiq Qtronic*. In Alexandre Petrenko, Margus Veanes, Jan Tretmans & Wolfgang Grieskamp, editors: *Testing of Software and Communicating Systems, 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007, Proceedings*, *Lecture Notes in Computer Science* 4581, Springer, pp. 1–12, doi:10.1007/978-3-540-73066-8_1.

[5] Microsoft: *Spec Explorer*. http://research.microsoft.com/en-us/projects/specexplorer/.

[6] Smarttesting: *Leirios Test Designer*. http://www.leirios.com/contenu.php?cat=25&id=44/.

[7] Smarttesting: *Smartesting CertifyIt*. http://www.smartesting.com/index.php/cms/en/product/certify-it/.

[8] Jan Tretmans, Florian Prester, Philipp Helle & Wladimir Schamai (2010): *Model-Based Testing 2010: Short Abstracts*. *Electronic Notes in Theoretical Computer Science* 264(3), pp. 85 – 99, doi:10.1016/j.entcs.2010.12.016. Available at http://www.sciencedirect.com/science/article/pii/S1571066110001635.

Constraint-Based Heuristic On-line Test Generation from Non-deterministic I/O EFSMs*

Danel Ahman†
Computer Laboratory
University of Cambridge
Cambridge, UK
danel.ahman@cl.cam.ac.uk

Marko Kääramees
Department of Computer Science
Tallinn University of Technology
Tallinn, Estonia
marko.kaaramees@ttu.ee

We are investigating on-line model-based test generation from non-deterministic output-observable Input/Output Extended Finite State Machine (I/O EFSM) models of Systems Under Test (SUTs). We propose a novel constraint-based heuristic approach (Heuristic Reactive Planning Tester (χRPT)) for on-line conformance testing non-deterministic SUTs. An indicative feature of χRPT is the capability of making reasonable decisions for achieving the test goals in the on-line testing process by using the results of off-line bounded static reachability analysis based on the SUT model and test goal specification. We present χRPT in detail and make performance comparison with other existing search strategies and approaches on examples with varying complexity.

1 Introduction

Model Based Testing (MBT) is one of various test automation approaches. We consider a version of MBT where the System Under Test (SUT) is represented by a formal model and treated as a "black-box" with an interface. MBT is commonly used to test conformance of the SUT to its model. A SUT may be modelled by a non-determistic model because of abstraction, distributed behaviour or freedom allowed in the specification. Testing conformance in that case requires on-line testing to react to the actual behaviour of the SUT.

A widespread approach to modeling SUTs for test generation is using either Finite State Machines (FSMs) or Extended Finite State Machines (EFSMs) [8, 1, 15]. Test generation that includes the input data generation from EFSM models has been handled with different methods, including evolutionary algrithms [5], scenarios [17] and symbolic techniques [15]. The formal symbolic framework and notation of conformance for models involving data components is handled in [2, 16]. On-line methods for test generation from non-deterministic models have also been studied by various authors [12, 11, 9, 15].

Although there is a variety of approaches for test generation from EFSMs, most of the methods are not applicable or tend to be inefficient when applied to non-deterministic and industrial-scale systems for on-line test generation for specified test goals (e.g., coverage criteria). In this paper we propose an Heuristic Reactive Planning Tester (χRPT) to improve the scalability and performance of Reactive Planning Tester [15, 4] by an heuristic constraint-driven on-line test generation technique. The only approaches we are aware of that have comparable goals are presented in [1, 6]. The comparison is presented in Section 4.

The integral part of χRPT is an on-line decision-making algorithm responsible for computing the stimuli to the SUT based on various constraints emerging from the model of the SUT. This algorithm

*This work was supported by the Estonian Science Foundation grant no. 7667 and ELIKO Competence Center.
†The first author was at the Tallinn University of Technology when the bulk of this work was carried out.

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 115–129, doi:10.4204/EPTCS.80.9

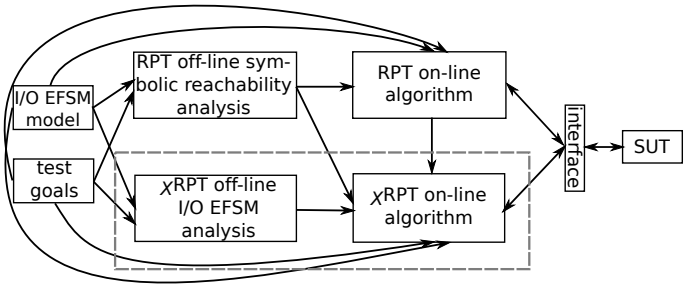


Figure 1: Workflow of I/O EFSM based on-line test generation and execution

draws inspiration from the paradigm of Constraint-Based Local Search (CBLS) [10], which has evolved into programming language Comet that we use for prototyping. As χRPT is based on on-line decision-making, it can also be used in reactive model-based planning in testing (cf. [18]). This is because χRPT computes only one move at a time and is able to react to the observed output of the SUT and the changes in test goals on-the-fly. General workflow of I/O EFSM based on-line test generation and execution discussed in this paper is depicted in Fig. 1. The scope of χRPT is highlighted with a *dashed rectangle*.

The rest of this paper is structured as follows. In Sect. 2 we outline the relevant background theory and preliminaries. Next, in Sect. 3, we describe χRPT in detail. The experimental results are described and analyzed in Sect. 4. Finally, Sect. 5 includes the discussion and further work.

2 Preliminaries

In this section we introduce the background theory of χRPT. At first, in Sect. 2.1 and 2.2, we define the modeling formalism and general testing framework. Next, we give the necessary description of the Reactive Planning Tester (RPT) in Sect. 2.3. The background theory is also illustrated with a simple three-variable counter example.

2.1 Input/Output Extended Finite State Machines

In this paper we assume that the SUT is modelled as an output-observable deterministic or non-deterministic I/O EFSM over a first-order theory. For simplicity, the definitions given in this paper use formulas of first-order theory of linear integer arithmetic. It is also applicable to other theories where the Satisfiablity Modulo Theories (SMT) problem is decidable.

Definition 1. *A constraint over variables X is a first-order formula of the chosen theory (e.g., over arithmetic expressions) where variables in X occur as free variables. It is assumed, but not required to be quantifier-free for efficiency reasons.*

Definition 2. *An I/O EFSM M is a tuple $(L, l_0, X, D, I, O, G, U, T)$ where L is a finite set of locations, l_0 is an initial location, $X = X_S \cup X_I \cup X_O \cup X_{Tr}$ is a disjoint set of finite sets of state, input, output and trap variables, D is a constraint over X constraining the domain of variables, I is a finite set of input labels that may have an associated set of parameters $x_1, ..., x_n$ $(x_k \in X_I)$, O is a finite set of output labels that may have an associated set of parameters $x_1, ..., x_n$ $(x_k \in X_O)$, G is a finite set of guard conditions (constraints), U is a finite set of transition update functions and $T \subset L \times I \times O \times G \times U \times L$ is a finite set of transitions.*

The definition of I/O EFSMs is inspired by UML State-charts and allows intuitive modelling of interactions between the system and its environment. The main difference with the straightforward semantics of the formalisms of Symbolic Transition Systems (STSs) and Input-Output Labelled Transition Systems (IOLTSs) [2] is that I/O EFSMs allow transitions to have both input and output labels assigned to them simultaneously. It is possible to define the semantics based on STSs such that a transition of I/O-EFSM corresponds to two consecutive transitions of STSs. However, we keep the input and output together because the presented method deals with interactions as unitary events. Guard conditions and all other constraints also implicitly include variable domain constraints D for all variables in X.

Functions $source(t)$, $target(t)$, $guard(t)$, $update(t)$ and $out(l) \subseteq T$ on I/O EFSMs serve as a shorthand reference to source and target location, guard condition and update function of a transition t and the set of outgoing transitions from location l respectively.

Definition 3. *A state $s \in S$ of I/O EFSM is a pair (l, α) of a location $l \in L$ and assignment α of state variables in X_S.*

Therefore, input variables X_I and output variables X_O are not considered to be a part of an I/O EFSM state. The values of X_I and X_O are relevant only for the current transition. The input variables that model the parameters of input can only occur in the guards and in the right hand sides of updates. Output variables that model the parameters of output can only occur in the left hand sides of updates.

A transition (l, i, o, g, u, l') is *enabled* and can be taken when an input i is received and guard g evaluates to $true$ on the current state and values of the input parameters. The receiving of input i is modelled on the logical level by a special input-variable $iLabel$. We can say that a transition is enabled when a formula $g \wedge D \wedge iLabel = i$ evaluates to $true$.

Definition 4. *An I/O EFSM is said to be non-deterministic, if there exists a state l for which two or more guard conditions of transitions in $out(l)$ are non-disjoint and therefore satisfiable using the same input and state variables assignment. Transitions $t \in out(l)$ satisfying this criteria are called rival transitions and are denoted $Rival_t$. Transitions $t' \in out(l)$ with $guard(t')$ equal to or weaker than $guard(t)$ (i.e., the guards are undistinguishable from each other for every assignment of input variables) are perfect rivals to t and denoted $Rival_t^P$.*

It is further assumed that the SUT is modeled as an *output observable* I/O EFSM. This means that even though in a given state (l, α) multiple rival transitions in $out(l)$ may be taken in response to the input, the observed output of the SUT determines the actual move and the next state unambiguously. It is possible to relax the condition in expense of increasing the complexity of the on-line computation to find the best next move from a set of possible ones. We find this kind of limited non-determinism to be practical both for modelling and test generation point of view.

To further clarify the given notions, Fig. 2 describes an output observable non-deterministic I/O EFSM model of a simple thee-variable counter where all variables have domains $[0, 25]$:
$M_1 = (L, l_0, X, D, I, O, G, U, T)$, where $L = \{l_0, l_1, l_2\}$, $X_S = \{x, y, z\}$, $X_I = \{i\}$, $X_O = \oslash$, $X_{tr} = \{trap\_t_3\}$,
$D = (0 \leq x \leq 25 \wedge 0 \leq y \leq 25 \wedge 0 \leq z \leq 25 \wedge 0 \leq i \leq 25 \wedge (trap\_t_3 = true \vee trap\_t_3 = false))$,
$I = \{START, COUNT, RESET\}$, $O = \{T0, Tx, Ty, Tz, T2, T3\}$
The model M_1 is non-deterministic as it has the following sets of rival transitions: t_x with t_y, t_y with (t_x, t_z), t_z with (t_y, t_2) and t_2 with t_z.

2.2 Modeling Test Goals

A *test goal* is a property of the SUT that is intended to be tested. The test goals are modeled as sets of *traps* attached to specific transitions. We classify traps as *uncovered, covered* or *discarded.*

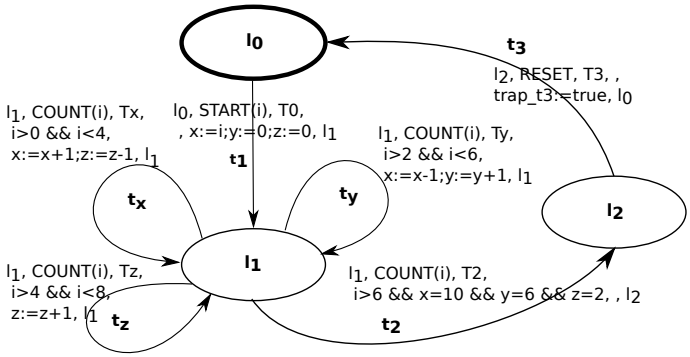


Figure 2: I/O EFSM model M_1 of a simple three-variable counter

Definition 5. *A trap is a pair (t_i, P_{tr}) where t_i is a transition and P_{tr} is a constraint on $X_S \cup X_I$. The trap (t_i, P_{tr}) is covered when the transition t_i has been taken from the state and input where P_{tr} was satisfiable.*

In the following, a trap (t_i, P_{tr}) is denoted with a boolean *trap variable* $tr \in X_{Tr}$ (or sometimes with $trap\_ti \in X_{Tr}$). For a trap (t_i, P_{tr}), the value of the trap variable tr is $false$ if the trap is either uncovered (initially all traps) or discarded and becomes $true$ when the trap gets covered (cf. Def. 5). An uncovered trap becomes discarded only when it is determined in the on-line algorithm that it can not be covered.

