EPTCS 111

Proceedings of the

Eighth Workshop on

Model-Based Testing

Rome, Italy, 17th March 2013

Edited by: Alexander K. Petrenko and Holger Schlingloff

Published: 2nd March 2013
DOI: 10.4204/EPTCS.111
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the proceedings of the Eighth Workshop on Model-Based Testing (MBT 2013), which was held in Rome on March 17, 2013 as a satellite workshop of the European Joint Conferences on Theory and Practice of Software (ETAPS 2013).

The first workshop on Model-Based Testing (MBT) in this series took place in 2004, in Barcelona. At that time model-based testing already had become a hot topic, but MBT 2004 was the first event devoted exclusively to this domain. Since that time the area has generated enormous scientific and industrial interest, and today there are several other workshops and conferences on software and hardware design and quality assurance covering also model based testing. For example, this year ETSI organizes the UCAAT (User Conference on Advanced Automated Testing) with a focus on "model-based testing in the testing ecosystem". Still, the MBT series of workshops offers a unique opportunity to share new technological and foundational ideas particular in this area, 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 methods, 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 formal development methods and verification tools;
- Application of the technology in the certification of safety-critical systems (this includes establishing acknowledged coverage criteria and specification-based test oracles);
- Use of new notations and new kinds of modeling formalisms along with the elaboration of approaches based on usual programming languages and specialized libraries;
- Integration of model-based testing into continuous development processes and environments (e.g., for software product lines).

The invited talk and paper of Jan Peleska in this volume gives a nice survey of current challenges. Furthermore, the submitted contributions, selected by the program committee, reflect the above research trends. Isabel Nunes and Filipe Luís consider the integration of model-based testing with algebraic specifications for the testing of Java programs. Neda Noroozi, Mohammad Reza Mousavi and Tim A.C. Willemse analyze criteria for the decomposability of models in the theory of input-output conformance (ioco) testing. Mikhail Chupilko and Alexander Kamkin extend model-based testing to runtime verification: they develop an online algorithm for conformance of timed execution traces with respect to timed automata. Stephan Weissleder and Hartmut Lackner compare different approaches for test generation from variant models and feature models in product line testing.

In 2012 the "industrial paper" category was added to the program. This year we have two accepted industrial presentations, both from the telecommunications domain: Jens Grabowski, Victor Kuliamin, Alain-Georges Vouffo Feudjio, Antal Wu-Hen-Chang and Milan Zoric report on the evaluation of four different model-based testing tools for standardization at ETSI, the European Telecommunications Standards Institute. Olga Grinchtein gave a talk on the experiences gained by the application of model-based testing for base stations of LTE, the European 4G mobile phone network.

We would like to thank the program committee members and all reviewers for their work in evaluating the submissions. We also thank the ETAPS 2013 organizers for their assistance in the preparation of the workshop and the EPTCS editors for help in publishing these proceedings.

Alexander K. Petrenko and Holger Schlingloff, February 2013.

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 (Universitat des Saarlandes, Germany)
- Lars Frantzen (Radboud University Nijmegen , Netherlands)
- 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)
- Bruno Legeard (Smartesting, France)
- Bruno Marre (CEA LIST, France)
- Laurent Mounier (VERIMAG, France)
- Alexander K. Petrenko (ISP RAS, Russia)
- Alexandre Petrenko (Computer Research Institute of Montreal, Canada)
- Fabien Peureux (University of Franche-Comte, France)
- Holger Schlingloff (Fraungofer FIRST, Germany)
- Julien Schmaltz (Open University of The Netherlands, Netherlands)
- Nikolai Tillmann (Microsoft Research, USA)
- Stephan Weissleder (Fraunhofer FOKUS, Germany)
- Nina Yevtushenko (Tomsk State University, Russia)

Additional reviewers

- Igor Burdonov (ISP RAS, Russia)
- Maxim Gromov (Tomsk State University, Russia)

Industrial-Strength Model-Based Testing - State of the Art and Current Challenges*

Jan Peleska
University of Bremen, Department of Mathematics and Computer Science, Bremen, Germany
Verified Systems International GmbH, Bremen, Germany
jp@informatik.uni-bremen.de

As of today, model-based testing (MBT) is considered as leading-edge technology in industry. We sketch the different MBT variants that – according to our experience – are currently applied in practice, with special emphasis on the avionic, railway and automotive domains. The key factors for successful industrial-scale application of MBT are described, both from a scientific and a managerial point of view. With respect to the former view, we describe the techniques for automated test case, test data and test procedure generation for concurrent reactive real-time systems which are considered as the most important enablers for MBT in practice. With respect to the latter view, our experience with introducing MBT approaches in testing teams are sketched. Finally, the most challenging open scientific problems whose solutions are bound to improve the acceptance and effectiveness of MBT in industry are discussed.

1 Introduction

1.1 Model-Based Testing

Following the definition currently given in Wikipedia[1]

> "*Model-based testing is application of Model based design for designing and optionally also executing artifacts to perform software testing. Models can be used to represent the desired behavior of an System Under Test (SUT), or to represent testing strategies and a test environment.*"

In this definition only software testing is referenced, but it applies to hardware/software integration and system testing just as well. Observe that this definition does not require that certain aspects of testing – such as test case identification or test procedure creation – should be performed in an automated way: the MBT approach can also be applied manually, just as design support for testing environments, test cases and so on. This rather unrestricted view on MBT is consistent with the one expressed in [2], and it is reflected by today's MBT tools ranging from graphical test case description aides to highly automated test case, test data and test procedure generators. Our concept of models also comprises computer programs, typically represented by per-function/method control flow graphs annotated by statements and conditional expressions.

Automated MBT has received much attention in recent years, both in academia and in industry. This interest has been stimulated by the success of model-driven development in general, by the improved understanding of testing and formal verification as complementary activities,

*The author's research is funded by the EU FP7 COMPASS project under grant agreement no.287829
[1] http://en.wikipedia.org/wiki/Model-based_testing, (date: 2013-0211).

A. Petrenko, H. Schlingloff (Eds.): Eighth Workshop on Model-Based Testing (MBT 2013)
EPTCS 111, 2013, pp. 3–28, doi:10.4204/EPTCS.111.1

and by the availability of efficient tool support. Indeed, when compared to conventional testing approaches, MBT has proven to increase both quality and efficiency of test campaigns; we name [21] as one example where quantitative evaluation results have been given.

In this paper the term model-based testing is used in the following, most comprehensive, sense: the behaviour of the *system under test (SUT)* is specified by a model elaborated in the same style as a model serving for development purposes. Optionally, the SUT model can be paired with an environment model restricting the possible interactions of the environment with the SUT. A *symbolic test case generator* analyses the model and specifies *symbolic test cases* as logical formulas identifying model computations suitable for a certain test purpose. Constrained by the transition relations of SUT and environment model, a *solver* computes concrete model computations which are *witnesses* of the symbolic test cases. The inputs to the SUT obtained from these computations are used in the test execution to stimulate the SUT. The SUT behaviour observed during the test execution is compared against the *expected* SUT behaviour specified in the original model. Both stimulation sequences and *test oracles*, i. e., checkers of SUT behaviour, are automatically transformed into *test procedures* executing the concrete test cases in a model-in-the-loop, software-in-the-loop, or hardware-in-the-loop configuration.

According to the MBT paradigm described here, the focus of test engineers is shifted from test data elaboration and test procedure programming to modelling. The effort invested into specifying the SUT model results in a return of investment, because test procedures are generated automatically, and debugging deviations of observed against expected behaviour is considerably facilitated because the observed test executions can be "replayed" against the model. Moreover, V&V processes and certification are facilitated because test cases can be automatically traced against the model which in turn reflects the complete set of system requirements.

1.2 Objectives of this Paper

The objective of this paper is to describe the capabilities of MBT tools which – according to our experience – are fit for application in today's industrial scale projects and which are essential for successful MBT application in practice. The MBT application field considered here is distributed embedded real-time systems in the avionic, automotive and railway domains. The description refers to our tool RT-Tester[2] for illustrating several aspects of MBT in practice, and the underlying methods that helped to meet the test-related requirements from real-world V&V campaigns. The presentation is structured according to the MBT researchers' and tool builders' perspective: we describe the ingredients that, according to our experience, should be present in industrial-strength test automation tools, in order to cope with test models of the sizes typically encountered when testing embedded real-time systems in the automotive, avionic or railway domains. We hope that these references to an existing tool may serve as "benchmarking information" which may motivate other researchers to describe alternative methods and their virtues with respect to practical testing campaigns.

[2]The tool has been developed by Verified Systems International in cooperation with the author's team at the University of Bremen. It is available free of charge for academic research, but commercial licenses have to be obtained for industrial application. Some components (e.g., the SMT solver) will also become available as open source.

1.3 Outline

In Section 2 a tool introduction is given. In Section 3, MBT methods and challenges related to modelling are discussed. Section 4 introduces a formal view on requirements, test cases and their traceability in relation to the test model. It also discusses various test strategies and their justification. A case study illustrating various points of our discussion of MBT is described in Appendix A. Section 5 presents the conclusion. We give references to alternative or competing methods and tools along the way, as suitable for the presentation.

2 A Reference MBT Tool

RT-Tester supports all test levels from unit testing to system integration testing and provides different functions for manual test procedure development, automated test case, test data and test procedure generation, as well as management functions for large test campaigns. The typical application scope covers (potentially safety-critical) embedded real-time systems involving concurrency, time constraints, discrete control decisions as well as integer and floating point data and calculations. While the tool has been used in industry for about 15 years and has been qualified for avionic, automotive and railway control systems under test according to the standards [33, 20, 38], the results presented here refer to more recent functionality that has been validated during the last years in various projects from the transportation domains and are now made available to the public.

The starting point for MBT is a concrete test model describing the expected behaviour of the system under test (SUT) and, optionally, the behaviour of the operational environment to be simulated in test executions by the testing environment (TE) (see Fig. 1). Models developed in a specific formalism are transformed into some textual representation supported by the modelling tool (usually XMI format). A model parser front-end reads the model text and creates an internal model representation (IMR) of the abstract syntax.

A transition relation generator creates the initial state and the transition relation of the model as an expression in propositional logic, referring to pre-and post-states. Model transformers create additional reduced, abstracted or equivalent model representations which are useful to speed up the test case and test data generation process.

A test case generator creates propositional formulas representing test cases built according to a given strategy. A satisfiability modulo theory (SMT) solver calculates solutions of the test case constraints in compliance with the transition relation. This results in concrete computation fragments yielding the time stamps and input vectors to be used in the test procedure implementing the test case (and possibly other test cases as well). An interpreter simulating the model in compliance with the transition relation is used to investigate concrete model executions continuing the computation fragments calculated by the SMT solver or, alternatively, creating new computations based on environment simulation and random data selection. An abstract interpreter supports the SMT solver in finding solutions faster by calculating the minimum number of transition steps required to reach the goal, and by restricting the ranges of inputs and other model variables for each state possibly leading to a solution. Finally, the test procedure generator creates executable test procedures as required by the test execution environment by mapping the computations derived before into time-controlled commands sending input data to the SUT and by creating test oracles from the SUT model portion checking SUT reactions on the fly, in dependency of the stimuli received before from the TE.

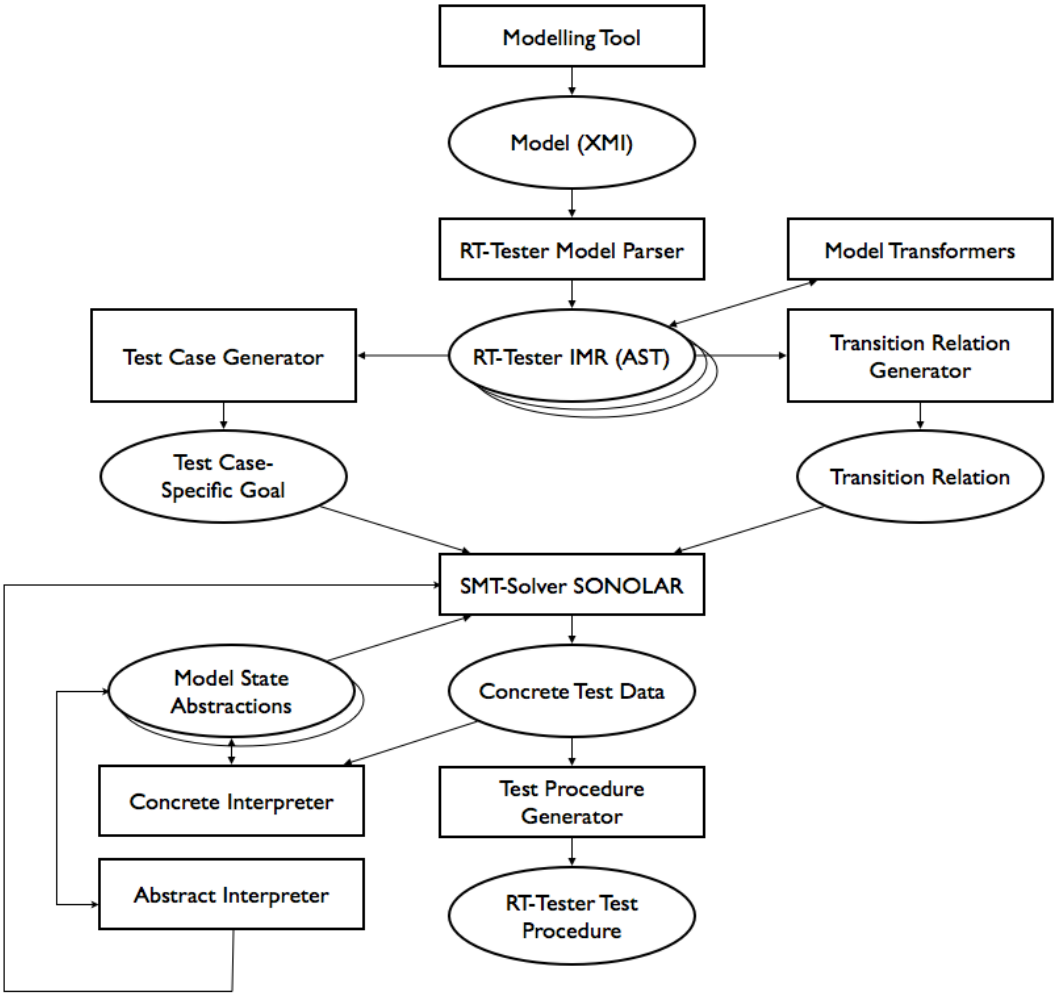


Figure 1: Components of the RT-Tester test case/test data generator.

3 Modelling Aspects

3.1 Modelling Formalisms

It is our expectation that the ongoing discussions about suitable modelling formalisms for reactive systems – from UML via process algebras and synchronous languages to domain-specific languages – will not converge to a single preferred formalism in the near future. As a consequence it is important to separate the test case and test data generation algorithms from concrete modelling formalisms.

RT-Tester supports subsets of UML [24] and SysML [23] for creating test models: SUT structure is expressed by composite structure or block diagrams, and behaviour is specified by means of state machines and operations (a small SysML-based case study is presented Appendix A). The parser front end reads model exports from different tools in XMI format. Another parser reads Matlab/Simulink models. For software testing, a further front end parses transition graphs of C functions.