The constraint P_{tr} can also include trap variables tr' of other traps $(tr \neq tr')$ which, in turn, introduces a *dependency relation* between the traps and enables one to talk about specific paths (i.e., sequences of trap-labeled transitions). Defining the test goals as sets of traps allows one to use different test strategies such as *state, transition, path* and *constrained path coverage* [13]. Furthermore, we say that a test goal is *fully satisfied* when all its traps have been covered.

2.3 Reactive Planning Tester

The Reactive Planning Tester (RPT) is an on-line tester for black-box conformance testing of SUTs that are modeled by non-deterministic output-observable I/O EFSMs. As the RPT has been thoroughly described in [15], we outline only the aspects relevant to χRPT.

Due to possible non-determinism, it is not possible to compute inputs for the SUT in advance, i.e., off-line. Therefore the workflow of the RPT is further divided into on-line and off-line procedures for efficient input generation and computationally hard symbolic analysis. The RPT off-line process performs symbolic reachability analysis and generates a system of reachability constraints describing the feasible paths (i.e., sequences of transitions) needed to be taken to cover the defined traps.

For every trap tr the RPT generates the following: 1) A *weakest constraint* $C^*_{l,tr}$ on state variables X_S and *path length* $\mathscr{L}^*_{l,tr}$ for every location l. $C^*_{l,tr}$ represents a symbolic state for which there is a path with length (i.e., the number of transitions in a path) less or equal to $\mathscr{L}^*_{l,tr}$ that covers the trap tr. 2) A *guarding constraint* $C^g_{l,tr}$ on state and input variables $X_S \cup X_I$ for every transition t, which represents a symbolic state and input for which the transition t is the initial transition of a shortest path to the trap tr. Assignment of $false$ to any of the constraints represents a failure to generate a feasible constraint.

The reachability constraints are generated by a recursive procedure backwards from the trap until one of the following termination conditions is met - (i) a fixpoint is reached, (ii) the constraints have been generated for the initial location or (iii) a predefined bounded depth limit is reached. The case (iii) is common to location-, transition- or path-wise large systems where it is computationally infeasible to generate the constraints until termination condition (i) or (ii) is satisfied.

During the on-line procedure, the goal of the RPT is to guide the SUT by using the reachability constraints generated off-line to satisfy the test goals. In general, the RPT on-line procedure works as follows. First, an uncovered trap with shortest path length $\mathscr{L}^*_{l,tr}$ and satisfiable constraint $C^*_{l,tr}$ is selected in the current state (l,α). Then the assignment of input variables is computed by solving $C^g_{t,tr}$ (adding the negations of the guards of rival transitions where necessary) for transition t in $out(l)$. Finally, the tester feeds the inputs to the SUT and observes its output to test conformance, simulates the transition and repeats the steps in the target location.

The RPT on-line algorithm is only applicable when the constraints have been generated for location l the SUT is in and $C^*_{l,tr}$ is satisfiable in the given state (l,α). If either of these conditions is unsatisfied, the generated reachability constraints can not be directly used in this state. To overcome this problem, the RPT currently incorporates simple but inefficient random and anti-ant search strategies [9] to make randomized intermediate moves when the constraint based input generation is not possible.

For illustrating the RPT, the reachability constraints generated for the model M_1 in Fig. 2 and trap $(t_3,true)$ (denoted by $trap\_3$, abbreviated as tr_3) using the bounded depth 2 are the following: $(C^*_{l_0,tr_3},\mathscr{L}^*_{l_0,tr_3}) \leftarrow (false,-)$, $(C^*_{l_1,tr_3},\mathscr{L}^*_{l_1,tr_3}) \leftarrow (x=10\wedge y=6\wedge z=2,2)$, $(C^*_{l_2,tr_3},\mathscr{L}^*_{l_2,tr_3}) \leftarrow (true,1)$, $C^g_{t_1,tr_3} \leftarrow false$, $C^g_{t_x,tr_3} \leftarrow true$, $C^g_{t_y,tr_3} \leftarrow (x=11\wedge y=5\wedge z=2)$, $C^g_{t_z,tr_3} \leftarrow (x=10\wedge y=6\wedge z=1)$, $C^g_{t_2,tr_3} \leftarrow (x=10\wedge y=6\wedge z=2)$ and $C^g_{t_3,tr_3} \leftarrow true$.

3 Heuristic Reactive Planning Tester (χRPT)

In this section we describe the Heuristic Reactive Planning Tester (χRPT) for I/O EFSM based on-line test generation. The aim of χRPT is to improve the scalability and efficiency of on-line test generation. χRPT is designed to be used when the RPT on-line algorithm is not directly applicable and is, similarly to the RPT, divided into off-line and on-line procedures described in Sect. 3.1 and 3.2.

3.1 Off-line Analysis of I/O EFSMs in χRPT

In this section we describe the off-line analysis of I/O EFSMs. The goal of this analysis is to provide extra input for the proposed on-line decision-making algorithm and to avoid unnecessary repetition of on-line computations. The analysis is made after the off-line symbolic reachability analysis in the RPT and prior to the on-line test generation and execution. This analysis is based on the I/O EFSM model M and the output generated by the RPT off-line process. As a result, distance matrix $Dist$, search neighbourhood $L^C_{tr,l}$ and sets Tr_+ , Tr_- of traps are generated as follows.

The *all-pairs shortest distance matrix Dist* is computed from the underlying directed control graph of the I/O EFSM with $Dist_{l,l}=0$ for reflexive transitions and $Dist_{l_1,l_2}=\infty$ when there is no path from l_1 to l_2 for all $l,l_1,l_2 \in L$. The use of graph based distances between locations is introduced to favour closer traps in the on-line decision making algorithm introduced in the next section.

The *search neighbourhood* L^C is a set indexed both by traps $tr \in X_{Tr}$ and locations $l \in L$ of sets of closest (control graph based) locations (not necessarily direct neighbours) to l that have reachability constraints generated for them for a given trap tr. If the location $l \in L$ has reachability constraints generated for itself for trap tr, $L^C_{tr,l}$ includes both l and the next closest set of locations that have reachability constraints assigned to them for trap tr. In addition, the locations whose reachability constraints are equivalent to or weaker than domain constraints are removed from the search neighbourhood as they provide no information to our on-line algorithm.

```
1: ON_LINE_ALGORITHM(M, l, α, X, D, L^C, L^T, Dist, Tabu, C*, Tr+, Tr−):
2: while Tr+ ≠ ∅ do
3:    (Tr+, Tr−) ← ON_LINE_RPT(l, α, C*, Tr+, Tr−)
4:    (MinHeap, Tr+, Tr−, L^T) ←
5:                 GENERATE_SOLUTION_CANDIDATES(l, α, X, D, L^C, L^T, Dist, Tabu, C*, Tr+, Tr−)
6:    best_move ← CHOOSE_MOST_PROMISING(l, α, X, D, L^C_tr,l, Dist, Tabu, C*, Tr+, Tr−)
7:    new_state ← INTERACT_WITH_SUT(M, l, α, best_move)
8:    if new_state = () then
9:       return TEST_FAILED
10:   else
11:      (Tabu, l, α) ← new_state
12: return TEST_FINISHED
```

Figure 3: On-line decision-making algorithm of χRPT consisting of four subroutines

```
1: ON_LINE_RPT(l, α, C*, Tr+, Tr−):
2: while π1(SAT_MODEL(X_I, α, C*_l,tr)) for any tr ∈ Tr+ then
3:    (l, α) ← RPT_ON_LINE_ALGORITHM(tr, l, α)
4:    SET_COVERED(tr) ; UPDATE_NEIGHBOURHOOD(Tr+, Tr−)
5: return (Tr+, Tr−)
```

Figure 4: Subroutine #1 that uses the RPT on-line algorithm to cover a given trap as soon as reachability constraints are satisfied

$Tr_+ \subset X_{Tr}$ and $Tr_- \subset X_{Tr}$ are the respective *sets of uncovered traps* that can be covered and that can not be covered from the current state. A function $update\_neighbourhood(Tr_+, Tr_-)$ is used to update the two sets by removing already covered traps from Tr_+ and moving new coverable traps from Tr_- to Tr_+. In this paper it is assumed that M has a connected underlying control graph. As a result, the partitioning of traps between Tr_+ and Tr_- in function $update\_neighbourhood(Tr_+, Tr_-)$ is based only on the dependency ordering of the traps, i.e., which trap variables are used in the constraints of other traps. After the off-line analysis has been completed, $Tr_+ \cup Tr_- = Tr$ holds.

In case the underlying control graph of M not being connected, one could also add a strongly connected component (SCC) analysis to the algorithm. This could then be used to give higher priority to traps in the current SCC to cover them before moving to the next SCC. Moreover, one could also add other selection and partitioning criteria but this is left as a further research.

3.2 On-line Decision-Making Algorithm of χRPT

In this section we describe the on-line algorithm of χRPT responsible for decision-making during on-line test generation in situations when the RPT on-line algorithm is not immediately applicable (cf. discussion in Sect. 2.3). The χRPT on-line algorithm consists of a top-level algorithm in Fig. 3 and four subroutines in Figs. 4, 5, 6, 7. They all make use of the I/O EFSM model M, the reachability constraints generated by the RPT off-line algorithm and the output of χRPT off-line algorithm discussed in Sect. 3.1.

In general, the on-line algorithm of χRPT works by making computationally inexpensive operations first and then iteratively excludes *solution candidates* (i.e., possible moves) as the computations become more costly until the most promising solution candidate has been selected. *Solution candidates* are tuples

1: **GENERATE_SOLUTION_CANDIDATES**$(l,\alpha,X,D,L^C,L^T,Dist,Tabu,C^*,Tr_+,Tr_-)$:
2: **for all** $run \in \{0,1\}$ **do**
3: **for all** $tr \in Tr_+$ **do**
4: $MinHeap' \leftarrow \emptyset$
5: **for all** $t \in out(l)$ **do**
6: $formula \leftarrow guard(t) \bigwedge_{t' \in Rival_t} \neg guard(t') \wedge D(X)[update(t)/X]$
7: $formula \leftarrow formula \wedge \neg Tabu_{tr,l}$
8: $(b,\alpha_i) \leftarrow$ SAT_MODEL$(t,X_I,\alpha,formula)$
9: **if** $\neg b$ **then continue**
10: **for all** $l_C \in L^C_{tr,l}$ **do**
11: **if** $(run = 0) \vee (run = 1 \wedge \neg(target(t) \in L^T_{tr} \wedge source(t) \in L^T_{tr}))$ **then**
12: $dist \leftarrow 1 + Dist_{target(t),l_C}$
13: $viol \leftarrow \nu(C^*_{l_C,tr}[update(t)/X][\alpha(x_s)/x_s][\alpha_i(x_i)/x_i])$
14: $f \leftarrow dist^2 + viol^2$
15: $MinHeap' \leftarrow MinHeap' \cup (t,\alpha_i,l_C,tr,f)$
16: **if** $MinHeap' = \emptyset$ **then**
17: **if** run = 0 **then**
18: $Tabu_{tr,l} \leftarrow \emptyset$; $L^T_{tr} \leftarrow L^T_{tr} \cup l$
19: **else**
20: SET_DISCARDED(tr) ; UPDATE_NEIGHBOURHOOD(Tr_+,Tr_-)
21: **else**
22: $MinHeap \leftarrow MinHeap \cup MinHeap'$
23: **if** $MinHeap \neq \emptyset$ **then**
24: **break**
25: **return** $(MinHeap,Tr_+,Tr_-,L^T)$

Figure 5: Subroutine #2 that generates solution candidates, excludes excessive ones and orders the remaining by fitness function values