The first versions of RT-Tester supported CSP [35] as modelling language, but the process-algebraic presentation style was not accepted well by practitioners. Support for an alternative textual formalism is currently elaborated by creating a front-end for CML [43], the COMPASS modelling language specialised on systems of systems (SoS) design, verification and validation. In CML, the problems preventing a wider acceptance of CSP for test modelling have been removed.

Some formalisms are domain-specific and supported on customers' request: in [21] automated MBT against a timed variant of Moore Automata is described, which is used for modelling control logic of level crossing systems.

3.2 A Sample Model

In Appendix A a case study is presented which will be used in this paper to illustrate modelling techniques, test case generation and requirements tracing. The case study models the turn indication and emergency flashing functions as present in modern vehicles. While this study is just a small simplified example, a full test model of the turn indication function as realised in Daimler Mercedes cars has been published in [26] and is available under `http://www.mbt-benchmarks.org`.

3.3 Semantic Models

In addition to the internal model representation which is capable of representing abstract syntax trees for a wide variety of formalisms, a semantic model is needed which is rich enough to encode the different behaviours of these formalisms. As will be described in Section 4, operational model semantics is the basis for automated test data generation, and it is also needed to specify the conformance relation between test model and SUT, which is checked by the tests oracles generated from the model (see below).

A wide variety of semantic models is available and suitable for test generation. Different variants of labelled transition systems (LTS) are used for testing against process algebraic models, like Hennessy's acceptance tree semantics [14], the failures-divergence semantics of CSP (they come in several variants [30]) and Timed CSP [35], the LTS used in I/O conformance test theory [39, 40], or the Timed LTS used for the testing theory of Timed I/O Automata [37]. As an alternative to the LTS-based approach, Cavalcanti and Gaudel advocate for the Unifying Theories of Programming [15], that are used, for example, as a semantic basis for the Circus formalism and its testing theory [8], and for the COMPASS Modelling Language CML mentioned above.

For our research and MBT tool building foundations we have adopted Kripke Structures, mainly because our test generation techniques are close to techniques used in (bounded) model checking, and many fundamental results from that area are formulated in the semantic framework of Kripke Structures [10]. Recall that a Kripke Structure is a state transition system $K = (S, S_0, R, L)$ with state space S, initial states $S_0 \subseteq S$, transition relation $R \subseteq S \times S$ and labelling function $L : S \to \mathbb{P}(AP)$ associating each state s with the set $L(s)$ of atomic propositions $p \in AP$ which hold in this state. The behaviour of K is expressed by the set of computations $\pi = s_0.s_1.s_2 \ldots \in S^\omega$, that is, the infinite sequences π of states fulfilling $s_0 \in S_0$ and $R(s_i, s_{i+1}), i = 0, 1, 2, \ldots$. In contrast to LTS, Kripke Structures do not support a concept of events, these have to be modelled by propositions becoming `true` when changing from one state to a successor

state. For testing purposes, states $s \in S$ are typically modelled by variable valuation functions $s : V \to D$ where V is a set of variable symbols x mapped by s to their current value $s(x)$ in their appropriate domain (bool, int, float, ...) which is a subset of D. The variable symbols are partitioned into $V = I \cup O \cup M$, where I contains the input variables of the SUT, O its output variables, and M its internal model variables which cannot be observed during tests. Concurrency can be modelled both for the synchronous ("true parallelism") [7] and the interleaving variants of semantics [10, Chapter 10]. Discrete or dense time can be encoded by means of a variable $\hat{t}$ denoting model execution time. For dense-time models this leads to state spaces of uncountable size, but the abstractions of the state space according to clock regions or clock zones, as known from Timed Automata [10] can be encoded by means of atomic propositions and lead to finite-state abstractions.

Observe that there should be no real controversy about whether LTS or Kripke Structures are more suitable for describing behavioural semantics of models: De Nicola and Vaandrager [22] have shown how to construct property-preserving transformations of LTS into Kripke Structures and vice versa.

3.4 Conformance Relations

Conformance relations specify the correctness properties of a SUT by comparing its actual behaviour observed during test executions to the possible behaviours specified by the model. A wide variety of conformance relations are known. For Mealy automata models, Chow used an input/output-based equivalence relation which amounted to isomorphism between minimal automata representing specification and implementation models [9]. in the domain of process algebras Hennessy and De Nicola introduced the relation of *testing equivalence* which related specification process behaviour to SUT process behaviour [11]. For Lotus, this concept was explored in depth by Brinksma [6], Peleska and Siegel showed that it could be equally well applied for CSP and its refinement relations [25], and Schneider extended these results to Timed CSP [34]. Tretmans introduced the concept of I/O conformance [39]. Vaandrager et. al. used bi-similarity as a testing relation between timed automata representing specification and implementation [37]. All these conformance relations have in common, that they are defined on the model semantics, that is, as relations between computations admissible for specification and implementation, respectively.

Conformance in the synchronous deterministic case. For our Kripke structures, a simple variant of I/O conformance suffices for a surprisingly wide range of applications: for every trace[3] $s_0.s_1 \ldots s_n$ identified for test purposes in the model, the associated test execution trace $s'_0.s'_1 \ldots s'_n$ should have the same length and satisfy

$$\forall i \in \{0, \ldots, n\} : s_i|_{I \cup O \cup \{\hat{t}\}} = s'_i|_{I \cup O \cup \{\hat{t}\}}$$

that is, the observable input and output values, as well as the time stamps should be identical.

This very simple notion of conformance is justified for the following scenarios of reactive systems testing: (1) The SUT is non-blocking on its input interfaces, (2) the most recent value passed along output interfaces can always be queried in the testing environment, (3) each concurrent component is deterministic, and (4) the synchronous concurrency semantics applies. At

[3] Traces are finite prefixes of computations.

first glance, these conditions may seem rather restrictive, but there is a wide variety of practical test applications where they apply: many SUT never refuse inputs, since they communicate via shared variables, dual-ported ram, or non-blocking state-based protocols[4]. Typical hardware-in-the-loop testing environments always keep the current output values of the SUT in memory for evaluation purposes, so that even message-based interfaces can be accessed as shared variables in memory (additionally, test events may be generated when an output message of the SUT actually arrives in the test environment (TE). For safety-critical applications the control decisions of each sequential sub-component of the SUT must be deterministic, so that the concept of *may tests* [14], where a test trace may or may not be refused by the SUT does not apply. As a consequence, the complexity and elegance of testing theories handling non-deterministic internal choice and associated refusal sets and unpredictable outputs of the SUT are not applicable for these types of systems. Finally, synchronous systems are widely used for local control applications, such as, for example, PLC controllers or computers adhering to the cyclic processing paradigm.

In RT-Tester this conformance relation is practically applied, for example, when testing software generated from SCADE models [12]: the SCADE tool and its modelling language adhere to the synchronous paradigm. The software operates in processing cycles. Each cycle starts with reading input data from global variables shared with the environment; this is followed by internal processing steps, and the output variables are updated at the end of the cycle. Time $\hat{t}$ is a discrete abstraction corresponding to a counter of processing cycles.

Conformance in presence of non-determinism. For distributed control systems the synchronous paradigm obviously no longer applies, and though single sequential SUT components will usually still act in a deterministic way, their outputs will interleave non-deterministically with those of others executing in a concurrent way. Moreover, certain SUT outputs may change non-deterministically over a period of time, because the exact behavioural specification is unavailable. These aspects are supported in RT-Tester in the following ways.

- All SUT output interfaces y are associated with (1) an acceptable deviation ε_y from the accepted value (so any observed value $s'(y)$ deviating from the expected value $s(y)$ by $|s'(y) - s(y)| \leq \varepsilon$ is acceptable), (2) an admissible latency δ_y^0 (so any observed value $s'(y)$ for the expected value $s(y)$ is not timed out as long as $s'(\hat{t}) - s(\hat{t}) \leq \delta_y^0$, and (3) an acceptable time δ_y^1 for early changes of y (so $s(\hat{t}) - s'(\hat{t}) \leq \delta_y^1$ is still acceptable).
- A time-bounded non-deterministic assignment statement `y = UNDEF(t,c)` stating that y's valuation is arbitrary for a duration of `t` time units, after which it should assume value `c` (with an admissible deviation and an admissible delay).
- A model transformation turning the SUT model into a test oracle: it
  - extends the variable space by one additional output variable y' per SUT output $y \in O$,
  - adds one concurrent checker component $\mathscr{O}_y$ per SUT output signal, operating on y and y',
  - adds one concurrent component $\mathscr{P}$ processing the timed input output trace as observed during the test execution, with observed SUT outputs written to y' (instead of y),

[4] In the avionic domain, for example, the sampling mode of the AFDX protocol [1] allows to transmit messages in non-blocking mode, so that the receiver always reads the most recent data value.

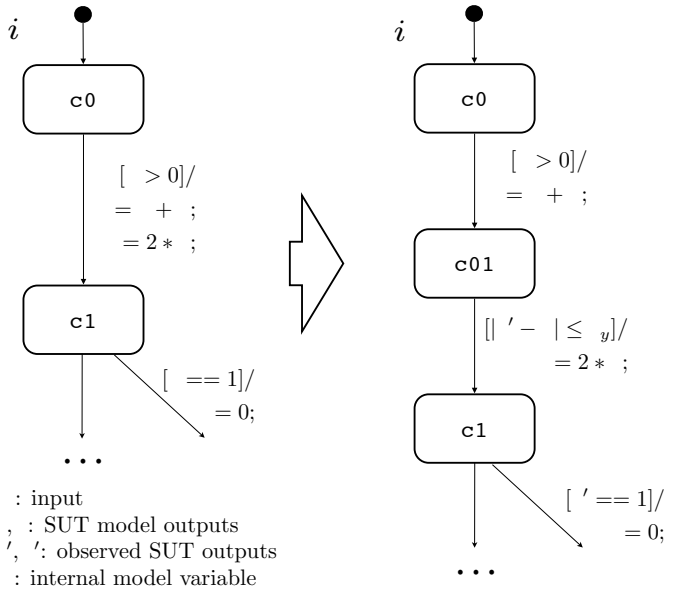


Figure 2: Example of original SUT component C_i and transformed component $\mathscr{C}_i$.

– transforms each concurrent SUT component C_i into $\mathscr{C}_i$.

This is described in more detail in the next paragraphs.

The transformed SUT components $\mathscr{C}_i$ operate as sketched in the example shown in Fig. 2. Every write of C_i to some output y is performed in $\mathscr{C}_i$ as well, $\mathscr{C}_i$ however, waits for the corresponding output value y' observed during test execution to change until it fits to the expected value of y (guard condition $|y'-y| \leq \varepsilon$). This helps to adjust to small admissible delays of in the expected change of y' observed in the test: the causal relation "a is written after y has been changed is preserved in this way. If C_i uses another output z (written, for example, by a concurrent component C_j) in a guard condition, it is replaced by variable z' containing the observed output during test execution. This helps to check for correctness of relative time distances like "output w is written 10ms after z has been changed", if the actual output on z' is delayed by an admissible amount of time.

The concurrent test oracles $\mathscr{O}_y$ operate as shown in Fig. 3: If some component $\mathscr{C}_i$ writes to an expected output y, the oracle traverses into control state `s2`. If the corresponding observed output y' is also adjusted in $\mathscr{P}$, such that $|y'-y| \leq \varepsilon_y$ holds before δ_y^0 time units have elapsed, the change to y' is accepted and the oracle transits to `s0`. Otherwise the oracle transits into the error state. If the observed value changes unexpectedly above threshold ε_y, the oracle changes into location `s3`. If the expected value y also changes shortly afterwards, this means that the SUT was just some admissible time earlier than expected according to the model, and the change is monitored via state `s2` as before. If y, however, does not change for at least δ_y^1 time units, we have uncovered an illegal output change of the SUT and transit into the error state.

A test execution (that is, an input/output trace) performed with the SUT conforms to the model if and only if the transformed model accepts the test execution processed by $\mathscr{P}$ in the sense that none of the oracles transits into an error state. RT-Tester uses this conformance

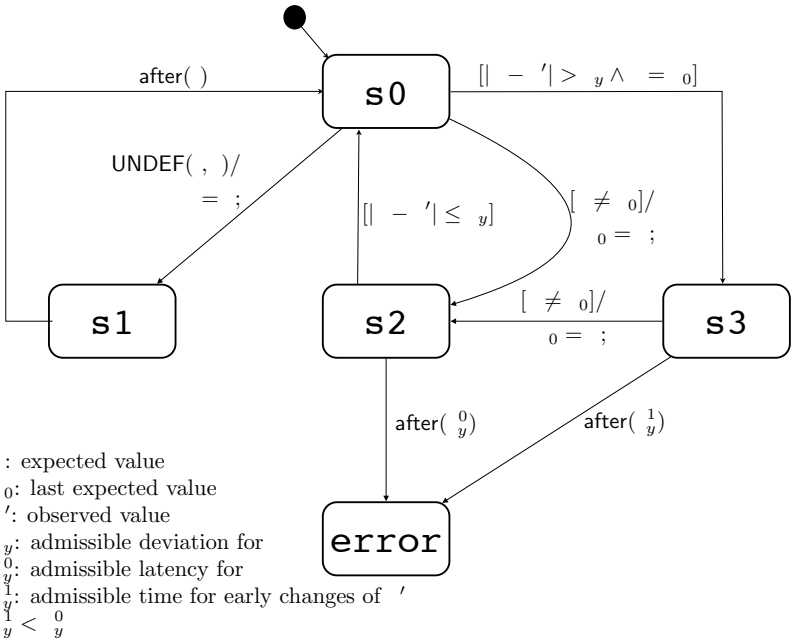


Figure 3: Test oracle component observing one SUT output interface y.

relation for hardware-in-the-loop system testing, as, for example, in the tests of the automotive controller network supporting the turn indication function in Daimler Mercedes vehicles [26].

3.5 Test-Modelling Related Challenges

With suitable test models available, test efficiency and test quality are improved in a considerable way. The elaboration of a model, however, can prove to be a major hurdle for the success of MBT in practice.

1. If complex models have to be completed before testing can start, this induces an unacceptable delay for the proper test executions.
2. For complex SUT, like systems of systems, test models need to abstract from a large amount of detail, because otherwise the resulting test model would become unmanageable.
3. The required skills for test engineers writing test models are significantly higher than for test engineers writing sequential test procedures.

We expect that problem 1 will be solved in the future by incremental model development, where test suites with increasing coverage and error detection capabilities can be run between model increments. The current methods based on sequential state machines as described by [41] may be extended to partially automated approaches where test model designers provide – apart from interface descriptions – initial architectural frames and suggestions for internal state variables, and automated machine learning takes these information into account. Furthermore, the explicit state machine construction may be complemented by incremental elaboration of transition relations: as pointed out by [27] for the purpose of test data generation, concurrent

real-time models with complex state space are often better expressed by means of their transition relation than by explicit concurrent state machines. Promising attempts to construct test models in an incremental way from actual observations obtained during SUT simulations or experiments with the actual SUT indicate that test model development can profit from "re-engineering" SUT properties or model fragments from observations [29].

The problem of model complexity can be overcome by introducing contracts for the constituent systems of a large system of systems. This type of abstractions is investigated, for example, in the COMPASS project[5].

With respect to the third problem it is necessary to point out in management circles that competent testing requires the same skills as competent software development. So if modelling skills are required for model-driven software and system development, these skills are required for test engineers as well.

4 Requirements, Test Cases and Trustworthy Test Strategies

4.1 Requirements

If a test model has been elaborated in an adequate way, it will reflect the requirements to be tested. At first glance, however, it may not be obvious to identify the model portions contributing to a given requirement. Formally speaking, a requirement is reflected by certain computations $\pi = s_0.s_1.s_2\ldots$ of the model. Computations can be identified, for example, by some variant of temporal logic, and we use Linear Temporal Logic (LTL) [10, Chapter 3] for this purpose[6].

Consider, for example, requirement REQ-001 (Flashing requires sufficient voltage) from the sample application specified in Appendix A, Table 1. It can be readily expressed in LTL as

$$\mathbf{G}(\text{Voltage} \leq 80 \Rightarrow \mathbf{X}(\neg(\text{FlashLeft} \vee \text{FlashRight})\ \mathbf{U}\ \text{Voltage} > 80)) \tag{1}$$

This is a black-box specification: it only refers to input and output interfaces of the SUT and is valid without any model. With a model at hand, however, the specification can be slightly simplified, because the relevant SUT reactions have been captured by state machine OUTPUT_CTRL (see Fig. 8)[7].

$$\mathbf{G}(\text{Voltage} \leq 80 \Rightarrow \mathbf{X}(\text{Idle}\ \mathbf{U}\ \text{Voltage} > 80))$$

In control state Idle the indication lights are never activated. Now the computations contributing to REQ-001 are exactly the ones finally fulfilling the premise Voltage ≤ 80, where the effect of the requirement may become visible, that is,

$$\mathbf{F}(\text{Voltage} \leq 80)$$

It is unnecessary to specify the effects of the requirement in this formula, because we are only considering valid model computations, and the effect is encoded in the model.

[5] `http://www.compass-research.eu`

[6] Recall that LTL uses 4 path operators: $\mathbf{G}\phi$ (globally ϕ) states that ϕ holds in every state of the computation. $\mathbf{F}\phi$ (finally ϕ) states that ϕ holds in some computation state. $\mathbf{X}\phi$ states that ϕ holds in the next state following the computation state under consideration. $\phi\ \mathbf{U}\ \psi$ states that finally ψ will hold in a computation state and ϕ fill hold in all previous states (if any).

[7] Control states are encoded as Boolean variables in the model state space, Idle = `true` means that state machine OUTPUT_CTRL is in control state Idle.

Observe that the application of LTL to characterise model computations associated with a requirement differs from its utilisation for black-box specification as in formula (1), where the behaviour required along those computations has to be specified in the formula, and only interface variables of the system may be referenced. It also differs from the application of temporal logics in property checking, where either all (a required property) or no computations (a requirements violation) of the model should fulfil the formula.

Referring to internal model elements frequently simplifies the formulas for characterising computations. Requirement REQ-002 (Flashing with 340ms/320ms on-off periods), for example, is witnessed by all computations satisfying (see Fig.9)

$$\mathbf{F}(\mathrm{OFF} \wedge \mathbf{X}\mathrm{ON}) \tag{2}$$

4.2 Requirements Tracing to the Model

The SysML modelling formalism [23] provides syntactic means to identify requirements in the model. In Fig.9, for example, the transitions ON $\rightarrow$ OFF and OFF $\rightarrow$ ON realise the flashing period specified by REQ-002. This is documented by means of the «satisfy» relation drawn from the transitions to the requirement. The interpretation of this relation is that every model computation finally covering one of the two transitions or both contributes to the requirement. Since computations cover OFF $\rightarrow$ ON if and only if they fulfil $\mathbf{F}(\mathrm{OFF} \wedge \mathbf{X}\mathrm{ON})$, the «satisfy» relation from ON $\rightarrow$ OFF to REQ-002 is redundant. Other examples for such simple relationships between model elements an requirements are shown in the state machine depicted in Fig 7. Formally speaking, these simple relationships are of the type

$$\mathbf{F}\langle \text{State Formula} \rangle \tag{3}$$

where the state formula expresses the condition that a model element related to the requirement is covered: for REQ-002, the formula (2) can be expressed in the form (3) as

$$\mathbf{F}(\mathrm{OFF} \wedge (\hat{t} - t_{\mathrm{OFF}}) \geq 320$$

Here t_{OFF} denotes the timer variable that stores the current time whenever control state OFF is entered and $\hat{t}$ is the current model execution time, so $(\hat{t} - t_{\mathrm{OFF}})$ expresses the fact that the relative time event after$(320ms)$ has occurred. In this case the transition OFF $\rightarrow$ ON must be taken, since UML/SysML state machine priority assigns higher priority to lower-level transitions: even if transitions FLASHING $\rightarrow$ FLASHING or FLASHING $\rightarrow$ Idle of the state machine in Fig. 8 are enabled, transition OFF $\rightarrow$ ON has higher priority because it resides in the sub-maschine of FLASHING.

Evaluations of system requirements in the automotive domain (in cooperation with Daimler) have shown that approximately 80% of requirements are reflected by model computations satisfying

$$\mathbf{F}\left(\bigvee_{i=0}^{h} \phi_i\right)$$

where the ϕ_i are state formulas, each one expressing coverage of a single model element.

About 20% of system requirements require more complex witnesses, whose LTL specification involve nested path operators and state formulas referring to model elements, variable valuations and time. For these situations, we use constraints containing the more complex LTL formulas,

and the constraints are linked to their associated requirements by means of the «satisfy» relation. Table 2 lists the requirements of the case study captured in Table 1, and associates the constraints characterising the witness traces for each requirement.

4.3 Test Cases

Since tests must terminate after a finite number of steps, they consist of traces $\iota = s_0 \dots s_k$ probing prefixes of relevant model computations $\pi = s_0 \dots s_k.s_{k+1} \dots$. If π is a witness for some requirement R characterised by LTL formula ϕ, a suitable test case ι has to be constructed in a way that at least does not violate ϕ while transiting through states $s_0 \dots s_k$, even though ϕ will be violated by many possible extensions of ι. This problem is well-understood from the field of bounded model checking (BMC), and Biere et. al. [3, 4] introduced a step semantics for evaluating LTL formulas on finite model traces. To this end, expression $\langle \varphi \rangle_i^{k-i}$ states that formula φ holds in state s_i of a trace of length $k+1$. For the operators of LTL, their semantics can then be specified inductively by[8]

- $\langle \mathbf{G}\ \varphi \rangle_0^k = \bigwedge_{i=0}^k \langle \varphi \rangle_i^{k-i}$ ($\mathbf{G}\phi$ is not violated on $\iota = s_0 \dots s_k$)
- $\langle \mathbf{X}\ \varphi \rangle_i^{k-i} = \langle \varphi \rangle_{i+1}^{k-i-1}$
- $\langle \varphi\ \mathbf{U}\ \psi \rangle_i^{k-i} = \langle \psi \rangle_i^{k-i} \vee (\langle \varphi \rangle_i^{k-i} \wedge \langle \varphi\ \mathbf{U}\ \psi \rangle_{i+1}^{k-i-1})$, $\quad \langle \mathbf{F}\psi \rangle_i^{k-i} = \langle \mathtt{true}\ \mathbf{U}\ \psi \rangle_i^{k-i}$

Using this bounded step semantics, each LTL formula can be transformed into formulas of the type

$$tc \equiv J(s_0) \wedge \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \wedge G(s_0, \dots, s_{n+1}) \tag{4}$$

which we call *symbolic test cases*[9] and which can be handled by the SMT solver. Conjunct $J(s_0)$ characterises the current model state s_0 from where the next test objective represented by some LTL formula ϕ should be covered. This formula has to be translated into a predicate $G(s_0, \dots, s_{n+1})$, using the semantic rules listed above. Predicate Φ is the transition relation of the model, and conjunct $\bigwedge_{i=0}^n \Phi(s_i, s_{i+1})$ ensures that the solution of $G(s_0, \dots, s_{n+1})$ results in a valid trace of the model, starting from s_0.

Example 1. Consider LTL formula

$$\phi \equiv (x = 0)\mathbf{U}(y > 0 \wedge \mathbf{X}(\mathbf{G}z = 1))$$

and suppose we are looking for a witness trace $\iota = s_0 \dots s_n \dots$ with a length of at least $n+1$ or longer. Then the SMT solver is activated with the following BMC instances to solve.
In step 0, try solving

$$bmc_0 \equiv \left(\bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \right) \wedge s_0(y) > 0 \wedge \left(\bigwedge_{i=1}^{n+1} s_i(z) = 1 \right)$$

[8] The semantics presented in [4] has been simplified for our purposes. In [4], the authors consider possible cycles in the transition graph which are reachable within a bounded number of steps from s_0. This is used to prove the existence of witnesses for formulas whose validity can only be proven on infinite paths. For testing purposes, we are only dealing with finite traces anyway; this leads to the slightly simplified bounded step semantics presented here.

[9] In the context of BMC, these formulas are called *bounded model checking instances*.

If this succeeds we are done: the solution of bmc_0 is a legal trace ι of the model, since $\Phi(s_i, s_{i+1})$ holds for each pair of consecutive states in ι. Formula ϕ holds on ι because $y > 0$ is true in s_0 and $z = 1$ holds for states $s_1 \ldots s_{n+1}$, so the right-hand side operand of **U** is fulfilled in the initial state of this trace.
Otherwise we try to get a witness for the following formula in step 1.

$$bmc_1 \equiv \left(\bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \right) \wedge s_0(x) = 0 \wedge s_1(y) > 0 \wedge \left(\bigwedge_{i=2}^{n+1} s_i(z) = 1 \right)$$

If no solution exists we continue with step 2.

$$bmc_2 \equiv \left(\bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \right) \wedge s_0(x) = 0 \wedge s_1(x) = 0 \wedge s_2(y) > 0 \wedge \left(\bigwedge_{i=3}^{n+1} s_i(z) = 1 \right)$$

and so on, until a solution is found or no solution of length $n+1$ is feasible. □

While LTL formulas are well-suited to specify computations fulfilling a wide variety of constraints, it has to be noted that it is also capable of defining properties of computations that will never be tested in practice, because they can only be verified on infinite computations and not on finite trace prefixes thereof (e.g., fairness properties). It is therefore desirable to identify a subset of LTL formulas that are tailored to the testers' needs for specifying finite traces with certain properties. This subset is called *SafetyLTL* and has been introduced in [36]. It is suitable for defining safety properties of computations, that is, properties that can always be falsified on a finite computation prefix. The SafetyLTL subset of LTL can be syntactically characterised as follows.

- Negation is only allowed before atomic propositions (so-called *negation normal form*).
- Disjunction $\vee$ and conjunction $\wedge$ are always allowed.
- Next operators **X**, globally operators **G** and weakly-until operators **W** are allowed[10].
- Semantically equivalent formulas also belong to SafetyLTL.

Concrete test data is created by solving constraints of the type displayed in Equation (4) using the integrated SMT solver SONOLAR [27]. Finally the test procedure generator takes the solutions calculated by the SMT solver and turns them into stimulation sequences, that is, timed input traces to the SUT. Moreover, the test procedure generator creates test oracles from the model components describing the SUT behaviour.

In requirements-driven testing, $G(s_0, \ldots, s_{n+1})$ specifies traces that are *witnesses* of a certain requirement R. Indeed, Formula (4) specifies an *equivalence class* of traces that are suitable for testing R. In model-driven testing, $G(s_0, \ldots, s_{n+1})$ specifies traces that are suitable for covering certain portions (control states, transitions, interfaces, ...) of the model. In the paragraphs below it will be explained how requirements-driven and model-driven testing are related to each other.

[10] Recall that the weakly-until operator is defined as $\phi \ \mathbf{W} \ \psi \equiv_{\text{def}} (\phi \ \mathbf{U} \ \psi) \vee \mathbf{G}\phi$, and that the until operator can be expressed by $\phi \ \mathbf{U} \ \psi \equiv (\phi \ \mathbf{W} \ \psi) \wedge \mathbf{F}\psi$.

4.4 Model Coverage Test Cases

Since adequate test models express all SUT requirements to be tested, it is reasonable to specify and perform test cases achieving model coverage. As we have seen above, a behavioural model element (state machine control state, transition, operation, ...) is covered by a trace $\iota = s_0 \dots s_k$, if the element's behaviour is exercised during some transition $s_i \to s_{i+1}$. For a control state c this means that $s_{i+1}(c) = \texttt{true}$, and, consequently, the state's entry action (if any) is executed. For a transition this means that its firing condition becomes true in some s_i. Operations f are covered when they are associated with actions of covered states or transitions executing f.

There exists a wide variety of model coverage strategies, many of them are discussed in [42]. The standards for safety-critical systems development and V&V have only recently started to consider the model-driven development and V&V paradigm. It seems that the avionic standard RTCA DO-178C [32] is currently the most advanced with respect to model-based systems engineering. It requires to achieve operation coverage, transition coverage, decision coverage, and equivalence class and boundary value coverage, when verifying design models [31, Table MB.6-1]. Neither the standard, nor [42], however, elaborate on coverage of timing conditions (e.g., clock zones in Time Automata) or the coverage of execution state vectors of concurrent model components.

In RT-Tester, the following model coverage criteria are currently implemented: (1) basic control state coverage, (2) transition coverage, MC/DC coverage, (3) hierarchic transition coverage[11] with or without MC/DC coverage, (4) equivalence class and boundary value coverage, (5) basic control state pairs coverage, (6) interface coverage and (7) block coverage.

Basic control state pairs coverage exercises all feasible control state combinations of concurrent state machines in writer-reader relationship. The equivalence class coverage technique in combination with basic control state pairs coverage also produces a (not necessarily complete) coverage of clock zones.

Each of these coverage criteria can be specified by means of LTL formulas or, equivalently, BMC instances.

Example 2. For state machine FLASH_CTRL (Fig. 6), the hierarchic transition coverage is achieved by test cases

$$
\begin{aligned}
tc_1 &\equiv \mathbf{F}(\mathrm{EMER\_OFF} \wedge \mathrm{EmerFlash}) \\
tc_2 &\equiv \mathbf{F}(\mathrm{EMER\_ACTIVE} \wedge \mathrm{TurnIndLvr} \neq 0 \wedge \\
&\quad ((\mathrm{TurnIndLvr} = 1) \neq \mathrm{Left1} \vee (\mathrm{TurnIndLvr} = 2) \neq \mathrm{Right1})) \\
tc_3 &\equiv \mathbf{F}(\mathrm{EMER\_ACTIVE} \wedge (\mathrm{Left1} \vee \mathrm{Right1}) \wedge \mathrm{TurnIndLvr} = 0) \\
tc_4 &\equiv \mathbf{F}(\mathrm{TURN\_IND\_OVERRIDE} \wedge \mathrm{TurnIndLvr} = 0) \\
tc_5 &\equiv \mathbf{F}(\neg\mathrm{EmerFlash} \wedge \mathrm{EMER\_ACTIVE} \wedge \\
&\quad ((\mathrm{TurnIndLvr} \neq 0 \wedge \mathrm{TurnIndLvr} = \mathrm{Left1} \vee \mathrm{TurnIndLvr} = \mathrm{Right1}) \vee \\
&\quad (\mathrm{TurnIndLvr} = 0 \wedge \neg(\mathrm{Left1} \vee \mathrm{Right1}))) \\
tc_6 &\equiv \mathbf{F}(\neg\mathrm{EmerFlash} \wedge \mathrm{TURN\_IND\_OVERRIDE} \wedge \mathrm{TurnIndLvr} \neq 0)
\end{aligned}
$$

□

[11] This applies to higher-level transitions of hierarchic state machines: they are exercised several times with as many subordinate control states as possible.