1: **CHOOSE_MOST_PROMISING**$(l,\alpha,X,D,L^C_{tr,l},Dist,Tabu,C^*,Tr_+,Tr_-)$:
2: $best\_f = \infty$; $best\_move = \emptyset$
3: **for up to** N **tuples** $(t,\alpha'_i,l_C,tr,f) \in MinHeap$ **do**
4: $formula \leftarrow guard(t) \wedge D(X)[update(t)/X] \wedge \neg Tabu_{tr,l}$
5: $formula \leftarrow formula \bigwedge_{t' \in Rival_t} \neg guard(t')$
6: $(b,\alpha_i) \leftarrow$ OPTIMIZE_MODEL$(t,X_I,\alpha,formula,\nu(C^*_{l_C,tr}[update(t)/X]))$
7: $dist \leftarrow 1 + Dist_{target(t),l_C}$
8: $viol \leftarrow \nu(C^*_{l_C,tr}[update(t)/X][\alpha(x_s)/x_s][\alpha_i(x_i)/x_i])$
9: $f \leftarrow dist^2 + viol^2$
10: **if** $f < best\_f$ **then**
11: $best\_f \leftarrow f$
12: $best\_move \leftarrow (\alpha_i,t,l_C,tr,f)$
13: **return** $best\_move$

Figure 6: Subroutine #3 that selects the most promising solution candidate as a best possible move from a given state

```
1:  INTERACT_WITH_SUT(M, l, α, best_move):
2:  (α_i, t, l_C, tr, f) ← best_move
3:  iLabel ← GET_ILABEL(α_i)
4:  actual_move ← FEED_TO_SUT(CREATE_MESSAGE(iLabel, α_i)))
5:  if SUT_CONFORMS(M, actual_move) then
6:    Tabu_{tr,l} ← Tabu_{tr,l} ∨ MAKE_TABU_ELEMENT(actual_move, best_move)
7:    (l, α) ← GET_STATE(SIMULATE_MOVE(actual_move))
8:    return (Tabu, l, α)
9:  else
10:   return ()
```

Figure 7: Subroutine #4 that creates a message based on the best move, feeds this message to the SUT and observes its output

$$
\begin{array}{ll}
A,B-logical\ formulas & \nu(a>=b)=abs(min(0,\nu(a)-\nu(b))) \\
a,b-arithmetic\ expressions & \nu(a>b)=abs(min(0,-1+\nu(a)-\nu(b))) \\
\nu(a=b)=abs(\nu(a)-\nu(b)) & \nu(a<b)=abs(max(0,1+\nu(a)-\nu(b))) \\
\nu(a\neq b)=\nu(a<b\vee a>b) & \nu(a<=b)=abs(max(0,\nu(a)-\nu(b))) \\
\nu(A\wedge B)=\nu(A)+\nu(B) & \nu(A\vee B)=min(\nu(A),\nu(B))
\end{array}
$$

Figure 8: Minimal set of computation rules for the violations degree function ν

(t,α_i,l_C,tr,f) consisting of a transition t, input variables assignment α_i, search neighbourhood location l_C, trap tr and fitness function value f which is used to measure and compare the quality of solution candidates. In this paper, the *fitness function* consists of the sum of squares of the control-graph based distance to the search neighbourhood location l_C and the violations degree (cf. Def. 6) of the reachability constraint $C^*_{l_C,tr}$. The χRPT on-line algorithm also makes use of the RPT on-line algorithm when the SUT has been guided to a state where reachability constraints for at least one trap are satisfied making the RPT on-line algorithm applicable. The four subroutines that the χRPT on-line algorithm (Fig. 3) consists of are described in the following paragraphs.

Definition 6. *The violations degree of a constraint C is the value of the function $\nu(C)$ that is inspired by fitness function computation in [14]. The minimal set of computation rules for $\nu(C)$ is given in Fig. 8. The negation of logical formulas is pushed inside and eliminated (if possible) by De Morgan's laws and arithmetic equivalences (i.e., $\nu(\neg(a>b))\equiv\nu(a<=b)$).*

Subroutine #1 The χRPT on-line algorithm uses the RPT on-line algorithm through the subroutine ON_LINE_RPT(...) outlined in Fig. 4 as soon as the SUT has been guided to a state (l,α) where reachability constraints $C^*_{l,tr}$ for some trap tr are satisfied. Then, the RPT on-line algorithm is called using the procedure RPT_ON_LINE_ALGORITHM(l,α,tr) to guide the SUT to a state (l',α') covering tr. Next, if $C^*_{l',tr'}$ is satisfied for any tr' in the state (l',α') such that $tr\neq tr'$, the routine is repeated.

Subroutine #2 The subroutine GENERATE_SOLUTIONS_CANDIDATES(...) in Fig. 5 is used to generate a limited amount of solution candidates by excluding excessive ones. The core idea of this routine is to collect solution candidates ordered by the calculated fitness using the min-heap $MinHeap$. Excessive solution candidates are excluded by strengthening the guards of transitions t in $out(l)$ for the *satisfiability test* procedure SAT_MODEL$(t,X_I,\alpha,formula)$ with the negations of guards of rival transitions and *tabu list* $Tabu_{tr,l}$ element (ll. 6-7). Given the assignment α of state variables, SAT_MODEL$(t,X_I,\alpha,formula)$

returns a pair (b, α_I) of a boolean value indicating whether the logic formula was satisfiable and a model for variables in X_I. The algorithm uses a *tabu list* to avoid converging into local optimums and therefore avoid guiding the SUT to infinite loops by keeping the partial history of previous moves. For every trap tr and location l, the tabu list element $Tabu_{tr,l}$ consists of a disjunction of conjunctions of I/O EFSM transition together with input and state variables assignments. These elements explicitly record the moves made in the on-line algorithm. Tabu list becomes *full* if none of the guards of $t \in out(l)$, strengthened with the negations of tabu list elements, are satisfiable in the given state. The general principles of tabu lists and related search strategies can be found from papers by Fred Glover et al. (e.g., [3]).

If no solution candidates for trap tr are collected to $MinHeap$ on the first run, then the guards are weakened by emptying the tabu list $Tabu_{tr,l}$. At the same time, the location l is added to L^T_{tr} (the set of locations l for each trap tr where tabu list $Tabu_{tr,l}$ has been emptied). In addition, in the second run, a new condition is added (l. 11) to avoid infinite looping. This condition states that if the tabu lists in the current location l and target location of transition $t \in out(l)$ have been emptied before (i.e., $target(t) \in L^T_{tr} \wedge source(t) \in L^T_{tr}$) then the move is not permitted. If no solution candidates are found for trap tr after the second run due to the new condition, then tr is marked as discarded. Unfortunately this condition does not guarantee the complete unreachability of tr but is merely an over-approximation which turns out to be strong enough for χRPT. Therefore the notion *discarded* is used instead of *unreachable*.

Subroutine #3 The subroutine CHOOSE_MOST_PROMISING(...) in Fig. 6 is used to choose the most promising solution candidate (i.e., the best possible move in a given state). This subroutine compares only up to $N \in \{1, 2, ..., size(MinHeap)\}$ solution candidates from all the solution candidates collected in $MinHeap$. We allow to vary N to allow different configurations to be used, e.g., for different time requirements. However, the selection cost in this round is higher than before because of the use of *constraint solving* in procedure OPTIMIZE_MODEL(t, X_I, α, C, f) (as opposed to SAT test used in Subroutine #2), which also optimizes the assignment α_i of input values to minimize the fitness function f.

Subroutine #4 The subroutine INTERACT_WITH_SUT(...) in Fig. 7 is used for interaction with the SUT using the most promising solution candidate *best_move* found in Subroutine #3. The message that will be fed to the SUT consists of an input label together with possibly empty list of parameters (cf. Def. 2). The input label is obtained from the valuation of the variable *iLabel*. The input label also determines the input variables whose valuation will be sent as input parameters. After feeding this input message to the SUT, the subroutine observes the actual move made. If the SUT conforms to the I/O EFSM model, a tabu list element $Tabu_{tr,l}$ is updated with the result of MAKE_TABU_ELEMENT$(actual_move, best_move)$ that is a constraint recording the actual or the best move (cf. comments below). Finally, the algorithm returns the updated tabu list and the new state corresponding to making *actual_move*.

It has to be noted that if one would only consider actual moves made by the SUT for tabu list element construction, then non-determinism of perfect rivals might force the algorithm to loop infinitely. The *actual_move* and *best_move* need not be the same and the tabu list would not reflect the history correctly. Therefore, in our algorithm, we enforce that *best_move* is used for tabu list element construction if *actual_move* is already present in the tabu list. Instead of this, a more sophisticated bounded fairness criteria could be introduced but it has been omitted from this paper due to space restrictions.

Using χRPT has two possible outcomes. First, testing can be declared *failed* if the observed behavior of the SUT does not conform to the given I/O EFSM model. Alternatively, testing is declared *finished* when Tr_+ becomes empty and no uncovered traps can be added to it. At this point, not all of the test goals might be satisfied because some of the traps might still be discarded or uncovered. One should then add these traps back to Tr_-, reset the SUT and run the on-line algorithm again from the initial state.

The decision-making time of this algorithm is dictated by SAT_MODEL and OPTIMIZE_MODEL as these are the two most costly operations (in the worst case double exponential to the size of constraints).

The size of the constraints depends non-trivially on the structure of the I/O EFSM model, RPT planning depth limit and simplifications involved. In χRPT, the number of calls to these operations in each state (l,α) is linear to the number of traps, locations in $L^C_{tr,l}$ and transitions in $out(l)$. Therefore, the performance could be risen by considering different heuristic (sub)methods for the most promising solution candidate selection to reduce the number of calls made to these operations. The analysis of such heuristic algorithms (e.g., simulated annealing or differential evolution) is omitted from this paper. In the worst case, χRPT falls back to an anti-ant-like strategy and has similar performance. On the other hand, experimental results in the next section give evidence that, on average, χRPT is superior when compared to anti-ant by offering significantly better performance and measures to ensure termination.

We continue with the three-variable counter example M_1 (we abbreviate $trap\_t3$ as tr_3) to illustrate χRPT further. First, we look at χRPT in the initial state $(l_0,\{x \leftarrow 0, y \leftarrow 0, z \leftarrow 0\})$ where there is only one solution candidate as the only transition in $out(l_0)$ is t_1 and the only location in $L^C_{tr_3,l_0}$ is l_1. In this situation, SAT_MODEL returns us a tuple $(true, i = 0)$ and thus the value of the fitness function is $f \leftarrow 1^2 + 18^2$. On the other hand, OPTIMIZE_MODEL minimizes the value of f and returns $(true, i = 10)$ and thus $f \leftarrow 1^2 + 8^2$.

Secondly, we look at χRPT in the next state $(l_1,\{x \leftarrow 10, y \leftarrow 0, z \leftarrow 0\})$. This time we have four transitions t_x, t_y, t_z, t_2 in $out(l_1)$ and one location l_1 in $L^C_{tr_3,l_1}$. SAT_MODEL first eliminates solution candidates for transitions t_x (as z would violate domain constraints) and t_2 (as the guard is not satisfied). On the other hand, both SAT_MODEL and OPTIMIZE_MODEL return $(true, i = 4)$ and $(true, i = 6)$ for other solution candidates corresponding to t_y, t_z. As a result, the fitness function values are $f_y \leftarrow 1^2 + 8^2$ and $f_z \leftarrow 1^2 + 7^2$, and therefore, the solution candidate corresponding to t_z is selected as the most promising. This process continues until the state $(l_1,\{x \leftarrow 10, y \leftarrow 6, z \leftarrow 2\})$ is reached where the reachability constraint $C^*_{l_1,tr_3}$ is satisfied and the RPT on-line algorithm can be applied to cover trap tr_3.