4.5 Automated Compilation of Traceability Data

Having identified the test cases suitable for model coverage, these can be related to requirements in an automated way.

- If requirement R is linked to model elements by «satisfy» relationships, then the test cases covering these elements are automatically related to R.
- If requirement R is characterised by a LTL formula ϕ not directly related to model elements, we proceed as follows.
  - Transform ϕ into disjunctive normal form $\phi \equiv \bigvee_{i=0}^{m} \phi_i$ and associate test cases for each ϕ_i separately.
  - Each test case $tc \equiv \psi$ derived from the model is related to R, if $\psi \Rightarrow \phi_i$ holds.
  - If test case $tc \equiv \psi$ is neither stronger nor weaker than the requirement in the sense that $\psi \wedge \phi_i$ has a solution, add a new test case $tc' \equiv \psi \wedge \phi_i$ and relate tc' to R.
  - If at least one of two test cases $tc_1 \equiv \mathbf{F}\psi_1$ and $tc_2 \equiv \mathbf{F}\psi_2$ implies the requirement and $tc' \equiv \mathbf{F}(\psi_1 \wedge \psi_2)$ has a solution, add tc' to the test case database and trace it to R.

Example 3. Consider requirement REQ-002 (Flashing with 340ms/320ms on-off periods) of the example from Table 1. It is characterised by covering transitions ON $\rightarrow$ OFF and OFF $\rightarrow$ ON (see Table 2). By tracing these transitions back to model coverage test cases, the following cases can be identified, and these trace back to REQ-002.

$$\begin{aligned}
tc_7 &\equiv \mathbf{F}(\mathrm{OFF} \wedge (\hat{t} - t_{\mathrm{OFF}}) \geq 320) \\
tc_8 &\equiv \mathbf{F}(\mathrm{OFF} \wedge (\hat{t} - t_{\mathrm{OFF}}) \geq 320 \wedge \mathrm{TurnIndLvr} = 1) \\
tc_9 &\equiv \mathbf{F}(\mathrm{OFF} \wedge (\hat{t} - t_{\mathrm{OFF}}) \geq 320 \wedge \mathrm{TurnIndLvr} = 2) \\
tc_{10} &\equiv \mathbf{F}(\mathrm{OFF} \wedge (\hat{t} - t_{\mathrm{OFF}}) \geq 320 \wedge \mathrm{EMER_ACTIVE}) \\
tc_{11} &\equiv \mathbf{F}(\mathrm{OFF} \wedge (\hat{t} - t_{\mathrm{OFF}}) \geq 320 \wedge \mathrm{TURN_IND_OVERRIDE})
\end{aligned}$$

□

The test cases listed here are only a subset of the complete list that traces back to REQ-002. Test cases tc_8,tc_9 result from combining interface coverage on SUT input TurnIndLvr with coverage of the OFF $\rightarrow$ ON. Cases tc_{10},tc_{11} result from combining basic control state pairs coverage with the transition coverage. Test case tc_7 is redundant if any of the others is performed. It is quite obvious that the test case generation technique defined above runs into combinatorial explosion problems. Even for the small sample system discussed here, the list of test cases from Example 3 could be extended by

$$\begin{aligned}
tc_{12} &\equiv \mathbf{F}(\mathrm{OFF} \wedge (\hat{t} - t_{\mathrm{OFF}}) \geq 320 \wedge \mathrm{EMER_ACTIVE} \wedge \mathrm{TurnIndLvr} = 0) \\
tc_{13} &\equiv \mathbf{F}(\mathrm{OFF} \wedge (\hat{t} - t_{\mathrm{OFF}}) \geq 320 \wedge \mathrm{EMER_ACTIVE} \wedge \mathrm{TurnIndLvr} = 1) \\
tc_{14} &\equiv \mathbf{F}(\mathrm{OFF} \wedge (\hat{t} - t_{\mathrm{OFF}}) \geq 320 \wedge \mathrm{EMER_ACTIVE} \wedge \mathrm{TurnIndLvr} = 2) \\
&\ldots
\end{aligned}$$

4.6 Test Case Selection According to Criticality

It is quite obvious that the number of test cases related to a requirement can become quite vast, and that some of the test cases investigate more specific situations than others. This problem

is closely related to the problem of exhaustive testing which will be discussed below. Since an exhaustive execution of all test case combinations related to a requirement will be impossible for fair-sized systems, a justified reduction of the potentially applicable test cases to a smaller collection is required. In the case of safety-critical systems development, such a justification should conform to the standards applicable for V&V of these systems.

In the case of avionic systems, the RTCA DO-178C standard [32] requires structural tests with respect to data and control coupling and full requirements coverage through testing, but does not specify when a requirement has been verified with a sufficient number of test cases. Instead, the standard gives test end criteria by setting code coverage goals, the coverage to be achieved depending on the SUT's criticality [31, MB.C-7]: for assurance level 1 systems (highest criticality), MC/DC coverage has to be achieved, for level 2 decision coverage, and for level 3 statement coverage. For levels 4 and 5, only high-level requirements have to be covered without setting any code coverage goals, and for assurance level 5 the requirement to test data and control coupling is dropped.

As a consequence, the model-based test case coverage can be tuned according to the code coverage achieved, whenever the source code is available and the assurance level is in 1 — 3: start with basic control state coverage cases related to the requirement, increase coverage by adding hierarchic and MC/DC coverage test cases until the required code coverage is achieved. Add interface and basic control state pairs coverage cases until the data and control coupling coverage has been achieved as well. For levels 4 or 5, no discussion is necessary, since here any "reasonable" test case assignment to each high-level requirement is acceptable, due to the low criticality of the SUT.

When MBT is applied on system level, however, it will generally be infeasible to measure code coverage achieved during system tests. For systems of systems, in particular, system-level tests will never cover any significant amount of code coverage, and the coverage values achieved will not be obtainable in most cases, both for technical and for security reasons. Here we suggest to proceed as follows.

- For assurance level 3, exercise
  - interface tests – this ensures verification of data and control coupling,
  - basic control state coverage test cases,
  - refine these test cases $tc \equiv \psi$ only if requirements have stricter characterisations ϕ_i; in this case add $tc' \equiv \psi \wedge \phi_i$.
- For assurance level 2, follow the same pattern, but use transition coverage test cases.
- For assurance level 1, exercise
  - interface tests,
  - basic control state pairs coverage test cases to refine the data and control coupling tests (recall that these test cases stem from writer-reader analyses),
  - MC/DC coverage test cases in combination with hierarchic transition coverage,
  - first-level refinements of test cases related to requirements as illustrated in Example 3,
  - second level refinements (as in test cases tc_{12},tc_{13},tc_{14} above), if the additional conjuncts have direct impact on the requirement.

Following these rules, and supposing that our sample system were of assurance level 1, the test cases displayed in Example 3 would be necessary. Test cases tc_{12},tc_{13},tc_{14}, however, would not

be required, since the TurnIndLvr has no impact on REQ-002 according to the model: the risk of a hidden impact of this interface on the requirement has already been taken into account when testing tc_8,tc_9.

4.6.1 Test Strategies Proving Conformance

An alternative for justifying test strategies consists in proving that they will finally converge to an exhaustive test suite establishing some conformance relation between model and SUT. This approach has a long tradition: one of the first contributions in this field was Chow's W-Method [9] applicable for minimal state machines, which was generalised and extended into many directions, so that even in the core of the exhaustive test strategy for timed automata [37] some argument from the W-Method is used.

Though execution of exhaustive test suites will generally be infeasible in practice, convergence to exhaustive test suites ensures that new test cases added to the suite will really increase the assurance level by a positive amount: intuitively designed test strategies often do not possess this property, because additional test cases may just re-test SUT aspects already covered by existing ones.

The known exhaustive strategies typically operate on finite data types (discrete events, or variables with data ranges that can easily be enumerated). It is an interesting research challenge whether similar results can be obtained in presence of large data types, if application of equivalence class partitioning is justified. In [13] the authors formalise the concept of equivalence class partitioning and prove that exhaustive suites can be constructed for white-box test situations. In [18] this approach is currently generalised within the COMPASS project with respect to black-box testing and semantic models that are more general than the one underlying the results presented in [13].

4.7 Challenges to Test Case Generation and Test Strategy Design

The size of SoS state spaces implies that exhaustive investigation of the complete concrete state space will certainly be infeasible. We suggest to tackle this problem by two orthogonal strategies, as is currently investigated in the COMPASS project [17].

- On constituent system level, different behaviours associated with the same local mission threads[12] will be comprised in equivalence classes. This reduces the complexity problem for SoS system testing to covering combinations of classes of constituent system behaviours instead of sequences of concrete state vector combinations.
- On SoS system level, "relevant" class combinations are identified by means of different variants of impact analysis, such as data flow analyses or investigation of contractual dependencies. Behaviours of constituent systems which do not affect the relevant class combinations under consideration will be selected according to the principle of orthogonal arrays [28], because this promises an effective combinatorial distribution of unrelated behaviours exercised concurrently with the critical ones.

Apart from size and complexity, SoS present another challenge, because they typically change their configuration dynamically during run-time. The dynamic adaptation of test objectives is

[12] Mission threads are end-to-end tests; in the context described here, mission threads are executed on constituent system level.

particularly relevant for run-time acceptance testing of changing SoS configurations. In contrast to development models for SoS, however, we only have to consider bounded changes of SoS configurations, because every test suite can only consider a bounded number of configurations anyway. It remains to investigate how to determine configurations possessing sufficient error detection strength. Results from the field of mutation testing will help to determine this strength in a systematic and measurable way.

A further problem for systems of SoS complexity is presented by the fact that not every behaviour can be full captured in the model, which results in under-specification and non-determinism. Test strategy elaboration in presence of this problem be achieved in the following way.

- The SoS system behaviour is structured into several top-level operational modes. It is expected that switching between these modes can be performed in a deterministic way for normal behaviour tests: it is unlikely that SoS performing operational mode changes only on a random basis are acceptable and "testworthy".
- Entry into failure modes is non-deterministic, but can be initiated in a deterministic way for test purposes by means of pre-planned failure injections.
- The behaviour in each operational mode is not completely deterministic, but can be captured by sets of constraints governing the acceptable computations in each mode. Test oracles will therefore no longer check for explicit output traces of the SUT but for compliance of the traces observed with the constraints applicable in each mode.
- For test stimulation purposes the SMT solver computes sequences of feasible mode switches and the test data provoking these switches.
- Incremental test model elaboration can be performed by adding constraints identified during test observations to the modes where they are applicable. To this end, techniques from machine learning seem to be promising.

Justification of test strategies will be performed by proving that they will "converge" to exhaustive tests proving some compliance relation between SUT and reference model.

5 Conclusion

In this article several aspects of industrial-strength model-based testing and its underlying methods have been presented. A reference tool has been described, so that the presentation may serve as a benchmark for alternative tools capable of handling test campaigns of equal or even higher complexity. Readers are invited to join the discussion on suitable benchmarks for MBT tools – initial suggestions on benchmarking can be found in [26] – and to contribute case studies and models to the MBT benchmark website `http://www.mbt-benchmarks.org`.

A further topic beyond the scope of this paper is of considerable importance for tool builders: MBT tools automating test campaigns for safety-relevant systems have to be *qualified*, and standards like RTCA DO-178C [32] for the avionic domain, CENELEC EN650128 [38] for the railway domain, and ISO 26262 [19] for the automotive domain have rather precise policies about how tool qualification can be obtained. A detailed comparison between tool qualification requirements of these standards is presented in [16], and it is described in [5] how tool qualification has been obtained for RT-Tester. We believe that the complexity of the algorithms required in MBT

tools justifies that effort is spent on their qualification, so that their automated application will not mask errors of the SUT due to undetected failures in the tool.

Acknowledgements. The author would like to thank the organisers of the MBT 2013 for giving him the opportunity to present the ideas summarised in this paper. Special thanks go to Jörg Brauer, Elena Gorbachuk, Wen-ling Huang, Florian Lapschies and Uwe Schulze for contributing to the results presented here.

References

[1] AERONAUTICAL RADIO, INC. (2009): *Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network*. AERONAUTICAL RADIO, INC., 2551 Riva Road, Annapolis, Maryland 21401-7435.

[2] Paul Baker, Oystein Haugen, Zhen Ru Dai, Clay Williams & Jens Grabowski (2008): *Model-Driven Testing – Using the UML Testing Profile*. Springer, Berlin Heidelberg.

[3] Armin Biere, Alessandro Cimatti, Edmund M. Clarke & Yunshan Zhu (1999): *Symbolic Model Checking without BDDs*. In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, Springer-Verlag, London, UK, UK, pp. 193–207, doi:10.1007/3-540-49059-0_14.

[4] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala & Viktor Schuppan (2006): *Linear Encodings of Bounded LTL Model Checking*. *Logical Methods in Computer Science* 2(5), pp. 1–64, doi:10.2168/LMCS-2(5:5)2006.

[5] Jörg Brauer, Jan Peleska & Uwe Schulze (2012): *Efficient and Trustworthy Tool Qualification for Model-Based Testing Tools*. In Brian Nielsen & Carsten Weise, editors: *Testing Software and Systems. Proceedings of the 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 2012*, *Lecture Notes in Computer Science* 7641, Springer, Heidelberg Dordrecht London New York, pp. 8–23, doi:10.1007/978-3-642-34691-0_3.

[6] E. Brinksma (1988): *A Theory for the Derivation of Tests*. In S. Aggarwal & K. Sabnani, editors: *Protocol Specification Testing and Verification VIII (PSTV '88)*, pp. 63–74.

[7] R. E. Bryant, P. Chauhan, E. M. Clarke & A. Goel (2000): *A Theory of Consistency for Modular Synchronous Systems*. In W. A. Hunt & S. D. Johnson, editors: *Formal Methods in Computer-Aided Design (FMCAD)*, *Lecture Notes in Computer Science* 1954, Springer, pp. 486–504, doi:10.1007/3-540-40922-X_30.

[8] A. L. C. Calvalcanti & M.-C. Gaudel (2011): *Testing for Refinement in Circus*. *Acta Informatica* 48(2), pp. 97–147, doi:10.1007/s00236-011-0133-z.

[9] Tsun S. Chow (1978): *Testing Software Design Modeled by Finite-State Machines*. *IEEE Transactions on Software Engineering* SE-4(3), pp. 178–186, doi:10.1109/TSE.1978.231496.

[10] Edmund M. Clarke, Orna Grumberg & Doron A. Peled (1999): *Model Checking*. The MIT Press, Cambridge, Massachusetts.

[11] R. De Nicola & M. Hennessy (1984): *Testing Equivalences for Processes*. *Theoretical Computer Science* 34, pp. 83–133, doi:10.1016/0304-3975(84)90113-0.

[12] Esterel Technologies: *SCADE Suite Product Description*. `http://www.estereltechnologies.com`.

[13] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte & Margus Veanes (2002): *Generating Finite State Machines from Abstract State Machines*. *ACM SIGSOFT Software Engineering Notes* 27(4), pp. 112–122, doi:10.1145/566171.566190.

[14] M. Hennessy (1988): *Algebraic Theory of Processes*. MIT Press, Cambridge, Massachusetts, London.

[15] C. A. R. Hoare & H. Jifeng (1998): *Unifying Theories of Programming*. Prentice-Hall.

[16] Wen ling Huang, Jan Peleska & Uwe Schulze (2013): *Test Automation Support*. Technical Report D34.1, COMPASS Comprehensive Modelling for Advanced Systems of Systems.

[17] Wen ling Huang, Jan Peleska & Uwe Schulze (to appear 2014): *Specialised Test Strategies*. Technical Report D34.2, COMPASS Comprehensive Modelling for Advanced Systems of Systems.

[18] Wen-ling Huang & Jan Peleska (2012): *Specialised Test Strategies*. Public Document, COMPASS Comprehensive Modelling for Advanced Systems of Systems.

[19] (2009): *Road Vehicles - Functional Safety - Part 8: Supporting Processes*. Technical Report, International Organization for Standardization. ICS 43.040.10.

[20] ISO/DIS 26262-4 (2009): *Road vehicles – functional safety – Part 4: Product development: system level*. Technical Report, International Organization for Standardization.

[21] Helge Loding & Jan Peleska (2010): *Timed Moore Automata: Test Data Generation and Model Checking. Software Testing, Verification, and Validation, 2008 International Conference on* 0, pp. 449–458, doi:10.1109/ICST.2010.60.

[22] Rocco De Nicola & Frits Vaandrager (1990): *Action versus State based Logics for Transition Systems*. In Irène Guessarian, editor: *Semantics of Systems of Concurrent Processe*, *LNCS* 469, Springer-Verlag, Berlin, Heidelberg, pp. 407–419, doi:10.1007/3-540-53479-2_17.

[23] Object Management Group (2010): *OMG Systems Modeling Language (OMG SysML^{TM})*. Technical Report, Object Management Group. OMG Document Number: formal/2010-06-02.

[24] OMG (2011): *OMG Unified Modeling Language (OMG UML) Superstructure ver. 2.4.1. www.uml.org/spec/UML/2.4.1/Superstructure/PDF/*.

[25] J. Peleska & M. Siegel (1997): *Test Automation of Safety-Critical Reactive Systems. South African Computer Jounal* 19, pp. 53–77.

[26] Jan Peleska, Artur Honisch, Florian Lapschies, Helge Löding, Hermann Schmid, Peer Smuda, Elena Vorobev & Cornelia Zahlten (2011): *A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain*. In Burkhart Wolff & Fatiha Zaidi, editors: *Testing Software and Systems. Proceedings of the 23rd IFIP WG 6.1 International Conference, ICTSS 2011*, *LNCS* 7019, IFIP WG 6.1, Springer, Heidelberg Dordrecht London New York, pp. 146–161, doi:10.1007/978-3-642-24580-0_1.

[27] Jan Peleska, Elena Vorobev & Florian Lapschies (2011): *Automated Test Case Generation with SMT-Solving and Abstract Interpretation*. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *Nasa Formal Methods, Third International Symposium, NFM 2011*, *LNCS* 6617, Springer, Pasadena, CA, USA, pp. 298–312, doi:10.1007/978-3-642-20398-5_22.

[28] M. S. Phadke (1989): *Quality Engineering Using Robust Design*. Prentice Hall, Englewood Cliff, NJ.

[29] F. Rogin, T. Klotz, G. Fey, R. Drechsler & S. Rulke (2009): *Advanced Verification by Automatic Property Generation. IET Computers & Digital Techniques* 3(4), pp. 338–353, doi:10.1049/iet-cdt.2008.0110. Available at `http://link.aip.org/link/?CDT/3/338/1`.

[30] A. W. Roscoe (2010): *Understanding Concurrent Systems*. Springer.

[31] RTCA SC-205/EUROCAE WG-71 (2011): *Model-Based Development and Verification Supplement to DO-178C and DO-278A*. RTCA/DO-331, RTCA, Inc., 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036.

[32] RTCA SC-205/EUROCAE WG-71 (2011): *Software Considerations in Airborne Systems and Equipment Certification*. RTCA/DO-178C, RTCA, Inc., 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036.

[33] RTCA,SC-167 (1992): *Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B*. RTCA.

[34] S. Schneider (1995): *An Operational Semantics for Timed CSP*. Information and Computation 116, pp. 193–213, doi:10.1006/inco.1995.1014.

[35] S. Schneider (2000): *Concurrent and Real-time Systems – The CSP Approach*. Wiley and Sons Ltd.

[36] A. P. Sistla (1994): *Liveness and Fairness in Temporal Logic*. Formal Aspects of Computing 6(5), pp. 495–512, doi:10.1007/BF01211865.

[37] J.G. Springintveld, F.W. Vaandrager & P.R. D'Argenio (2001): *Testing timed automata*. Theoretical Computer Science 254(1-2), pp. 225–257, doi:10.1016/S0304-3975(99)00134-6.

[38] European Committee for Electrotechnical Standardization (2001): *EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, Brussels.

[39] Jan Tretmans (1996): *Test generation with inputs, outputs and repetitive quiescence*. Software – Concepts and Tools 17(3), pp. 103–120.

[40] Jan Tretmans (1999): *Testing Concurrent Systems: A Formal Approach*. In J.C.M. Naeten & S. Mauw, editors: *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, Lecture Notes in Computer Science 1664, Springer, pp. 46–65, doi:10.1007/3-540-48320-9_6.

[41] Frits Vaandrager (2012): *Active Learning of Extended Finite State Machines*. In Brian Nielsen & Carsten Weise, editors: *Testing Software and Systems. Proceedings of the 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 2012*, Lecture Notes in Computer Science 7641, Springer, Heidelberg Dordrecht London New York, pp. 5–7, doi:10.1007/978-3-642-34691-0_2.

[42] Stephan Weißleder (2010): *Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines*. Doctoral thesis, Humboldt-University Berlin, Germany.

[43] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa & S. Perry (2012): *Features of CML: a Formal Modelling Language for Systems of Systems*. IEEE Systems Journal 6, doi:10.1109/SYSoSE.2012.6384144.

A Case Study: Turn Indication Control Function

As a case study we consider the turn indication function of a vehicle providing left/right indication and emergency flashing by means of exterior lights flashing with a given frequency. Left/right indication is switched on by means of the turn indicator lever with its positions 0 (neutral), 1 (left), and 2(right). Emergency flashing is controlled by means of a switch with positions 0 (off) and 1 (on). Activating the indication lights is subject to the condition that the available voltage is sufficiently high. The requirements for the turn indication function are as shown in Table 1.

The SysML test model for this system structured into TE and SUT blocks, as shown in Fig. 4. The interfaces shown in this diagram are the observable SUT outputs and writable inputs that may be accessed by the TE. RT-Tester allows for SysML properties and signal events to be exchanged between SUT and TE model components. The tool provides interface modules mapping their valuations onto concrete software or hardware interfaces and vice versa. In a software integration test the turn indication lever values and the status of the emergency switch may be passed to the SUT, for example, by means of shared variables. The SUT outputs (left-hand side lamps on/off, right-hand side lamps on/off) can also be represented by Boolean output variables of the SUT. In a HW/SW integration test interface modules would map the turn indication lever status and the emergency flash button to discrete inputs to the SUT. In a system integration test the actual voltage and the current placed by the SUT on the indication

Table 1: Requirements of the turn indication control system

Requirement	**Description**
REQ-001 Flashing requires sufficient voltage	Indication lights are only active, if the electrical voltage (input Voltage) is $> 80\%$ of the nominal voltage.
REQ-002 Flashing with 340ms/320ms on-off periods	If any lights are flashing, this is done synchronously with a 340ms ON – 320ms OFF period.
REQ-003 Switch on turn indication left	An input change from turn indication lever state TurnIndLvr = 0 or 2 to TurnIndLvr = 1 switches indication lights left (output FlashLeft) into flashing mode and switches indication lights right (output FlashRight) off.
REQ-004 Switch on turn indication right	An input change from turn indication lever state TurnIndLvr = 0 or 1 to TurnIndLvr = 2 switches indication lights right (output FlashRight) into flashing mode and switches indication lights left (output FlashLeft) off.
REQ-005 Emergency flashing on overrides left/right flashing	An input change from EmerFlash = 0 to EmerFlash = 1 switches indication lights left (output FlashLeft) and right (output FlashRight) into flashing mode, regardless of any previously activated turn indication.
REQ-006 Left-/right flashing overrides emergency flashing	Activation of the turn indication left or right overrides emergency flashing, if the latter has been activated before.
REQ-007 Resume emergency flashing	If turn indication left or right is switched off and emergency flashing is still active, emergency flashing is continued or resumed, respectively.
REQ-008 Resume turn indication flashing	If emergency flashing is turned off and turn indication left or right is still active, the turn indication is continued or resumed, respectively.
REQ-009 Tip flashing	If turn indication left or right is switched off before three flashing periods have elapsed, the turn indication will continue until three on-off periods have been performed.

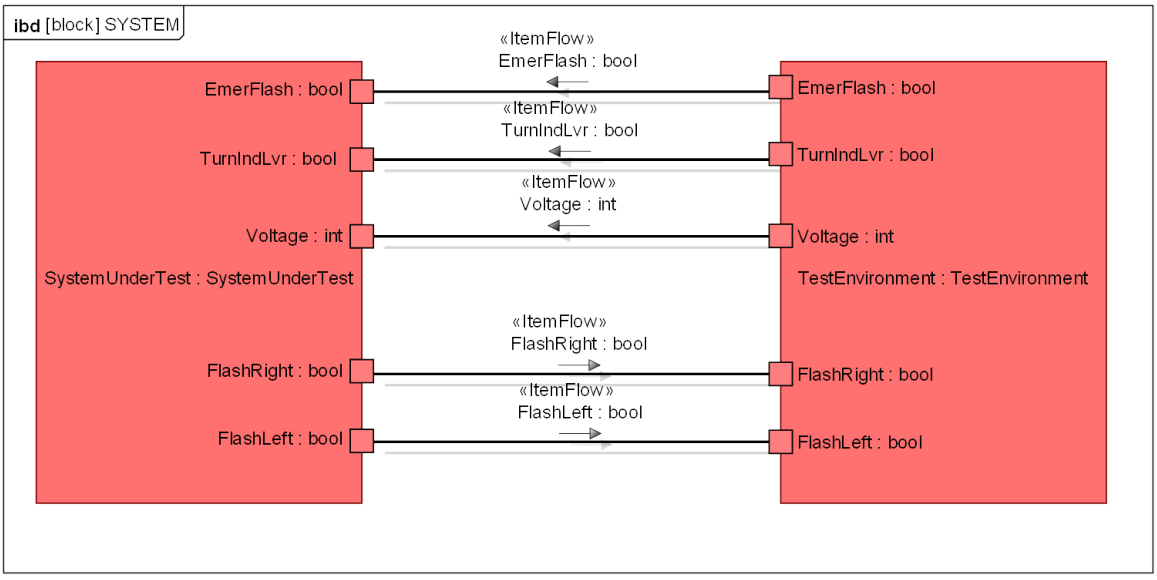


Figure 4: Interface between TE and SUT.

lamps would be measured. The interface abstraction required for the test level is specified by a signal map that associates abstract SysML model interfaces with concrete interfaces of the test equipment.

The structural view on the SUT has to be decomposed further, until each block is associated with a sequential behaviour. For the case study discussed here, the SUT is further decomposed into two concurrent functions as depicted in Fig. 5. Functional component FLASH_CTRL performs the decisions about left/right indication or emergency flashing. The decision is communicated to component OUTPUT_CTRL by means of internal interface Left (flashing on left-hand side indication lights if Left = 1) and Right (flashing on right-hand side indication lights if Right = 1). Block OUTPUT_CTRL controls the flashing cycles and switches off indication lamps if the voltage gets too low. The FLASH_CONTROL component operates as follows.

- As long as the emergency flash switch has not been activated, Left/Right are set according

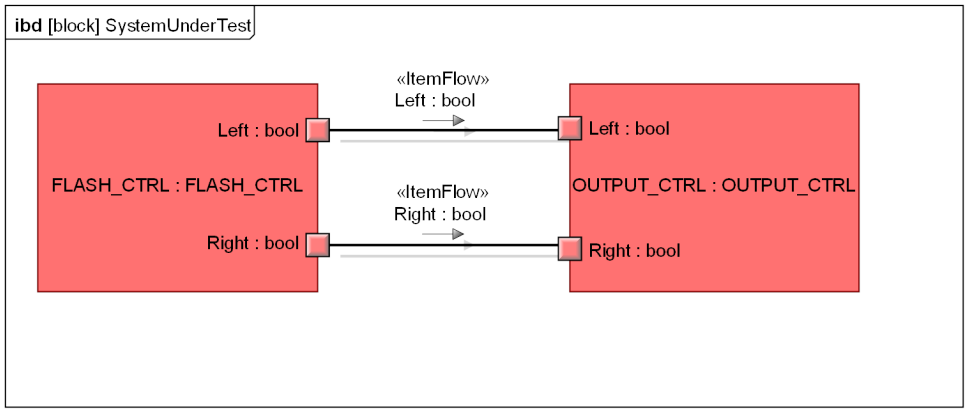


Figure 5: Functional decomposition of the SUT.

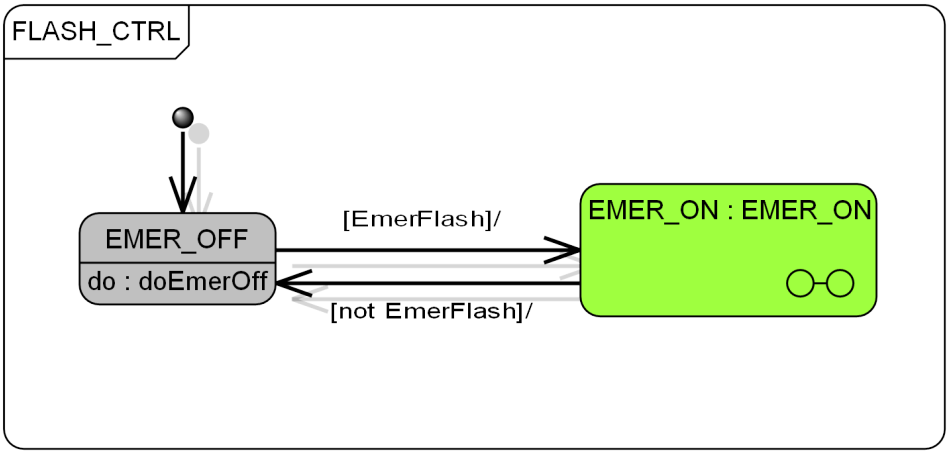


Figure 6: State machine controlling left/right and emergency flashing.

to the turn indication lever status. This is specified in do activity doEmerOff.