4 Experimental Results

In this section we compare different strategies for generating test sequences for specific test goals. In particular, we compare the performance of χRPT with other search strategies such as the RPT off-line algorithm [4, 15] and the randomized version of the anti-ant strategy [9]. As χRPT can be viewed as heuristic explicit-state forward reachability analysis rather than the symbolic analyzis done in the RPT, we have also chosen an explicit-state model checking tool UPPAAL [7] with modelling language close to EFSM for comparison. It gives a comparison between the guided (χRPT) and random (UPPAAL) explicit-state forward analysis. Although our method is intended for non-deterministic models, the comparison is easier to make on deterministic models. The following experiments were conducted on a 64-bit personal computer with 2.4GHz Intel Core 2 Duo CPU and 8GB of DDR3 RAM using prototypes written in Comet.

4.1 Single Trap Test Goals

In this section we analyse the three-variable counter introduced in Fig. 2 and the Inres Initiator depicted in Fig. 9. We consider only test goals containing one trap to give evidence of the performance of χRPT.

The Inres protocol is a well-known case study model in software testing and verification communities. The connection-oriented protocol consists of an Initiator that sets up a connection and sends data and a Responder that receives the data and closes the connection. In this paper we consider only the Inres Initiator depicted in Fig. 9 which mimics the formalization given in [1].

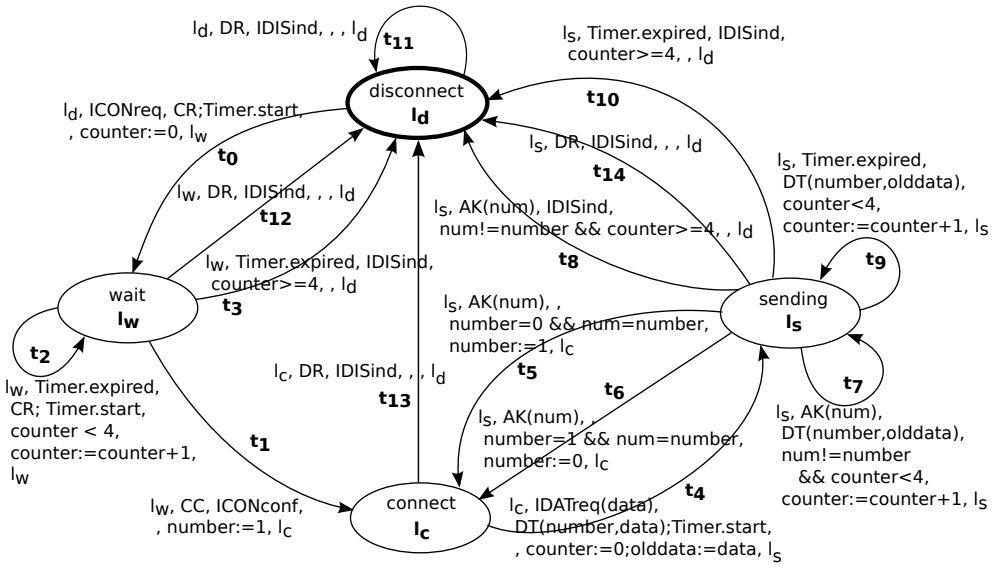


Figure 9: I/O EFSM model M_2 of Inres Initiator

Experimental results in Table 1 describe the performance of χRPT on three different test goals. One of the goals is defined on the three-variable counter model M_1 consisting of trap $trap\_t_3 = (t_3, true)$. The other two are defined on the Inres Initiator model M_2 - $trap\_t_8 = (t_8, true)$ and $trap\_t_5 = (t_5, true)$. The respective initial states of I/O EFSM models M_1 and M_2 are $(l_0, \{x \leftarrow 0, y \leftarrow 0, z \leftarrow 0\})$ and $(l_d, \{number \leftarrow 0, counter \leftarrow 0\})$. For all three traps, the reachability constraints are generated to depth 2 by the bounded RPT off-line algorithm. The results in Table 1 outline the optimal length of paths generated by the RPT off-line algorithm and UPPAAL together with algorithm run times. The results also describe the length of paths generated by χRPT (completing the path with the RPT on-line algorithm as soon as it is possible) and the paths generated by the randomized anti-ant search strategy.

The experimental results first indicate that χRPT is able to satisfy all three test goals. Moreover, it also outperformed the randomized anti-ant strategy significantly in all three cases. The path generated by χRPT is optimal in case of $trap\_t_8$ and $trap\_t_5$, but differs from the optimal computed by the RPT off-line algorithm and UPPAAL in case of $trap\_t_3$. The difference is not significant when compared to the failure of the anti ant search strategy and is caused by the combination of assignments in $update(t_1)$ and the optimization of input variables assignment. In Sect. 4.3 we also show that the difference between optimal and generated paths stays insignificant in case of larger industrial systems.

χRPT also performed efficiently time-wise. In all of the three test goals, the average time spent on decision making in each state is well below 10ms making χRPT a feasible candidate for industrial on-line testing frameworks. In addition, it is clearly visible that the combination of χRPT on-line algorithm together with bounded RPT off-line algorithm also time-wise outperformed the RPT (off-line + on-line).

4.2 Multiple Trap Test Goals

In this section we analyse the paths generated by χRPT for test goals consisting of multiple traps defined on the Inres Initiator model M_2 (Fig. 9). All of the traps considered here are dependent on preceding traps which means the traps can only be covered in the order they are defined. The reachability constraints are generated to depth 1 by the bounded RPT off-line algorithm and the initial state is taken from earlier.

Many of the test goals in Table 2 are inspired and partially taken from [1, 5]. We consider both (i)

Table 1: Experimental results of test goals consisting of one trap showing the lengths of the generated paths and algorithm run times

	M_1 $trap\_t_3$	M_2 $trap\_t_8$	M_2 $trap\_t_5$
Complete off-line RPT Path length (Work time (s))	11 (14.98)	8 (6.84)	6 (6.49)
Bounded off-line RPT Depth (Work time (s))	2 (2.09)	2 (4.5)	2 (4.5)
χRPT Total path length (Work time (s))	23 (0.20)	8 (0.043)	6 (0.037)
On-line path length (χRPT + RPT)	21 + 2	6 + 2	4 + 2
Time spent in each state (s)	0.0082	0.0061	0.0074
Randomized anti-ant Work time (s) (min/avg/max)	-	0.064 / 0.35 / 0.88	0.017 / 0.11 / 0.21
Path length (min/avg/max)	-	17 / 80 / 193	6 / 25 / 48
UPPAAL Path length (Work time (s))	11 (0.53)	8 (0.50)	6 (0.49)

traps that can be covered immediately one after another and (ii) traps that can not. In particular, traps of form (ii) force the SUT to be guided through a set of intermediate states before the next trap can be covered. Here, the definition of Inres Initiator given in [1] is used. The definition in [5] differs from the former by t_0 and t_4 and causes some of the test goals from [5] initially consisting of traps of form (i) to be actually of the form (ii).

The experimental results in Table 2 are given as 8 pairs of test goals and corresponding generated paths. These test goals consist each of 8 implicitly given and dependently defined traps. They are given as a sequence of transitions each of which has an implicit trap defined for it such that the trap constraint consist of a conjunction of trap variables of all the preceding traps in the test goal. We do not give explicit performance analysis in this section as the average time-wise performance conforms with the results from the previous section.

The test goals 1 - 3 consist of traps of form (i), which means that every trap can be covered immediately after the preceding one. The test goals of form (i) are used to confirm that they are indeed trivial for the combination of χRPT and the RPT. Test goals 4 and 5 both contain one trap of form (ii). It is clearly visible that although these traps can not be immediately covered after preceding ones, χRPT is able to guide the SUT through the necessary intermediate states. The last three test goals 6 - 8 contain traps randomly chosen from all possible traps and therefore contain multiple traps of form (ii). The results again confirm that χRPT is able to generate near-optimal paths for the test goals of form (ii).

4.3 Industrial Scale Telecom Billing System

In this section we consider an industrial scale telecom billing system. As this example originates from industry, we are unfortunately not permitted to depict it explicitly. Instead, we can only give a description of the given I/O EFSM by its general characteristics which includes 13 locations and 43 transitions between them, 2 input variables having domains $[0,11]$ and $[1,32000]$ and 8 state variables having domains $[0,1]$ (1), $[0,1000]$ (1) and $[0,32000]$ (6). On average, transition guards in this model consist of 20 state and input variables connected with logic and arithmetic operations.

Table 2: Generated paths (of the EFSM model transitions) for test goals consisting of multiple traps defined on the Inres Initiator model. Lists of transitions in parentheses illustrate how the SUT is guided through intermediate states to cover the next trap

No.	Test goal	Generated path
1	$t_0-t_1-t_4-t_7-t_6-t_4-t_5-t_4$	$t_0-t_1-t_4-t_7-t_6-t_4-t_5-t_4$
2	$t_{11}-t_0-t_1-t_4-t_6-t_4-t_5-t_4$	$t_{11}-t_0-t_1-t_4-t_6-t_4-t_5-t_4$
3	$t_0-t_2-t_1-t_4-t_7-t_6-t_4-t_5$	$t_0-t_2-t_1-t_4-t_7-t_6-t_4-t_5$
4	$t_0-t_3-t_0-t_1-t_4-t_6-t_4-t_7$	$t_0-(t_2,t_2,t_2,t_2,t_3)-t_0-t_1-t_4-t_6-t_4-t_7$
5	$t_0-t_2-t_1-t_4-t_7-t_7-t_7$	$t_0-t_2-t_1-t_4-t_7-t_7-t_7-(t_9,t_8)$
6	$t_1-t_8-t_{13}-t_5-t_{14}-t_2-t_9-t_{11}$	$(t_0,t_1)-(t_4,t_9,t_9,t_9,t_9,t_8)-(t_0,t_1,t_{13})-$ $(t_0,t_1,t_9,t_9,t_9,t_9,t_6,t_4,t_5)-(t_4,t_{14})-(t_0,t_2)-$ $(t_1,t_4,t_9)-(t_{14},t_{11})$
7	$t_2-t_{10}-t_6-t_3-t_4-t_{11}-t_7-t_0$	$(t_0,t_2)-(t_1,t_4,t_9,t_9,t_9,t_9,t_{10})-(t_0,t_1,t_4,t_6)-$ $(t_{13},t_0,t_2,t_2,t_2,t_2,t_3)-(t_0,t_1,t_4)-(t_{14},t_{11})-$ $(t_0,t_1,t_4,t_7)-(t_{14},t_0)$
8	$t_3-t_{11}-t_{13}-t_{10}-t_5-t_4-t_8-t_{12}$	$(t_0,t_2,t_2,t_2,t_2,t_3)-t_{11}-(t_0,t_1,t_{13})-$ $(t_0,t_2,t_1,t_4,t_9,t_9,t_9,t_9,t_{10})-$ $(t_0,t_1,t_4,t_9,t_9,t_9,t_9,t_6,t_4,t_5)-t_4-(t_9,t_9,t_9,t_9)-$ $t_8-(t_0-t_{12})$

The test goal whose analysis is given in Table 3 consists of a sequence of traps corresponding to exceeding the monthly mobile internet usage limit. As this model is considerably larger both location- and state-wise than the previous models M_1 and M_2, we also use it to compare how different bounds of the RPT off-line algorithm affect χRPT. We consider 4 different search depth bounds - 100, 50, 10 and 2 iterations. The optimal path has length 189 and it is found by the RPT off-line algorithm in 1.3 hours.

As one might expect, the anti-ant strategy failed to generate a successful path in reasonable time due to the significantly large search space. Similarly, UPPAAL also failed to generate a successful trace (and therefore also a test sequence) because the size of the search-space caused the explicit-state model checking to run out of memory. Moreover, UPPAAL's failure was independent of used configuration, e.g., depth-first/breadth-first search, state-space representation and reduction strategies.