- As soon as the emergency flash switch EmerFlash is switched on, Left/Right are set as specified in sub-state machine EMER_ON (Fig 7).
- When entering EMER_ON, Left/Right are both set to true and the state machine remains in control state EMER_ACTIVE.
- When the turn indication lever is changed to left or right position, emergency flashing is overridden, and left/right indication is performed.
- Emergency flashing is resumed if the turn indication lever is switched into neutral position.

Function OUTPUT_CTRL sets the SUT output interfaces FlashLeft and FlashRight (Fig. 8 and 9). The indication lamps are switched according to the internal interface state Left/Right, if the voltage is greater than 80% of the nominal voltage. After the lamps have been on for 340ms, they are switched off and stay so until 320ms have passed. A counter FlashCtr is maintained: if the turn indication lever is switched from left or right back to the neutral position before 3 flashing periods have been performed, left/right indication will remain active until the end of these 3 periods.

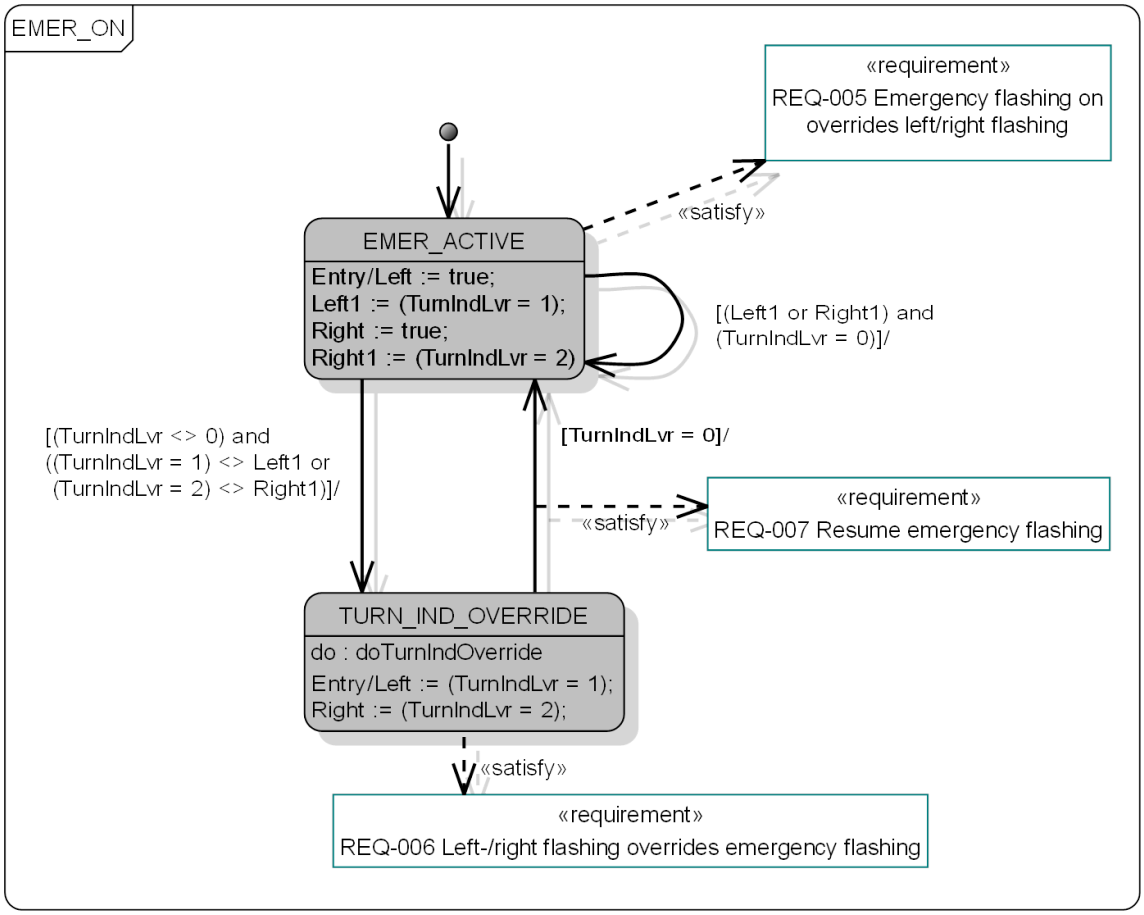


Figure 7: Decomposition of control state EMER_ON.

Table 2: Requirements and associated constraints identifying witness computations.

Requirement	**Constraint**
REQ-001 Flashing requires sufficient voltage	«Constraint» $\mathbf{F}(\mathit{Voltage} \leq 80)$
REQ-002 Flashing with 340ms/320ms on-off periods	«Transition» ON $\rightarrow$ OFF «Transition» OFF $\rightarrow$ ON
REQ-003 Switch on turn indication left	«Constraint» $\mathbf{F}(\mathrm{FlashLeft} = 1 \wedge \mathrm{FlashRight} = 0)$
REQ-004 Switch on turn indication right	«Constraint» $\mathbf{F}(\mathrm{FlashLeft} = 0 \wedge \mathrm{FlashRight} = 1)$
REQ-005 Emergency flashing on overrides left/right flashing	«Constraint» $\mathbf{F}(\mathrm{EMER\_OFF} \wedge \mathrm{TurnIndLvr} > 0 \wedge \mathrm{EmerFlash})$
REQ-006 Left-/right flashing overrides emergency flashing	«Atomic State» TURN_IND_OVERRIDE
REQ-007 Resume emergency flashing	«Transition»TURN_IND_OVERRIDE $\rightarrow$ EMER_ACTIVE
REQ-008 Resume turn indication flashing	«Constraint» $\mathbf{F}(\mathrm{EMER\_ACTIVE} \wedge \neg\mathrm{EmerFlash} \wedge \mathrm{TurnIndLvr} > 0)$
REQ-009 Tip flashing	«Constraint» $\mathbf{F}(\mathrm{Voltage} > 80 \wedge \neg(\mathrm{Left} \vee \mathrm{Right}) \wedge \mathrm{Left1} + \mathrm{Right1} = 1 \wedge \mathrm{FlashCtr} < 3)$

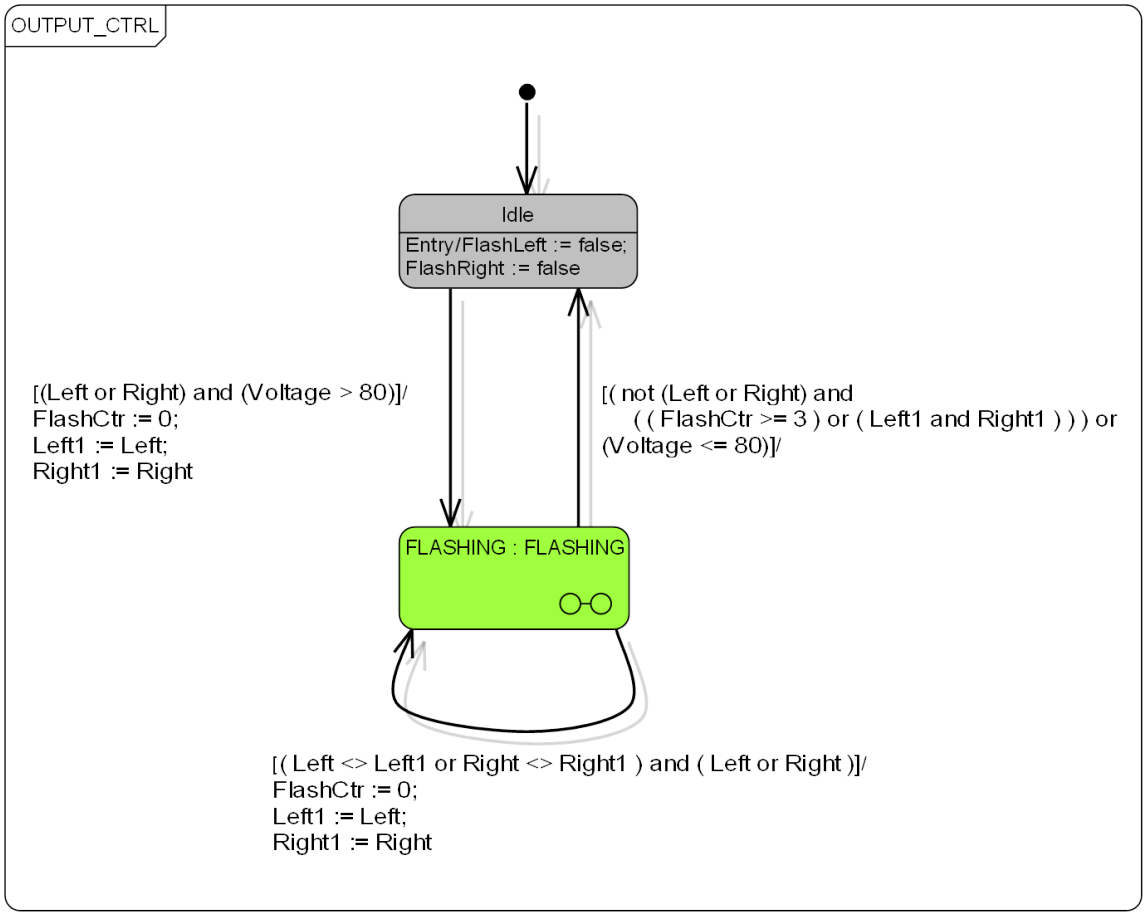


Figure 8: State machine switching indication lights.

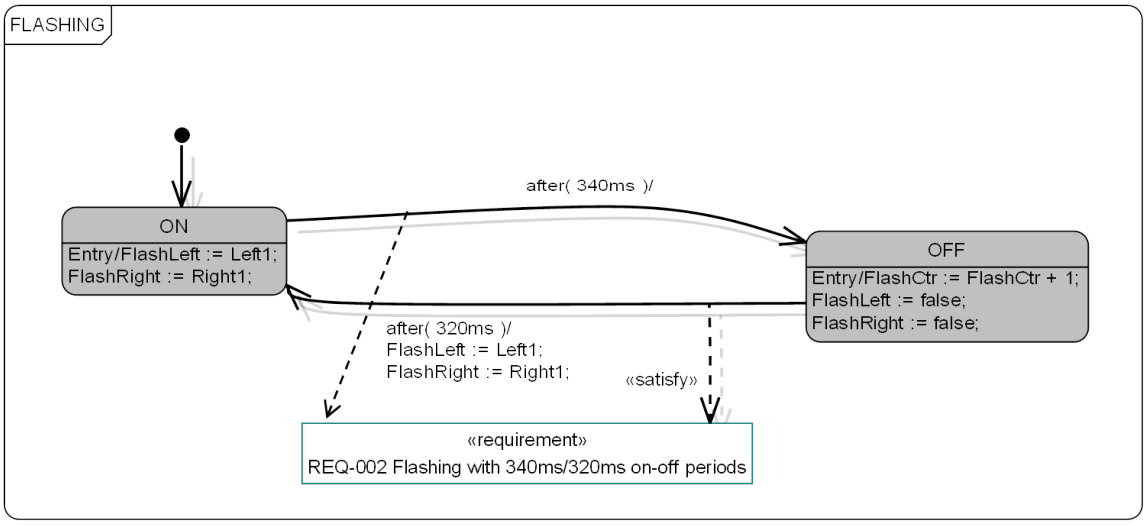


Figure 9: Decomposition of control state FLASHING.

Model-Based testing for LTE Radio Base Station

Olga Grinchtein
Ericsson AB, Sweden
olga.grinchtein@ericsson.com

The presentation describes experiences of applying model-based testing to LTE Radio Base Station. The presentation shows results from MBT project which was carried out at the organization in Ericsson, which is responsible for integration and verification of LTE Radio Access Network. The "LTE, Long Term Evolution, is the next generation mobile network beyond 3G. LTE Radio Access Network consists of LTE Radio Base Station, which supports the LTE air interface and performs radio resource management. The presentation is focused on LTE feature, which requires combinatorial testing. The presentation describes what kind of problems we faced during modelling and concretization. We describe benefits and disadvantages of using Spec Explorer tool for modelling and test generation.

A. Petrenko, H. Schlingloff (Eds.): Eighth Workshop on
Model-Based Testing (MBT 2013)
EPTCS 111, 2013, pp. 29–29, doi:10.4204/EPTCS.111.2

Towards the Usage of MBT at ETSI

Jens Grabowski
University of Göttingen, Germany
grabowski@informatik.uni-goettingen.de

Victor Kuliamin
ISP RAS, Russia
kuliamin@ispras.ru

Alain-Georges Vouffo Feudjio
Thales, Germany
alain-georges.vouffofeudjio@thalesgroup.com

Antal Wu-Hen-Chang
Ericsson, Hungary
antal.wu-hen-chang@ericsson.com

Milan Zoric
ETSI, France
milan.zoric@etsi.org

In 2012 the Specialists Task Force (STF) 442 appointed by the European Telcommunication Standards Institute (ETSI) explored the possibilities of using Model Based Testing (MBT) for test development in standardization. STF 442 performed two case studies and developed an MBT-methodology for ETSI. The case studies were based on the ETSI-standards GeoNetworking protocol (ETSI TS 102 636) and the Diameter-based Rx protocol (ETSI TS 129 214). Models have been developed for parts of both standards and four different MBT-tools have been employed for generating test cases from the models. The case studies were successful in the sense that all the tools were able to produce the test suites having the same test adequacy as the corresponding manually developed conformance test suites. The MBT-methodology developed by STF 442 is based on the experiences with the case studies. It focusses on integrating MBT into the sophisticated standardization process at ETSI. This paper summarizes the results of the STF 442 work.

1 Introduction

Driven by technological advances and an ever-growing need for software and systems quality improvements, MBT has matured in the last decade from a topic of research into an industrial technology. MBT has been successfully used for the automatic generation of test documentation and test scripts in a wide range of application areas including information and communication technology, embedded systems and medical software. This trend is reflected by the availability of various commercial tools and increasing efforts in MBT-related standardization. The utilization of MBT in industry show significant gains in productivity, in particular due to savings in the test maintenance phase.

In 2010, the ETSI Technical Committee (TC) on Methods for Testing and Specification (MTS) published a first ETSI standard on MBT (ES 202 951) [3] as the result of a joint effort of different stakeholders at ETSI including MBT tool vendors, major users, service providers, and research institutes. In order to enable the use of this technology at ETSI, the applicability of MBT in ETSI processes has to be shown and methodology guidelines for applying MBT in the context of standardized test development are needed. For this purpose ETSI TC MTS started in 2012 STF 442. STF 442 consists of five experts from industry and academia with 30 working days each. The work was conducted from February 2012 to December 2012. STF 442 performed two case studies from the ETSI domains Intelligent Transportation Systems (ITS) and Universal Mobile Telecommunications System (UMTS) and used the gained experience for developing ETSI MBT methodology guidelines.

A. Petrenko, H. Schlingloff (Eds.): Eighth Workshop on
Model-Based Testing (MBT 2013)
EPTCS 111, 2013, pp. 30–34, doi:10.4204/EPTCS.111.3

In the following, we present the case studies, describe the methodology and discuss problems encountered when applying MBT in the case studies.

2 Case Studies

The following four MBT tools have been used for the case studies:

- **Conformiq Designer** is the MBT tool of Conformiq Inc. [1]. Conformiq models are written in a combination of Java code and UML statecharts, i.e., in the Conformiq Modeling Language (QML). The models describe the expected external behavior of the System Under Test (SUT). Java code is used to describe the data processing of the SUT, to declare data types and classes, to express arithmetics and conditional rules as well as others. UML statecharts are used to capture high-level control flow and life cycle of objects. The core of Conformiq Designer is its semantics driven, symbolic execution based test generation algorithm. The algorithm traverses a part of the (usually infinite) state space of the system model. The test generation heuristics that Conformiq Designer uses realize various well-known test generation strategies, e.g., requirements coverage, transition coverage, branch coverage, atomic condition coverage, and boundary value analysis.
- **Microsoft Spec Explorer** for VisualStudio 2010 is a Microsoft MBT tool [10]. Spec Explorer uses state-oriented model programs that are coded in C#. Test generation is performed by exploring the state space of the system model and recording the traces. These traces are transformed into test cases. The main technique for dealing with state space explosion provided by Spec Explorer is scenario-based slicing. A scenario limits the potential executions of the model state graph, while preserving the test oracle and other semantic constraints from the system model. Slicing scenarios along with test data used as input for model operations are defined in the scripting language Cord.
- **Sepp.med MBTsuite** is the MBT framework from the sepp.med GmbH [11]. For applying MBTsuite, a graphical model of the SUT has to be provided. In our case studies, UML state and activity diagrams have been used. MBTsuite excutes models and transforms the execution traces into test cases. Apart from full path coverage, other generation strategies are available (e.g. guided generation, random generation). If defined in the model, guard conditions and priorities are taken into account at execution time. Thus, only logically consistent execution traces are obtained and processed into test cases. It is possible to filter the execution traces prior to test case generation using several built-in heuristics like, e.g., node coverage, edge coverage, requirement coverage, but also heuristics based on test management information (costs, duration).
- **Fraunhofer MDTester** is an academic MBT tool developed by the Fraunhofer FOKUS competence center MOTION [9]. MDTester is part of Fokus!MBT, a flexible and extensible test modeling environment based on the UML Testing Profile (UTP), which facilitates the development of model-based testing scenarios for heterogeneous application domains. MDTester is a modeling tool that guides the development of UTP models. UTP models are test models and not system models, i.e., they include tester knowledge like, e.g., setting of test verdicts, knowledge about test components, or default behavior. For modeling, MDTester provides the following diagrams types: test requirements diagram (based on class diagram), test architecture diagram (based on class diagram), test data diagram (based on class diagram), test architecture diagram, test behavior diagram (based on sequence and activity diagrams).

The case studies were based on ITS and UMTS protocols standardized by ETSI. In addition, STF 442 conducted the academic example of a simple automated teller machine to gain experience with the tools.

For the ITS-based case study, conformance tests for the location service functionality of the GeoNetworking protocol (ETSI TS 102 636) [6] have been generated from previously developed models. The GeoNetworking protocol belongs to the ITS network layer. The location service functionality is used to discover units with certain addresses and to maintain data on their geographical location.

The Rx interface (ETSI TS 129 214) [8] of UMTS provides the base for the second case study. The Rx interface supports the transfer of session information and policy/charging data between Application Function and Policy/Charging Rules Function on top of the Diameter protocol.

In both case studies, the modeled behavior of the System Under Test (SUT) can be described with approximately 12 control states and a slightly higher number of transitions between them. However, the main complexity of the SUT-model behavior is related to data stored and used during operation. For the GeoNetworking case study, this data refers to addresses and geographical locations; whereas session settings and policy rules are most important the behavior of the Rx interface case study.

Two different approaches have been used for modeling. The first approach started from the manually developed test purposes [7, 5] and resulted in SUT-models sufficient to cover all the test purposes, meanwhile adding some more details from standard requirements. The second approach was based on the requirements in the base standard. The constructed SUT model tried to reflect all of them in their behavior. Both approaches were successful in a sense that the models were suitable for test generation.

In spite of the fact the different tools use different formalisms as input for SUT models and provide different means to control test generation, all tools managed to generate test suites that cover almost the manually developed test purposes. Thus, from a technical point of view, modern MBT tools are able to support test development in standardization.

The case studies are documented in [4]. The report includes detailed descriptions of the SUT behavior and the models, a discussion of modeling approaches, the generated tests, and overall evaluation.

3 Methodology Guidelines

The second goal of the STF work was the development of methodology guidelines for an MBT-based development of conformance tests at ETSI [2]. ETSI has a very sophisticated test development procedure shown on the left side of Figure 1. Test development starts with the identification of requirements followed by the creation of Implementation Conformance Statement (ICS) and Interoperable Function Statement (IFS). ICS/IFS define implementation options for a standard. In the testing process, they are used for test case selection. The ICS/IFS creation is followed by the specification of the test suite structure, which in most cases arises from the functionality to be tested. Afterwards, high-level test descriptions, i.e., test purposes, are stepwise refined leading to the test cases, which are finally validated. The test development steps lead to documents represented by the ellipses in the middle of Figure 1.

The integration of MBT in the ETSI process is shown on the right side of Figure 1. The modeling for testing is based on standard and requirements. If possible, implementation options (i.e., ICS/IFS) are considered in modeling. The modeling process can be seen as an additional validation step for the standard, the requirements and the implementation options. Problems in modeling may identify ambiguities in the standard or untestable requirements. The model serves as input for the test generation. Problems identified during test generation or in the generated tests may identify problematic requirements or require adaptations in the SUT model. For integrating MBT into the ETSI test development process, documents describing test suite structure, test purposes, test descriptions and test cases have to be generated.

Even though this embedding of MBT into the ETSI test development process looks straightforward, several issues need to be solved before MBT can improve the existing process. A main problem is the

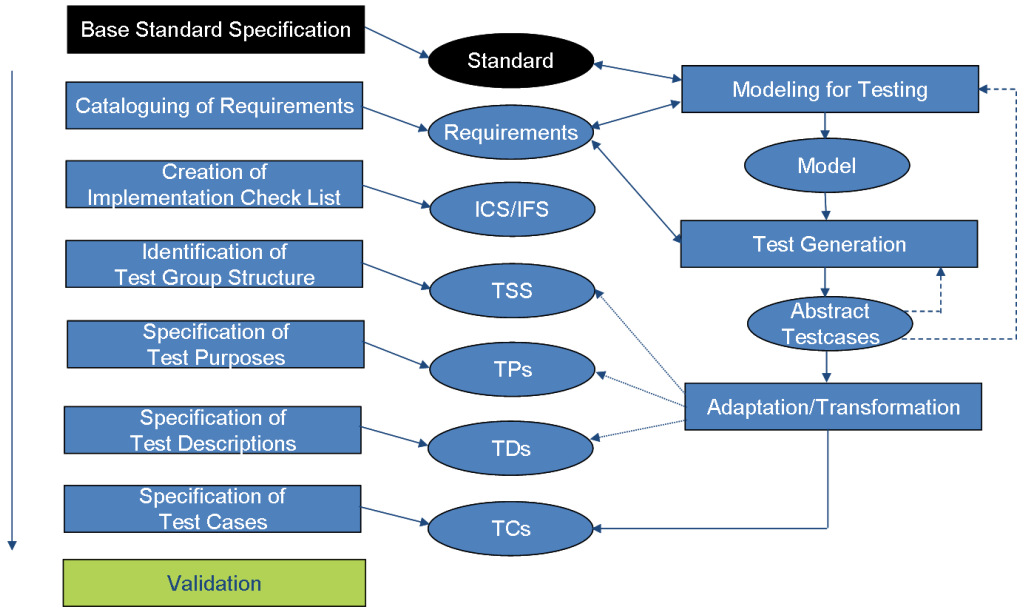


Figure 1: Using MBT within the ETSI test development process

maintenance and consistency of model and test documents. On the one hand, MBT only requires maintenance and further development of models while test cases are generated and not manually developed. On the other hand, each test case is an asset and its implementation can be very costly. Reviews and discussions are therefore mainly based on individual test descriptions and not on models. Another issue is the selection of a modeling language. Even though all MBT tools used for the case studies allow state-oriented modeling, the input languages differ considerably. A pragmatic solution to this problem may include the standardization of an ETSI modeling language.

In addition to issues regarding the test development process, the ETSI MBT methodology guidelines also offer guidance for identification and modeling of requirements, establishing traceability from models to standard requirements, choosing model scope and abstraction level, selecting test coverage criteria, improving maintainability and parameterization of generated tests, as well as assessing the quality of models and tests.

4 Summary and Conclusions

STF 442 has successfully applied MBT to generate conformance tests for two ETSI protocols. Both case studies have been performed with all tools. All tools were able to generate to test suites having an adequacy level comparable with manually designed tests. Based on the case studies, ETSI MBT methodology guidelines have been developed. The methodology guidelines focus on integrating MBT into the standardization process at ETSI. Some challenges have been identified during the STF work:

- An efficient usage of MBT in standardization requires significant expertise in several areas, like e.g., the domain of the SUT, modeling, MBT tool application, and test development. Experts experienced in all areas are difficult to find.
- There exists an abstraction gap between automatically generated and manually specified test cases. Manually test cases are usually more maintainable and can be subject of a review. By considering

parameterization, manually developed test cases allow an easy adaptation to different implementations of a standard. Solving this issue can be seen as a requirement for future MBT tools.

- The conformance test development process at ETSI is tightly intertwined with test suite maintenance issues and with handling each test case as a separate artifact. Test cases are designed individually and are subject of discussions and reviews. In contrast to the ETSI process, one of the main MBT advantages is the transfer of all maintenance work to the modeling, while tests are considered to be generated automatically as often as needed, i.e., maintenance of automatically generated tests is not necessary. For ETSI, taking full advantage of MBT may require new processes changing from the test case centric development to model standardization and maintenance.

Acknowledgements

The authors thank ETSI TC MTS for supporting the work presented in this paper.

References

[1] *Conformiq Inc.: Conformiq Inc. products Web page for Conformiq Designer*. http://www.conformiq.com/products/conformiq-designer.

[2] *ETSI Draft EG 203 130: "Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Methodology for standardized test specification development" V.1.1.1 (2013-02).*

[3] *ETSI ES 202 951: "Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations" V1.1.1 (2011-07).* http://www.etsi.org/deliver/etsi_es/202900_202999/202951/01.01.01_60/es_202951v010101p.pdf.

[4] *ETSI TR/MTS 103 133: "Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Application of MBT in ETSI case studies" V.1.1.1 (2013-02).*

[5] *ETSI TS 101 580-2: "IMS Network Testing (INT); Diameter Conformance testing for Rx interface; Part 2: Test Suite Structure (TSS) and Test Purposes (TP)" V1.1.1 (2012-04).* http://www.etsi.org/deliver/etsi_ts/101500_101599/10158002/01.01.01_60/ts_10158002v010101p.pdf.

[6] *ETSI TS 102 636-4-1: "Intelligent Transport Systems (ITS); Vehicular communications; GeoNetworking; Part 4: Geographical addressing and forwarding for point-to-point and point-to-multipoint communications; Sub-part 1: Media-Independent Functionality " V1.1.1 (2011-06).* http://www.etsi.org/deliver/etsi_ts/102600_102699/1026360401/01.01.01_60/ts_1026360401v010101p.pdf.

[7] *ETSI TS 102 871-2: "Intelligent Transport Systems (ITS); Testing; Conformance test specifications for GeoNetworking ITS-G5; Part 2: Test Suite Structure and Test Purposes (TSS&TP)" V1.1.1 (2011-06).* http://www.etsi.org/deliver/etsi_ts/102800_102899/10287102/01.01.01_60/ts_10287102v010101p.pdf.

[8] *ETSI TS 129 214: "Universal Mobile Telecommunications System (UMTS); LTE; Policy and charging control over Rx reference point" (3GPP TS 29.214) V10.6.0 (2012-03).* http://www.etsi.org/deliver/etsi_ts/129200_129299/129214/10.03.00_60/ts_129214v100300p.pdf.

[9] *Fraunhofer FOKUS competence center MOTION: MOTION Web page*. http://www.fokus.fraunhofer.de/en/motion/index.html.

[10] *Microsoft Corporation: Microsoft Developer Network Web pages for Spec Explorer.* http://msdn.microsoft.com/en-us/library/ee620411.

[11] *sepp.med GmbH: sepp.med products Web page for MBTsuite.* http://www.seppmed.de/produkte/mbtsuite.html.

Testing Java implementations of algebraic specifications

Isabel Nunes
Faculty of Sciences, University of Lisbon
Lisboa, Portugal
in@di.fc.ul.pt

Filipe Luís
Faculty of Sciences, University of Lisbon
Lisboa, Portugal
fluis@di.fc.ul.pt

In this paper we focus on exploiting a specification and the structures that satisfy it, to obtain a means of comparing implemented and expected behaviours and find the origin of faults in implementations. We present an approach to the creation of tests that are based on those specification-compliant structures, and to the interpretation of those tests' results leading to the discovery of the method responsible for an eventual test failure. Results of comparative experiments with a tool implementing this approach are presented.

1 Introduction

The development and verification of software programs against specifications of desired properties is growing weight among software engineering methods and tools for promoting software reliability. In particular, finding the software element containing a given fault is highly desirable and several approaches exist that tackle this issue, that can be quite different in the way they approach the problem.

ConGu [14, 15] is both an approach and a tool for the runtime verification of Java implementations of algebraic specifications. It verifies that implementations conform to specifications by monitoring method executions in order to find any violation of automatically generated pre and post-conditions.

The ConGu tool [7] picks a module of axiomatic specifications, together with a Java implementation and a refinement that maps specifications to Java types, and responds to an erroneous implementation by outputing the specification constraint that was violated; this is often insufficient to find the faulty method, because all methods involved in the violated constraint become equally suspect.

A ConGu companion tool – the GenT tool [3, 4] – generates JUnit test cases from ConGu specifications. Generating test cases that are known to be comprehensive, i.e. that cover all constraints of the specification, as GenT does, is a very important activity, because the confidence we may gain on the correction of the software we use greatly depends on it. But, in order for these tests to be of effective use, we should be able to use their results to localize the faulty components. Here again, executing the JUnit tests generated by GenT fails to give the programmer clear hints about the faulty method – all methods used in failed tests are suspect. The ideal result of a test suite execution would be the exact localization of the fault.