On the other hand, χRPT was able to satisfy the test goal in each of the five cases of different RPT search depth bounds. From Table 3 we can first conclude that the generated path indeed depends on the RPT search depth bounds. This corresponds to the intuition that χRPT is complementing the backward RPT off-line algorithm with a forward on-line algorithm and the farther the RPT generates the reachability constraints, the more information they provide to χRPT. Secondly, we can conclude that the difference between the generated path and the optimal path does not strictly depend on the I/O EFSM size. Although the generated path for the simple counter model M_1 in Sect. 4.1 was 2 times longer than the optimal, the difference here for bounds 10 to 100 is less than 1.5 times for a significantly larger model. Only for depth 2 the generated path is significantly longer than optimal.

In conclusion, χRPT works efficiently with the given industrial model by generating near-optimal paths time-wise efficiently. As the SUT in this example is a relatively large component of an industrial system, we are able to give empirical backing to the capability of χRPT for handling components of industrial scale systems. Moreover, χRPT is also able to handle larger models when the RPT off-line

Table 3: Experimental results on the industrial scale telecom billing system

Complete off-line RPT Path length (Work time (s))	189 (4644)			
Bounded off-line RPT Depth (Work time (s))	100 (2120)	50 (1086)	10 (95)	2 (16)
χRPT Total path length (Work time (s))	230 (6,7)	255 (17,4)	275 (17,0)	1051 (153,4)
On-line path length (χRPT + RPT)	130 + 100	205 + 50	265 + 10	1049 + 2
Avg. time in each state	0,051	0,084	0,063	0,146
Randomized anti-ant	-	-	-	-
UPPAAL	-			

algorithm search depth bounds and χRPT configuration variables are modified accordingly.

5 Conclusions and Further Work

The motivation behind this research was to improve the scalability and efficiency of on-line test generation from non-deterministic output-observable I/O EFSMs. Although test generation and execution from EFSMs has been studied extensively, on-line test generation from non-deterministic models tends to confine itself to computationally inexpensive but inefficient strategies such as random search or anti-ant. In this paper, we proposed a constraint-based heuristic approach (Heuristic Reactive Planning Tester (χRPT)) for I/O EFSM-based on-line test generation that could be used when the RPT on-line algorithm [15, 4] is not applicable. χRPT is based on reachability constraints and properties of the underlying control graph of the I/O EFSM. We compared χRPT with other search strategies such as the RPT [15], a randomized version of the anti-ant strategy [9] and also the ones implemented in the model-checking tool UPPAAL [7] on a three-variable counter, Inres Initiator and an industrial telecom billing system.

The models considered in this paper are limited to EFSM models over linear arithmetics. This is an important extension compared to modelling systems using FSMs, but not all SUTs can be easily modelled using only linear arithmetics. All the results are applicable to models over different theories, provided that we have a satisfiability solver, optimization procedure and a function for calculating violations degree of the formulae of the used theory.

We have confined ourselves to only dependency based test goal and trap selection criteria and left additional analysis as further work. Further research is also needed for the use of χRPT with not connected, hierarchical and distributed I/O EFSMs. Moreover, further work will include improvements to the fitness function computation and stronger trap discarding and ordering conditions.

6 Acknowledgements

We thank Jüri Vain and Kullo Raiend for many valuable comments and discussions.

References

[1] Karnig Derderian, Robert Hierons, Mark Harman & Qiang Guo (2010): *Estimating the feasibility of transition paths in extended finite state machines*. Automated Software Engineering 17, pp. 33–56,

doi:10.1007/s10515-009-0057-9.

[2] L. Frantzen, J. Tretmans & T. Willemse (2006): *A Symbolic Framework for Model-Based Testing*. In Klaus Havelund, Manuel Núñez, Grigore Rosu & Burkhart Wolff, editors: *Formal Approaches to Software Testing and Runtime Verification*, *LNCS* 4262, Springer Berlin / Heidelberg, pp. 40–54, doi:10.1007/11940197_3.

[3] Fred Glover & Manuel Laguna (1996): *Tabu Search*. Kluwer.

[4] Marko Kääramees, Jüri Vain & Kullo Raiend (2010): *Synthesis of on-line planning tester for non-deterministic EFSM models*. In: *Proc of the 5th international academic and industrial conference on Testing - practice and research techniques*, TAIC PART'10, Springer-Verlag, Berlin, Heidelberg, pp. 147–154, doi:10.1007/978-3-642-15585-7_14.

[5] A. Kalaji, R.M. Hierons & S. Swift (2008): *Automatic generation of test sequences form EFSM models using evolutionary algorithms*. Working Paper, Brunel University.

[6] Abdul Salam Kalaji, Robert Mark Hierons & Stephen Swift (2009): *Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM)*. In: *Proc of the 2009 International Conference on Software Testing Verification and Validation*, IEEE Computer Society, Washington, DC, USA, pp. 230–239, doi:10.1109/ICST.2009.29.

[7] Kim G. Larsen, Paul Pettersson & Wang Yi (1997): *Uppaal in a nutshell*. *International Journal on Software Tools for Technology Transfer (STTT)* 1, pp. 134–152, doi:10.1897/IEAM_2009-036.1.

[8] D. Lee & M. Yannakakis (1996): *Principles and methods of testing finite state machines-a survey*. In: *Proceedings of the IEEE*, 84, pp. 1090–1123, doi:10.1109/5.533956.

[9] Huaizhong Li & C. Peng Lam (2005): *Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams*. In Ferhat Khendek & Rachida Dssouli, editors: *Testing of Communicating Systems*, *LNCS* 3502, Springer Berlin / Heidelberg, pp. 405–405, doi:10.1007/11430230_6.

[10] Laurent Michel & Pascal Van Hentenryck (2002): *A Constraint-Based Architecture for Local Search*. In: *OOPLSA'02*, ACM, pp. 83–100, doi:10.1145/582419.582430.

[11] M.Veanes, P.Roy & C.Campbell (2006): *Online testing with reinforcement learning*. In: *Proceedings of FATES/RV*, *LNCS* 4262, Springer, pp. 240–253.

[12] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann & W. Grieskamp (2004): *Optimal strategies for testing nondeterministic systems*. In: *ISSTA04, vol. 29 of Software Engineering Notes*, ACM, pp. 55–64, doi:10.1145/1013886.1007520.

[13] Luay Ho Tahat, Atef Bader, Boris Vaysburg & Bogdan Korel (2001): *Requirement-Based Automated Black-Box Test Generation*. In: *Proc of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, IEEE Computer Society, Washington, DC, USA, pp. 489–495, doi:10.1109/CMPSAC.2001.960658.

[14] Nigel Tracey, John Clark & Keith Mander (1998): *Automated Program Flaw Finding using Simulated Annealing*. In: *In the proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). Pages*, pp. 73–81, doi:10.1145/271771.271792.

[15] Jüri Vain, Marko Kääramees & Maili Markvardt (2011): *Online Testing of Nondeterministic Systems with the Reactive Planning Tester*. In Luigia Petre, Kaisa Sere & Elena Troubitsyna, editors: *Dependability and Computer Engineering: Concepts for Software-Intensive Systems*, IGI Global, pp. 113–150, doi:10.4018/978-1-60960-747-0.ch007.

[16] Margus Veanes & Nikolaj Bjørner (2011): *Alternating simulation and IOCO*. *International Journal on Software Tools for Technology Transfer (STTT)*, pp. 1–19.

[17] Margus Veanes, Colin Campbell, Wolfram Schulte & Nikolai Tillmann (2005): *Online testing with model programs*. In: *Proceedings of the 10th European software engineering conference*, ESEC/FSE-13, ACM, New York, NY, USA, pp. 273–282, doi:10.1145/1081706.1081751.

[18] Brian C. Williams & P. Pandurang Nayak (1997): *A Reactive Planner for a Model-based Executive*. In: *15th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1178–1185.

Model-Based Testing of Safety Critical Real-Time Control Logic Software

Yevgeny Gerlits Alexey Khoroshilov
Institute for System Programming of the Russian Academy of Sciences
Alexander Solzhenitsyn st. 25, 1009004
Moscow, Russia
gerlits@ispras.ru khoroshilov@ispras.ru

The paper presents the experience of the authors in model based testing of safety critical real-time control logic software. It describes specifics of the corresponding industrial settings and discusses technical details of usage of UniTESK model based testing technology in these settings. Finally, we discuss possible future directions of safety critical software development processes and a place of model based testing techniques in it.

1 Introduction

Role of safety critical systems in our life is increasing at a rapid pace. The distinctive characteristic of such systems is that their failure can be very dangerous for people or environment. As a result, development of safety critical systems is usually regulated by government agencies according to appropriate safety certification standards (e.g. DO-178B for avionics, BS EN 50128 for railways, IEC 60880 for nuclear, IEC 61508 for industry, IEC 62304 for medical devices). Requirements of these specifications make development and verification of safety critical software noticeably different from traditional processes.

The key differences that had an influence on our experience of model based testing development are as follows:

- requirement and architecture documents are carefully developed and maintained up to date;
- there is a need for requirements based testing including explicit requirements traceability;
- there is a need for source code coverage measurements depending on criticality level of a component under test;
- tool qualification is required including tools used to automate verification processes.

Other important elements of the standards such as safety analysis affect our work to a very little degree.

Usually requirement documents consist of two levels: HLR (high level requirements) and LLR (low level requirements). HLR describe what is expected behavior of a component. LLR and software design documents provide sufficiently detailed description how to implement HLR. LLR based tests covers source code well even in terms of advanced source code coverage metrics such as MC/DC [1] because source code is usually very close to LLR. Compliance between LLR and HLR is mostly verified using manual formal inspections. One more HLR verification technique is system-level tests based on HLR.

It is well known that model based testing works well when high quality testing is required. There are specialized model based testing technologies for real-time systems based on explicit automata definition, e.g. RT Tester [2], UPPAAL-TRON [3], Timed-TorX [4] and TTG [5]. On the other hand, there

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 130–144, doi:10.4204/EPTCS.80.10

are model based functional testing technologies such as SpecExplorer [6] and UniTESK [8] that automates test sequence generation from implicitly defined automaton models. If advanced test coverage is needed and model automata are rather complex, these approaches are more efficient than manual test development. In comparison with explicit automata definition, the implicit one is less tend to lose clarity and accuracy when the complexity of the functionality under test increases because the increase in complexity results in the growth of the number of states and transitions in the automaton model of the SUT (system under test). Information about usage of these approaches in the domain of real time safety critical software is very limited, so we present our experience regarding usage of UniTESK model based testing technology to real time systems with rather complex functionality.

The rest of the paper is organized as follows. Section 2 gives a short informal description of real time logic control subsystems. Section 3 briefly describes the key ideas of the UniTESK model based testing technology. Section 4 provides technical details of our usage of UniTESK technology for testing of safety critical real time logic control subsystems. Finally section 5 outlines future works and possible improvements in development processes of safety critical software.

2 Real-time logic control subsystems

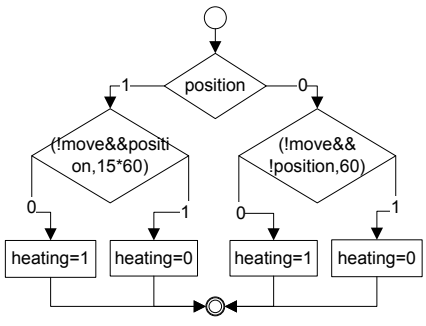
In this section we give a short informal description of the object under test and its environment, i.e. the target real-time logic control subsystem and the whole architecture of the RTES (real time embedded system) software respectively. We also provide a simple example of an iron automatic shut-off control subsystem. This subsystem is used in the subsequent sections to illustrate the most complicated aspects of our testing approach for control subsystems of RTES.

The RTES software consists of a number of subsystems. The responsibility of control subsystems includes analysis of the input data, decision making on the RTES reaction and generation of the output data needed to perform the reaction. Other subsystems are not limited but usually include subsystems that produce the input data for control subsystems and subsystems that process the output data produced by control subsystems. As a rule, subsystems producing the input data for control subsystems usually process the output data of some sensors and subsystems processing the output data produced by control subsystems usually perform the reaction of the RTES.

The subsystems being part of the RTES software are supposed to be called in the global control loop, i.e. they are subsequently called on each turn of the loop. The next turn of the control loop does not begin after the last subsystem has finished its work, but it starts after a certain period of time has expired. Let us refer to a turn of the control loop as a *cycle* and to the period of time after which the next cycle begins as the *cycle period*. The total execution time of all the subsystems on each cycle may not exceed the established cycle period for the RTES, otherwise the behavior of the RTES software is considered to be incorrect.

Control subsystems of RTES consist of a number of decision making algorithms. The base part of a decision making algorithm consists of a scheme of branch instructions which can often be very complex. Figure 1 contains a simple iron automatic shut-off control subsystem. The names move and position are the input parameters of the subsystem. The parameter *move* takes the value of 1 if the iron movement sensor detected that the iron has been moved since the beginning of the current cycle; otherwise the parameter takes the value of 0. The parameter *position* takes the value of 1 if the iron position sensor detected that the iron is in the vertical position on the current cycle; otherwise the parameter takes the value of 0. The name *heating* is the only output parameter of the subsystem. The value of 0 of this parameter prevents the iron sole from being heated; the value of 1 allows it to be heated.

Figure 1: Flow-chart of iron automatic shut-off control subsystem



Control subsystems of RTES use internal state variables to save data between cycles. Some state variables may be accessible for reading to other subsystems, some may not. The iron automatic shut-off control subsystem is simple. It does not have any state variables.

The key feature in designing of decision making logic in control subsystems of RTES is the possibility to build conditions in branch instructions on the basis of the notion of time. An example of a temporal condition is a Boolean formula which is required to be true for a period of time. Two such temporal conditions are used in the flow chart in figure 1. The condition $(!move\&\&!position, 60)$ evaluates to true if the Boolean formula $(!move\&\&!position)$ has evaluated to true for 60 seconds.

How does a control subsystem of RTES calculate that a Boolean formula keeps the value of true for a period of time? Let *system time* be the period of time passed since the start of the RTES execution. The value of the system time is fixed by the kernel of the RTES in the beginning of each cycle and remains fixed for all the subsystems of the RTES during the cycle period. The system time being constant during the cycle period is a natural architectural decision usually taken in the software design stage as it introduces determinism into the behavior of the RTES software. To calculate that a Boolean formula keeps the value of true for a period of time, the control subsystem evaluates the value of the Boolean formula on each cycle and using the system time accumulates the time interval during which the Boolean formula keeps the value of true from cycle to cycle.

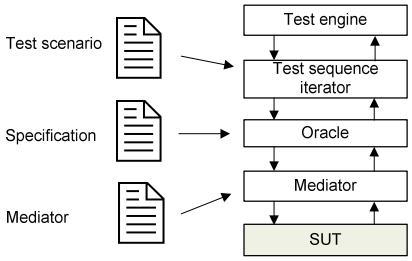
3 UniTESK technology

In this section we briefly describe the UniTESK testing approach for software systems which provide a synchronous interface [8]. An interface is considered to be synchronous if the next *stimulus* may only be applied after the *reaction* to the previous one has been received. The notions of the stimulus and the reaction are parts of the UniTESK terminology. Real life examples of a stimulus applied to the SUT are a method call or a form submission. Examples of a reaction of the SUT include the return value of a method call or a web page reload.

The SUT is considered as a black box in UniTESK and is supposed to provide a number of *interface functions* to access its functionality. The SUT may have an internal state. A call to an interface function with a set of values of the parameters is considered as an application of a stimulus to the SUT. It can result in a change in the internal state of the SUT and if the called interface function specifies the return value, it is returned and is considered as a reaction.

Figure 2 contains the universal UniTESK test system architecture. *Test engine* is a library compo-

Figure 2: UniTESK test system architecture



nent which implements a traversal algorithm for the abstract automaton. Information which can only be supplied by the tester is concentrated in three components: *oracle*, *mediator* and *test sequence iterator*. Compact formal descriptions are proposed for these components: *specification*, *mediator* and *test scenario* accordingly. The formal descriptions are developed in programming languages which extend industrial programming languages with a number of syntactic constructions. Those are the *specification extensions* of C and Java programming languages.[1]

Functional requirements are formalized in specifications from which oracles are generated. Oracles check the reactions of the SUT to the applied stimuli. The requirements specification technique used in UniTESK is based on the well-known Design by contract method [9]. The state of the specification models the state of the SUT. *State invariants* represent requirements to which all the specification states obtained during the execution of the test system must satisfy. The specification declares a *specification function* for each interface function being tested. The parameters of a specification function model the parameters of the corresponding interface function. The requirements to the behavior of the interface function are formalized in the corresponding specification function in the form of a *precondition* and *postcondition* of the interface function call. The specification also formulates a coverage criterion on the basis of the structure of the formalized requirements [10].

The precondition describes legal calls to the interface function on the basis of the current state of the specification and the parameter values of the specification function. A *specification stimulus* in UniTESK is a call to a specification function. It results in the check of the precondition. Violation of the precondition indicates that the test system tried to perform an illegal call to the interface function. Before checking the postcondition, the test system saves the current specification state and calls the mediator function with the set of values of the specification function parameters.

The formal description called mediator is intended to bind a specification to the SUT and must contain a *mediator function* for each specification function in specification. The signature and the type of the return value of the mediator function must be identical to those of the corresponding specification function. The mediator function transforms the specification representation of the input parameters of the interface function into the representation of the SUT, calls the interface function, transforms the return value of the interface function call into the representation of the specification, synchronizes the specification state with the state of the SUT and returns the specification representation of the return value. The specification representation of the return value is considered in the postcondition of the specification function as the return value of the specification function and is called *specification reaction*.

After the mediator function call has completed, the test system initiates the check of the postcondition

[1]The renunciation of specification extensions in favor of natural language constructs becomes a new trend in the evolution of UniTESK. The number of supported languages is growing. C++ and Python implementations of UniTESK are now available.

of the specification function. The specification reaction and the specification state after the call to the interface function are verified in the postcondition on the basis of the values of the input parameters of the specification function and so called the pre-state of the specification, i.e. the specification state at the time of the specification function call, which we know was preliminary saved.

The test scenario formalizes a model of the SUT in the form of the abstract automaton. The abstract automaton is used to automatically generate a sequence of *test actions*. The test scenario describes the abstract automaton implicitly. The base part of the test scenario consists of the following functions: a number of *scenario functions*, the *initialization function*, the *state generation function*, and the *finalization function*. A scenario function is responsible for organization of calls to specification functions. A simple case implies the development of one scenario function for each specification function. Although one scenario function may perform a number of calls to specification functions, one scenario function is considered to describe one *test action* from the point of view of the test scenario, i.e. a call to a scenario function is considered to perform one test action.[2]

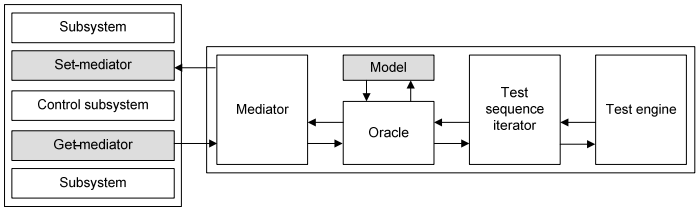
The abstract automaton of the SUT is being dynamically constructed during the execution of the test scenario. The initialization function moves the SUT and its specification into their initial states which must certainly be synchronized. After the initialization function has finished its execution, the state generation function is called. This function generates a state of the abstract automaton on the basis of the current state of the specification and/or the SUT. The test engine component either performs a test action which is defined for the current state of the abstract automaton and has not been performed yet or it performs an action defined for the current state to get to a state for which a test action is defined and has not been performed yet. Let the test engine applies a test action which has not been performed yet. A new transition is created from the state where the test action has been performed. The transition is marked with the test action performed.[3] The state of the SUT might change because of the test action. The state generation function is called to generate the end state for the transition. Construction of the test sequence terminates when the test engine has performed all the test actions defined for all the states reached during the execution of the test scenario. The finalization function is called at the end of the testing process. This function performs any actions related to the end of the testing process. For instance, it can release the allocated resources.

The main goal of the test scenario developer is to specify a set of states of the abstract automaton and a number of test actions for each state so that if the test system performs all the test actions, the target coverage criteria will be satisfied. The test scenario developer has to take into account some restrictions that the test engine component imposes on the abstract automaton. Although the abstract automaton is being dynamically constructed during the execution of the test scenario, the restrictions are imposed on the final abstract automaton. There are two main implementations of the test engine: a test engine that can traverse deterministic abstract automata [11] and a test engine that can traverse nondeterministic abstract automata [12]. The test engine for deterministic abstract automata demands the final abstract automaton to be finite, deterministic and strongly connected. The test engine for nondeterministic abstract automata demands the final automaton to be finite and to contain a deterministic, strongly connected, total, spanning subautomaton.

[2]If a scenario function contains some iteration statements, it generally describes a number of test actions for each state of the abstract automaton.

[3]If the scenario function being called does not contain any iteration statements, the transition is marked with the name of the scenario function; otherwise the transition is marked with the name of the scenario function complemented with the set of values of all the iteration variables.

Figure 3: Test system architecture for CSUT



4 Application of UniTESK for real-time logic control subsystems

The original UniTESK test system architecture undergoes some changes when applied to control subsystems of RTES. In this section we describe those changes and some features in the implementations of the individual architectural components.

4.1 Test system architecture

Software and hardware resources are often limited in the computer system on which the software of RTES must run. Running resource consuming test system components on the target computer system is impossible therefore the test system should be implemented as a separate program in order that it may be run on a separate computer system. The architecture of the test system for functional testing of control subsystems of RTES is represented in figure 3. It is composed of two programs which communicate synchronously. The software of RTES with two injected subsystems named set-mediator and get-mediator is in the figure 3 on the left. The main part of the test system is on the right. We will further refer to it simply as the test system.

The whole composition operates in the following way. At the beginning of each cycle, the test system generates a set of values for the input parameters of the CSUT (control subsystem under test). The mediator component sends the generated set of parameter values to the set-mediator subsystem which initializes the input parameters of the CSUT with them. After the CSUT has finished its execution, the get-mediator subsystem reads the values of the output parameters and accessible state variables of the CSUT, gets the system time at the current cycle and transfers all the data to the mediator component. The test system verifies the behavior of the CSUT at the current cycle and the whole composition goes on to the next cycle.

Requirements are formalized in UniTESK in the form of a precondition and a postcondition as implicit specification. Meanwhile, LLR usually transform inputs to outputs explicitly. Development of an explicit behavioral model from LLR is easier than development of an implicit model therefore a new component named model is introduced into the test system architecture which implements the behavioral model of the CSUT explicitly. More precisely, the model is intended to produce reference values for the input parameters and the state variables of the CSUT at each cycle on the basis of the state and the input parameters of the CSUT in the model representation. The postcondition uses the referenced values produced by the model to compare them with the values produced by the CSUT.

4.2 Model

The model is intended to represent LLR in a formal way. Such a representation usually preserves the structure of the branch instructions fixed in LLR. Each decision control algorithm from LLR is mapped

to a separate function. One function is designed to be the entry point to the model. Let us call it *interface function of the model*. A model representation is developed for the input and output parameters and state variables of the CSUT. The input parameters in the model representation become part of the input parameters of the interface function of the model. The output parameters in the model representation become part of the output parameters of the interface function. The state variables in the model representation become part of both the input and output parameters of the interface function.

The input parameters and state variables of the CSUT in the model representation are not enough to evaluate temporal conditions in the model therefore the arguments of the interface function of the model must be complemented with some new parameters. By way of example, let us design a model representation for the temporal conditions used in the iron automatic shut-off control subsystem and complement the arguments of the interface function with parameters that would be enough to evaluate the modeled conditions. Here are the temporal conditions in the flow-chart of the iron automatic shut-off control subsystem:

1. $(!move\&\&!position, 60) = (!move, 60)\&\&(!position, 60)$;
2. $(!move\&\&position, 15*60) = (!move, 15*60)\&\&(position, 15*60)$.

The list of different temporal predicates assigned to a unique identifier where each predicate is a member of a temporal condition:

1. $move\_eq\_f\_t1 = (!move, 60)$;
2. $position\_eq\_f\_t1 = (!position, 60)$;
3. $move\_eq\_f\_t2 = (!move, 15*60)$;
4. $position\_eq\_t\_t2 = (position, 15*60)$.

Listing 1 represents a structure type which combines the identifiers of the temporal predicates. The arguments of the interface function of the model must be complemented with a parameter of this type. Let us call this parameter $time\_flags$.

Listing 1: Structure type combining identifiers of temporal predicates

```
typedef struct{
  bool move_eq_f_t1;
  bool move_eq_f_t2;
  bool position_eq_f_t1;
  bool position_eq_t_t2;
} t_time_flags;
```

Listing 2 represents the source code of the model for the iron automatic shut-off control subsystem. The two temporal conditions are formalized on the basis of the identifiers of the temporal predicates. The value for the parameter $time\_flags$ is calculated by the mediator component on the basis of the previous interactions with the CSUT. This question is discussed later.

Listing 2: Model for iron automatic shut-off control subsystem

```
bool model(bool position, t_time_flags time_flags){
  if(position)
    if(time_flags.move_eq_f_t2 && time_flags.position_eq_t_t2)
      return false;
    else return true;
  else
    if(time_flags.move_eq_f_t1 && time_flags.position_eq_f_t1)
      return false;
    else return true;
}
```

4.3 Specification

The specification contains only one specification function because control subsystems of RTES are considered to provide one interface function. The input and output parameters of the CSUT in the model representation become the input and output parameters of the specification function accordingly. The state of the specification includes the state variables of the CSUT in the model representation. If temporal conditions are used in the branch instructions of the CSUT, the state of the specification is complemented with variables of the data types designed to evaluate the temporal conditions in the model.

Let us consider the postcondition of the specification function. At first, the interface function of the model is called to get referenced values for the output parameters and state variables of the CSUT in the model representation. Those parameters of the interface function of the model which correspond to the state variables of the CSUT get the pre-values of their counterparts in the specification state. Those parameters which correspond to temporal conditions get the post-values of their counterparts in the specification state. The verdict in the postcondition is returned on the basis of the compare of the output parameters and the state variables of the CSUT in the model representation with their referenced counterparts calculated by the model. Listing 3 contains the specification for the iron automatic shut-off control subsystem.

Listing 3: Specification for iron automatic shut-off control subsystem

```
t_time_flags time_flags;
specification bool spec(bool move, bool position) writes time_flags{
    post {return spec == model(position, time_flags);}
}
```

Some part of HLR can be represented in UniTESK tests as data invariants and extra checks in postconditions. It allows providing verification of HLR using the same LLR-based module-level tests.

UniTESK provides a number of coverage criteria on the basis of the structure of the requirements formalized in specifications [10]. When UniTESK is applied for control subsystems of RTES, it is less labor intensive to measure the coverage of the source code of the model. One can easily use a structural coverage metric based on the control flow graph like the branch coverage, condition coverage, modified condition decision coverage and etc [1, 13].

4.4 Mediators

The previous section describes internals of the specification for control subsystems of RTES. In this section we consider the mediator component for this specification and refine the communication protocol between the mediators.

4.4.1 Communication protocol between mediators

The central part of the mediator is a single mediator function, the signature of which is determined by the signature of the specification function. The mediator function transforms the model representation of the input parameters of the CSUT into the representation of the CSUT, sends the values of the transformed parameters to the set-mediator subsystem and goes to the idle mode where it waits for the values of the output parameters and state variables of the CSUT and the system time at the current cycle. At each cycle the set-mediator subsystem starts and finishes its execution before the control subsystem has started. At first the set-mediator subsystem goes to the idle mode where it waits for values of the input parameters of the CSUT. After the values have been received, the set-mediator subsystem initializes

the input parameters of the CSUT with them and finishes its execution. After the control subsystem has finished its execution, the get-mediator subsystem starts. At first it reads the values of the output parameters and accessible state variables of the CSUT. It also reads the value of the system time at the current cycle or calculates it. The get-mediator subsystem sends all the collected data to the mediator component and finishes its execution. The mediator function receives the data sent by the get-mediator subsystem. The values of the output parameters of the CSUT are transformed into the representation of the model. They will be returned as the return value of the mediator function at the end of its execution. Before doing this the mediator function starts to synchronize the specification state with the state of the CSUT.

4.4.2 Synchronization of specification state with implementation state

The state variables of the CSUT received from the get-mediator subsystem are transformed into the representation of the model. After that they are assigned to the specification state variables which model the state variables of the CSUT. Some state variables of the CSUT may not be accessible for reading to other subsystems including the get-mediator subsystem. The model representation for the inaccessible state variables is evaluated according to the principle of working with the hidden state, i.e. in the assumption that the CSUT operates without errors in compliance with the specification.

The mediator component should also calculate values for the specification state variables used to model temporal conditions. By way of example, listing 4 contains an algorithm which evaluates the temporal predicate $(p == VAL, T)$. The mediator function is supposed to execute the algorithm at each cycle. The algorithm uses the following labels: p is an input parameter of the CSUT, VAL is a value of p, T is a time interval during which p should preserve VAL, sys_time is the system time at the current cycle, p_sys_time is the system time since which p has preserved VAL or -1 if p is not equal to VAL.

Listing 4: Algorithm evaluates temporal predicate

```
int p_sys_time = -1;
int get_p_eq_VAL_T(int p, int VAL, int T, int sys_time, int* p_sys_time){
  if(p != VAL) {*p_sys_time = -1;  return 0;}
  else{
    if(*p_sys_time == -1) *p_sys_time = sys_time;
    if(sys_time    *p_sys_time >= T) return 1;
    else return 0;
  }
}
```

4.4.3 Control of cycle period

The set-mediator subsystem goes to the idle mode at the first step of its execution. Being in the idle mode it is waiting for values for the input parameters of the CSUT. It is not ruled out that this delay becomes too long at a cycle so that the total time of all subsystems execution will exceed the cycle period. In this case the behavior of the RTES software is considered incorrect by reason which does not depend on the RTES software. Let us propose two solutions for the problem specified.

The first solution is the following. The cycle period can be increased so that the total time of all subsystems execution would not exceed the cycle period. In order to enhance the accuracy of the method, the test system might control that the total time does not exceed the cycle period at each cycle. The test system will indicate at the end of the testing process whether this condition has hold or not. If it has hold,

the tester may start to analyze the test reports produced. If the condition has not hold, the cycle period must be increased.

The second solution requires that the RTES software implements two features: the *streaming mode* and the system time accessible for writing to subsystems. The streaming mode makes the next cycle start immediately after the last subsystem has finished its execution. The streaming mode implies that the cycle period is not controlled and the subsystems may exceed it without any reaction of the RTES engine. The RTES software usually implements the streaming mode as it is required to easy debug the RTES software or it can be implemented exclusively to facilitate testing. If the system time is accessible for writing, one can implement a subsystem which starts at the beginning of each cycle and sets the system time at the current cycle to the sum of the system time at the previous cycle and the cycle period. From one hand, the two features can be used to simulate usual conditions of operation when each cycle starts right after the cycle period has expired. On the other hand, the total time of all subsystems execution cannot exceed the cycle period.

4.5 Test scenario

Development of test scenarios is seemed to be the most complicated task. Let us explain it in this section by an example, i.e. by developing principal parts of a test scenario for the iron automatic shut-off control subsystem. At first, the target coverage criterion should be established. Let it be the branch coverage of the model. The main goal of the test scenario developer consists in designing such an abstract automaton, traversal of which by the test engine satisfies the target coverage criterion.

4.5.1 States of abstract automaton

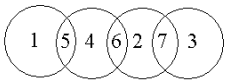
If the number of possible states of the specification is not high, i.e. several hundreds, the state generation function of the test scenario may take the current state of the specification as the state of the abstract automaton. If the number of possible states of the specification is dramatically high or even infinite, the approach of state generalization is used in UniTESK to construct the states of the abstract automaton. According to this approach the states of the specification are partitioned into equivalence classes which are used as the states of the abstract automaton. A universal method called coverage-targeted reduction of the model [14] is used to design generalized states of the abstract automaton and test actions in scenario functions so that the traversal of the abstract automaton would satisfy the target coverage criterion. This method has already been applied in case of coverage criteria on the basis of the structure of the requirements in the specification. Here we give an example of its use in case of a structural criterion on the basis of the control flow graph, i.e. the target branch coverage of the model.

At the first step, a set of test cases is extracted. This set of test cases should satisfy the target coverage criterion. Using the flow-chart of the iron automatic shut-off control subsystem, one can find that the following test cases satisfy the branch coverage of the model:

1. $position \&\& (!move \&\& position, 15 * 60)$;
2. $position \&\& !(!move \&\& position, 15 * 60)$;
3. $!position \&\& (!move \&\& !position, 60)$;
4. $!position \&\& !(!move \&\& !position, 60)$.

At the second step the test cases are considered as conditions which cut out subregions in the space of both possible states of the specification and possible values of the input parameters of the specification function:

Figure 4: Generalized states of abstract automaton



1. $position \&\& move\_eq\_f\_t2 \&\& position\_eq\_t\_t2$;
2. $position \&\& !(move\_eq\_f\_t2 \&\& position\_eq\_t\_t2)$;
3. $!position \&\& move\_eq\_f\_t1 \&\& position\_eq\_f\_t1$;
4. $!position \&\& !(move\_eq\_f\_t1 \&\& position\_eq\_f\_t1)$.

At the third step the subregions are projected on the space of possible states of the specification. Each projection extracts a subspace of states in which the corresponding test case is able to occur. In other words, a set of values for the input parameters may be found which covers the test case in each state from the projection. The conditions extracted at the previous step are projected on the following subspaces of states:

1. $move\_eq\_f\_t2 \&\& position\_eq\_t\_t2$;
2. $!(move\_eq\_f\_t2 \&\& position\_eq\_t\_t2)$;
3. $move\_eq\_f\_t1 \&\& position\_eq\_f\_t1$;
4. $!(move\_eq\_f\_t1 \&\& position\_eq\_f\_t1)$.

The projections may have nonempty intersections. In other words, there are states in which several test cases may be covered depending on the values of the input parameters. At the fourth step all different intersections of the projections are taken. Figure 4 outlines this operation on the projections obtained at the previous step. The intersections are numbered from 5 to 7. The final set of generalized states partitions the whole space of states into equivalence classes because two calls to the specification function in different states from the same generalized state result in the same set of test cases being covered.

The state generation function must not explicitly take different intersections of the projections on the space of states. The current state of the specification and the set of conditions which specify the projections are enough to build the generalized state of the abstract automaton by returning the vector $(\alpha_1, \alpha_2, \ldots, \alpha_n) \in \{0,1\} : \alpha_i = 1$ if the specification state belongs to projection i; otherwise $\alpha_i = 0$.

Extraction of conditions which specify the projections of test cases to the state space of the specification may be a really complicated task. Long-term experience in development of test scenarios for software systems of different classes was investigated in the paper [15]. An interesting idea consists in extracting states of the abstract automaton on the basis of the data structures which model the state of the SUT in the specification. The paper describes some approaches to development of test scenarios. Each approach is based on a widely used data structure. The approaches are represented in the form of design patterns. The statistics collected during that research conforms that the collected patterns are used in more than 80% of the test scenarios investigated.

4.5.2 Test actions

Let the states of the abstract automaton have been developed with the method of coverage-targeted reduction of the model. The method of coverage-targeted reduction of the model describes the way in which the test actions in scenario functions must be defined. The test actions are considered as the equivalence classes of calls to the specification function by the target coverage criterion. In other words, one test

action corresponds to a set of calls to the specification function where each call covers the same coverage item selected by the coverage criterion.

Any alternative method of defining test actions must result in the target coverage criterion satisfied. Having this in mind, we recommend developing scenario functions of two types. Scenario functions of type 1 make reachable states of the abstract automaton needed to satisfy the target coverage criterion. Scenario functions of type 2 iterate values of the parameters of the specification function in each state to maximize the value of the target coverage metric. This approach is effective in practice and is more intuitive.

4.5.3 Restrictions imposed by test engine

We have focused so far on the problem of designing such an abstract automaton, traversal of which would satisfy the target coverage criterion. It is not the only problem which usually appears during development of test scenarios. Restrictions imposed by the test engines on the final abstract automaton have to be in mind during development of both states and test actions. It is difficult to control the restrictions during development of the test scenario because the restrictions are imposed on the final abstract automaton but the abstract automaton of the SUT is being dynamically constructed during the testing process. The most complicated property to satisfy in practice is the determinacy of the abstract automaton and the determinacy of the spanning subautomaton. The method of coverage-targeted reduction of the model does not ensure that the final abstract automaton will satisfy one of these properties. The following methods can be used to modify the abstract automaton so that it would satisfy the determinacy or determinacy of the spanning subautomaton [14, 15]: *splitting of states, injection of connective transitions, generalization of transitions*. These methods do not directly concern control subsystems of RTES. We do not consider them in this paper.

4.5.4 Test sequence reduction methods

The total time of execution of test scenarios may be dramatically high in some cases. This happens because the number of test actions applied during execution of test scenarios is high. It should be noted here that one test action specified in a scenario function may be applied many times during execution of the test scenario because the traversal algorithm implemented in the test engine applies known test actions to get to a state where there are some test actions not applied yet. Decreasing the number of test actions is considered as a big problem because the target coverage criteria must be ultimately satisfied whatever is happened with the abstract automaton.

The following methods are suggested to solve the problem: *testing piecemeal, filtering of input parameters in scenario functions, enlargement of states*. These methods are based on a practical observation and at the same time a logical consideration according to which reducing the number of both possible states of the abstract automaton and test action specifications in scenario functions results in decreasing the number of test actions really applied during execution of the test scenario. We do not formalize the methods in this paper and do not formally proof their effectiveness, but describe them informally and argument why they are useful and usually deliver the expected result in practice.

Testing piecemeal. According to this method the CSUT is partitioned into logical parts. Each part is tested by a separate test scenario. Parts can be algorithms, flow-charts, parts of algorithms or even separate branch instructions. It depends on the complexity of the CSUT and the target coverage criteria. The test scenario for a part of the CSUT is expected not to define more states and test action specifications in scenario functions than the test scenario for the CSUT as a whole. Testing piecemeal results in the

input parameters as well as the states of the CSUT gone over piecemeal therefore the total number of test actions performed by the test scenarios for the parts of the CSUT is expected not to exceed the length of the test sequence generated by the test scenario for the CSUT as a whole. If possible, the test scenarios for the parts of the CSUT can be run simultaneously to further reduce the total time of execution.

Implementation of the method implies that the test scenario takes control over the data and control flows in the CSUT. Some input parameters are set to values which ensure that the data and control flows reach the target part of the CSUT, the other parameters are gone over to satisfy the coverage of the target part of the CSUT.

Some coverage criteria do not allow for testing piecemeal because all structure of the SUT or requirements must be taken into account to satisfy the criteria. A typical example is MCC (Multiple Condition Coverage) [1].

Filtering of input parameters in scenario functions. Filters are conditions on a set of variables which filter out some sets of their values. The method reduces the number of test action specifications in scenario functions by applying filters in iteration statements to filter out those set of values for the input parameters of the CSUT which do not enlarge the coverage. The syntax rules for filters in iteration statements can be found in [10].

Enlargement of states. The goal of the method consists in reducing the number of states of the abstract automaton by re-generalizing the specification states so that the new set of generalized states would contain more specification states than the prevous set on the average. Let a test scenario is implemented for the iron automatic shut-off subsystem and the states of the specification act as the states of the abstract automaton. We can remember that the state of the specification is determined by the values of four temporal predicates therefore the upper bound for the number of possible states of the abstract automaton is 16. We obviously might calculate that only 9 states are really possible. As we showed the method of coverage-targeted reduction of the model results in 7 states of the abstract automation. Both 9 and 7 states cover the whole space of possible states of the specification. It is a true fact that we have constructed 7 states that are larger that 9 original states on the average.

4.6 Practical results

All solutions described in this paper were implemented and applied in projects on testing of avionics related control subsystems. As an example, we provide some details of our joint project with Russian System Corporation, where a system under test was the control subsystem of AAFSS (Airborne Active Flight Safety System) developed by Russian System Corporation.

AAFSS is designed to increase flight safety and effectiveness of the airborne complex. The system performs monitoring of the operational status of the airborne systems, survival facilities, operational conditions and adequacy of the behavior of the crew, as well as decision control on recovery of flight safety in critical situations.

The AAFSS software consists of a number of subsystems. The control subsystem is the most critical one because it takes all decisions in AAFSS. The control subsystem consists of about 10 decision control algorithms, 30 input parameters, 10 state variables, 10 output parameters and 80 temporal predicates. The functional requirements to the control subsystem are well-structured and carefully described mainly in flow-charts.

The numbers of the input parameters, state variables and temporal predicates do not give an alternative to the method of testing piecemeal. 15 test scenarios were developed for the control subsystem of AAFSS. Each scenario performs about 58000 test actions on the average. Some critical bugs were revealed by those test scenarios in development versions of control software.

We got added evidence that an important advantage of a model based testing technology complemented with a strong process of formalization of requirements consists in possibility to reveal bugs at earlier stages [7]. In particular, some problems concerning correctness, unambiguity and completeness [16, 17] of the decision control algorithms were revealed.

5 Conclusions

The paper presents our experience in model based testing of control subsystems of safety critical RTES using the UniTESK testing technology. The input for testing process is LLR that describe implementation details of the control logic in an informal way. This description is usually similar to flow charts enriched by timing properties. The tests to be developed have to cover all LLR and have to provide appropriate source code coverage. The latter means that the tests have to cover all the decisions and/or conditions that basically inherited from LLR.

In these settings model based testing techniques demonstrate themselves as an efficient means to achieve the required coverage level in a semi-automated way. Basically, the UniTESK model based testing approach suggests to formalize LLR and then to take benefits from the automation of the verification activities on the base of this model. Formalization of LLR allows to reveal issues in LLR themselves. Additionally, UniTESK allows to provide verification of HLR on the base of the same LLR-based module-level tests.

The next logical step could be to formalize LLR from the very beginning using some formal notation. The possible benefits of this step include:

- generation of source code from LLR;
- generation of UniTESK models from LLR if testing against LLR is required;
- earlier bug revealing in LLR during the formalization step;
- partial automation of LLR verification against HLR.

There are already several tools available that support formalization of LLR. The most known of them are SCADE and Simulink. Qualified code generators have been developed for the models produced by these tools. If qualified code generators are available in a project, there is no need in LLR-based tests. In this case model based testing techniques can be valuable to automate test generation in the context of HLR verification. If qualified tools are not available in a project, LLR-based tests described above are still required for certification purposes. In this case most parts of the UniTESK test suite could be generated from LLR except for the test scenarios, where manual tuning is required to prevent the state explosion.

References

[1] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J. & Rierson Leanna K.. 2001. *A Practical Tutorial on Modified Condition/Decision Coverage*. NASA Langley Research Center. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.1317`.

[2] Bahareh Badban, Martin Franzle, Jan Peleska & Tino Teige. 2006. *Test Automation for Hybrid Systems*. In *Proceedings of the Third International Workshop on Software Quality Assurance (SOQUA 2006)*. ACM, New York, NY, USA, pp. 14–21. doi:`10.1145/1188895.1188902`.

[3] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson & Arne Skou. 2008. *Testing real-time systems using UPPAAL*. In *Formal methods and testing*. Lecture Notes In Computer Sci-

ence, Vol. 4949. Springer-Verlag, Berlin, Heidelberg, pp. 77-117. isbn:3-540-78916-2, 978-3-540-78916-1. doi:10.1007/978-3-540-78917-8_3.

[4] Henrik Bohnenkamp & Axel Belinfante. 2005. *Timed testing with torx*. In *Proceedings of the International Symposium of Formal Methods Europe*. Lecture Notes In Computer Science, Vol. 3582. Springer-Verlag, Berlin, Heidelberg, pp. 173-188. doi:10.1007/11526841_13.

[5] Moez Krichen & Stavros Tripakis. 2009. *Conformance testing for real-time systems*. Formal Methods in System Design 34, 3, pp. 238–304. doi:10.1007/s10703-009-0065-1.

[6] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann & Lev Nachmanson. 2008. *Model-based testing of object-oriented reactive systems with spec explorer*. In *Formal methods and testing*. Lecture Notes In Computer Science, Vol. 4949. Springer-Verlag, Berlin, Heidelberg, pp. 39-76. isbn:3-540-78916-2, 978-3-540-78916-1. doi:10.1007/978-3-540-78917-8_2.

[7] A. Grinevich, A. Khoroshilov, V. Kuliamin, D. Markovtsev, A. Petrenko & V. Rubanov. 2006. *Formal Methods in Industrial Software Standards Enforcement*. In *Proceedings of PSI'2006*. Novosibirsk, Russia, pp. 26–30. doi:10.1007/978-3-540-70881-0_41.

[8] Igor Bourdonov, Alexander Kossatchev, Victor Kuliamin & Alexander Petrenko. 2002. *UniTesK Test Suite Architecture*. In *Proceedings of the International Symposium of Formal Methods Europe*. Lecture Notes In Computer Science, Vol. 2391. Springer-Verlag, Berlin, Heidelberg, doi:10.1007/3-540-45614-7_5.

[9] Bertrand Meyer. 1992. *Applying "Design by Contract"*. Computer 25, 10, pp. 40–51. doi:10.1109/2.161279.

[10] *CTesK 2.2 Users Guide*. Available at http://www.unitesk.com/download/papers/ctesk/ce/CTesK2.2CEUserGuide.eng.pdf.

[11] Igor Bourdonov, Alexander Kossatchev, Victor Kuliamin. 2003. *Irredundant traversal algorithms for oriented graphs: deterministic case*. Programming and Computer Software 25, 5, pp. 59–69. Available at http://panda.ispras.ru/~kuliamin/docs/Graphs-2003-ru.pdf (in Russian).

[12] Igor Bourdonov, Alexander Kossatchev, Victor Kuliamin. 2004. *Irredundant traversal algorithms for oriented graphs: nondeterministic case*. Programming and Computer Software 30, 1, pp. 2–17. Available at http://panda.ispras.ru/~kuliamin/docs/Graphs-2004-ru.pdf (in Russian).

[13] Hong Zhu, Patric A. V. Hall, John H.R. May. 1997. *Software Unit Test Coverage and Adequacy*. ACM Computing Surveys (CSUR) 29, 4, pp. 366–427. doi:10.1145/267580.267590.

[14] Igor Bourdonov, Alexander Kossatchev, Victor Kuliamin. 2000. *Use of finite state machines for testing programs*. Programming and Computer Software 26, 2, pp. 61–73. Available at http://panda.ispras.ru/~kuliamin/docs/FSM-2000-ru.pdf (in Russian).

[15] Vadim Mutilin. 2006. *Design patterns for test scenarios*. In *Proceedings of ISP RAS*. Vol. 9. ISP RAS, Moscow, Russia, pp. 97–128. Available at http://ispras.ru/ru/proceedings/docs/2006/9/isp_9_2006_97.pdf (in Russian).

[16] *IEEE standard 830-1998 IEEE Recommended Practice for Software Requirements Specifications*. Available at http://standards.ieee.org/findstds/standard/830-1998.htmlf.

[17] *19. IEEE standard 1233-1996 IEEE Guide for Developing System Requirements Specifications*. Available at http://standards.ieee.org/findstds/standard/1233-1996.html.