In this paper we enrich ConGu, by giving it the capability of locating the methods that are responsible for detected faults. We present a technique that builds upon structures satisfying the specification to obtain a means to observe the implemented behaviour against the intended one, and to locate faulty methods in implementations. Unlike several existing approaches, ours does not inspect the executed code; instead, it exploits the specification and conforming structures in order to be able to interpret some failures and discover their origin.

A tool was built – the Flasji tool – that implements the presented technique, and a comparative experiment was undertaken to evaluate its results. A summary of these results, which were very encouraging, is presented in this paper.

A. Petrenko, H. Schlingloff (Eds.): Eighth Workshop on
Model-Based Testing (MBT 2013)
EPTCS 111, 2013, pp. 35–50, doi:10.4204/EPTCS.111.4

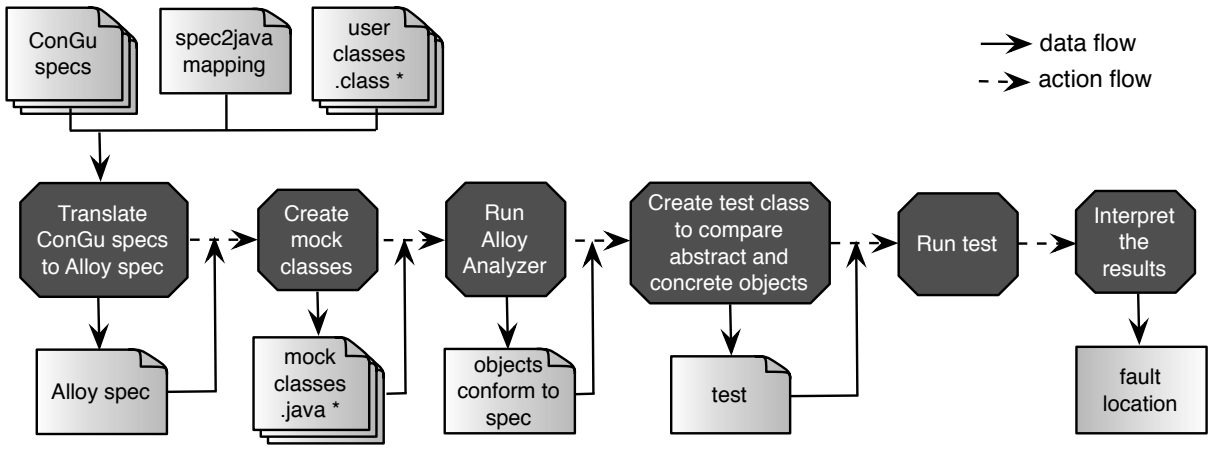


Figure 1: Overview of the Flasji approach.

The unit of fault Flasji is able to detect is the method, leaving to the programmer the task of identifying the exact instruction within it that is faulty. If more than one fault exists, the repeated application of the process, together with the correction of the identified faulty method, should be adopted. In what concerns integration testing strategy, Flasji applies an incremental one in the sense that the Java types implementing the specification are not tested all together; instead, each one is tested conditionally, presuming all others from which it depends are correctly implemented. This incremental integration is possible since the overall specification is given as a structured collection of individual specifications (a ConGu module), whose structure is matched by the structure of the Java collection of classes that implements it.

The remainder of the paper is organized as follows: section 2 introduces the ConGu specification language through an example that will be used throughout the paper, and gives an overview of the Flasji approach; section 3 details every Flasji step, from picking a specification module and corresponding implementation, to the identification of the faulty method, explaining the several items Flasji produces; an evaluation experiment of the Flasji tool is presented in section 4 where results are compared with the ones obtained using two other tools; in section 5 we focus our discussion on relevant aspects related to the work presented in this paper; finally, section 6 concludes.

2 Approach Overview

In this section we give a general overview of the Flasji approach. An example is introduced that will be used throughout the paper.

2.1 The approach in a nutshell

As illustrated in figure 1, the Flasji approach integrates a series of steps from an initial input comprising a module of ConGu specifications and corresponding Java implementations (together with a mapping defining correspondences between the two), to a final output comprising the method identified as the one containing the fault, whenever possible, and a list of other methods suspect of being faulty. The

whole process leading from the initial input to the final output is automated, without any further user intervention.

The main strategy underlying the Flasji process is the comparison between what we call "abstract" and "concrete" objects; the former are objects that are well-behaved in the sense that they conform to the specification, while the latter are objects that behave according to the classes implementing the specification, which we want to investigate for faults.

We capitalize on the Alloy Analyzer [1] tool which is capable of finding structures that satisfy a collection of constraints – a specification. The specifications this tool works with are written in the Alloy [11] language.

Flasji begins by translating the ConGu specification module into an Alloy specification, in order to be able to, in a posterior phase, generate structures satisfying it. It then creates Java classes whose instances will represent objects satisfying the specification (the "abstract" objects) – these classes are called "mock" classes; in order for "abstract" objects to represent structures that satisfy the specification, they are given the ability of storing and retrieving the results of applying each and every operation of the specification, as will be seen later.

A third step feeds the Alloy Analyzer tool with the specification, asking the tool for a collection of structures satisfying the specification. This collection will be used in the next step to define the abstract objects, which will present the expected, correct, behaviours.

In a fourth step, a test class is created that contains instructions to instantiate both the mock classes and the implementation classes given as input, and to compare the behaviour of the "concrete" objects against the "abstract" ones, in order to identify the faulty method. Flasji then executes this test class and interprets its results to obtain the faulty method.

Remember that all these steps are automatically processed, thus transparent to the Flasji user. Section 3 describes them in detail.

2.2 A specification and corresponding implementation

Simple sorts, sub-sorts and parameterized sorts can be specified with the ConGu specification language, and mappings between those sorts and Java types, and between those sorts' operations and Java methods, can be defined using the ConGu refinement language.

We present a classical yet rich example, of a ConGu specification of the SortedSet parameterized data type, representing a set of Orderable elements, together with the specification for its parameter (figure 2).

In a specification, we define *constructors*, *observers* and other operations, where constructors compose the minimal set of operations that allow to build all instances of the sort, observers can be used to analyse those instances, and the other operations are usually comparison operations or operations derived from the others; depending on whether they have an argument of the sort under specification (self argument) or not, constructors are classified as *transformers* or *creators* (transformers are also referred to as *non-creator constructors*).

All operations that are not constructors must have a self argument. Any function can be partial (denoted by $\rightarrow ?$), in which case a *domains* section specifies the conditions that restrict their domain. *Axioms* define every value of the type through the application of observers to constructors – see, e.g., the axiom isEmpty(empty()); that specifies that the result of applying the observer operation isEmpty to a SortedSet instance obtained with the creator constructor empty is true, and the axiom **not** isEmpty(insert(S, E)); saying that the result of applying isEmpty to any SortedSet instance to which the transformer constructor

```
specification SortedSet[TotalOrder]
  sorts
    SortedSet[Orderable]
  constructors
    empty: --> SortedSet[Orderable];
    insert: SortedSet[Orderable] Orderable --> SortedSet[Orderable];
  observers
    isEmpty: SortedSet[Orderable];
    isIn: SortedSet[Orderable] Orderable;
    largest: SortedSet[Orderable] -->? Orderable;
    ...
  domains
    S: SortedSet[Orderable];
    largest(S) if not isEmpty(S);
  axioms
    E, F: Orderable;  S: SortedSet[Orderable];
    isEmpty(empty());
    not isEmpty(insert(S, E));
    not isIn(empty(), E);
    isIn(insert(S,E), F) iff E = F or isIn(S, F);
    largest(insert(S, E)) = E if isEmpty(S);
    largest(insert(S, E)) = E if not isEmpty(S) and geq(E, largest(S));
    largest(insert(S, E)) = largest(S) if
                            not isEmpty(S) and not geq(E, largest(S));
    ...
end specification
```

```
specification TotalOrder
  sorts
    Orderable
  observers
    geq: Orderable Orderable;
  axioms
    E, F, G: Orderable;
    E = F if geq(E, F) and geq(F ,E);
    geq(E, F) if E = F;
    ...
end specification
```

Figure 2: Parts of the ConGu specifications for the SortedSet parameterized data type and its parameter.

```
public class TreeSet<E extends IOrderable<E>> {
       public TreeSet() {...}
       public void insert(E e) {...}
       public boolean isEmpty() {...}
       public boolean isIn(E e) {...}
       public E largest() {...}
       ...
 }
```

```
public interface IOrderable<E> {
       boolean greaterEq(E e);
 }
```

Figure 3: Excerpt from a Java implementation of the ConGu specification for SortedSet.

insert has been applied is false.

Generic Java class `TreeSet` in figure 3 represents a Java implementation of the SortedSet parameterized data type; in the same figure, interface `IOrderable` represents a Java type restraining the `TreeSet` parameter. We want to investigate the `TreeSet` class for faults, independent of any specific implementation of its parameter type.

The correspondence between ConGu and Java types must be defined in order for implementations to be checked. This correspondence is described in terms of *refinement mappings*; figure 4 shows a refinement mapping from the specifications SortedSet and TotalOrder (figure 2) to Java types `TreeSet` and `IOrderable` (figure 3).

These mappings associate ConGu sorts and operations to Java types and corresponding methods. The insert operation of sort SortedSet is mapped to the `TreeSet` class method with the same name with the signature **`void`** `insert(E e)`. Notice that the TotalOrder parameter sort is mapped to a Java type variable that is used as the parameter of the generic `TreeSet` implementation; this specific mapping is interpreted as constraining any instantiation of the `TreeSet` parameter to a Java type `Some` containing a method with signature **`boolean`** `greaterEq(Some e)`.

Detailed information about the ConGu approach can be found in [14, 15].

```
refinement <E>
  SortedSet[TotalOrder] is TreeSet<E> {
     empty: --> SortedSet[Orderable] is TreeSet();
     insert: SortedSet[Orderable] e:Orderable --> SortedSet[Orderable] is void insert(E e);
     ...
  }
  TotalOrder is E {
     geq: Orderable e:Orderable is boolean greaterEq(E e);
  }
end refinement
```

Figure 4: Refinement mapping from ConGu specifications to Java types.

3 Flasji step-by-step

As already said in the previous section, the main goal of the Flasji approach is to verify whether "concrete" Java objects behave the same as corresponding "abstract" ones; the deviations to the expected behaviour are interpreted in order to find the location of the faulty method.

Only one of the implementing classes is under verification – the *core* type, that is, the one that implements the *core* sort –, which is the one at the root of the class association graph (the `TreeSet` class in the example). Thus, both "abstract" and "concrete" objects will be created for this type, in order to compare behaviours. This does not apply for non-core types since they are not under verification. However, as we shall see, "abstract" parameter objects must be created.

Let us now detail the several steps of the Flasji approach.

3.1 Translating the ConGu specification module

Flasji creates an Alloy specification equivalent to the ConGu specification module, in order to be able, ahead in the process, to obtain a collection of objects that conform to the specification, thus defining expected, correct behaviour.

This step capitalizes on already existing work, referred to in the introduction of this paper, namely the GenT tool [3, 4], of which Flasji uses the ConGuToAlloy module.

3.2 Creating mock classes

Flasji creates mock classes for the Java types implementing the specification core and parameter sorts; these classes' instances will represent the "abstract" objects. In the running example, two mock classes must be created, one corresponding to the `TreeSet` class – `TreeSetMock` –, and other corresponding to the `IOrderable` interface – `OrderableMock`.

This mock class will be used to generate parameter objects that will be inserted not only in "abstract" sorted sets (`TreeSetMock` instances as explained below), but also in "concrete" ones (`TreeSet` instances); the idea, as said before, is to test the implementation of the core signature for any parameter instantiation that correctly implements the Orderable sort.

Each instance of a mock class defines an object conforming to the specification, including its "behaviour", that is, the results of applying to it all the operations of the type (respecting the corresponding domain conditions). Since we only compare "abstract" and "concrete" objects of the core type, fundamental differences exist between the mock class for this core type and the others. Let us see first the mock class for the Orderable parameter of the running example, which is a non-core type.

```
1 public class OrderableMock implements IOrderable⟨OrderableMock⟩{
2   private HashMap⟨OrderableMock, Boolean⟩ greaterEqResult = new HashMap⟨OrderableMock,
        Boolean⟩();
3   public boolean greaterEq(OrderableMock e) {
4     return greaterEqResult.get(e);}
5   public void add_greaterEq(OrderableMock e, Boolean result) {
6     greaterEqResult.put(e, result);
7   }
8 }
```

Listing 1: Mock class corresponding to the Orderable parameter sort.

For each method `X` corresponding to a specification operation X, an attribute is defined to keep the information about the results of `X`, for every combination of the method's parameters (see line 2 for the method `greaterEq` in interface `Iorderable`, corresponding to operation geq in sort Orderable); the `add_X` method "fills" that attribute (lines 5 and 6), and the `X` method retrieves the result for given values of the method's parameters (lines 3 and 4).

The class that represents "abstract" objects of the core type (class `TreeSetMock` in the example) is also generated. The idea here is not to use these "abstract" objects on both abstract and concrete contexts, as we do with parameter ones, but to use them to inform us, for every operation, of the results we should expect when applying corresponding methods to corresponding "concrete" objects in order to verify whether the latter behave as they should.

The fundamental difference lies in the information that core type "abstract" objects keep for operations whose result is of the core type (insert in the example). Since we want to be able to know whether the "concrete" `TreeSet` object that results from applying the `insert` method of the `TreeSet` class to a "concrete" object `concObj` is the correct one, we give the corresponding "abstract" object `absObj` information that allows us to verify it – we "feed" `absObj` with the "concrete" object that should be expected when applying that operation. Ahead in this paper we show how this is achieved; for now, we just present the `TreeSetMock` mock class in listing 2, where attribute and methods in lines 19 to 24 allow "abstract" objects to keep and inform about "concrete", expected results of applying `insert` for different values of the method's parameter:

```
1 public class TreeSetMock ⟨T⟩{
2
3   // operations whose result is of a non-core type
4   private HashMap⟨T,Boolean⟩ isInResult = new HashMap⟨T,Boolean⟩();
5   private boolean isEmptyResult;
6   private T largestResult;
7
8   public boolean isIn (T e){return isInResult.get(e);}
9   public void add_isIn (T e, Boolean result){
10    isinResult.put(e, result);}
11  public boolean isEmpty(){return isEmptyResult;}
12  public void add_isEmpty(boolean result){
13    isEmptyResult = result;}
14  public T largest(){return largestResult;}
15  public void add_largest(T result){
16    largestResult = result;}
17
18  // operation whose result is of the core type
19  private HashMap⟨T,TreeSet⟨T⟩⟩ insertResult = new HashMap⟨T,TreeSet⟨T⟩⟩ ();
20
21  public TreeSet⟨T⟩ insert (T e){
22    return insertResult.get(e); }
23  public void add_insert (T e, TreeSet⟨T⟩ concVal){
24      insertResult.put(e, concVal); }
```

```
25 }
```

Listing 2: Mock class corresponding to the SortedSet sort.

Line 19 declares and initializes the attribute that will store the information about the results of method `insert` – for each value of the parameter `T e`, it will store a "concrete" object. Methods `insert` and `add_insert`, in lines 21 to 24, allow to retrieve and define, respectively, the result of `insert` for every value of the operation's parameter.

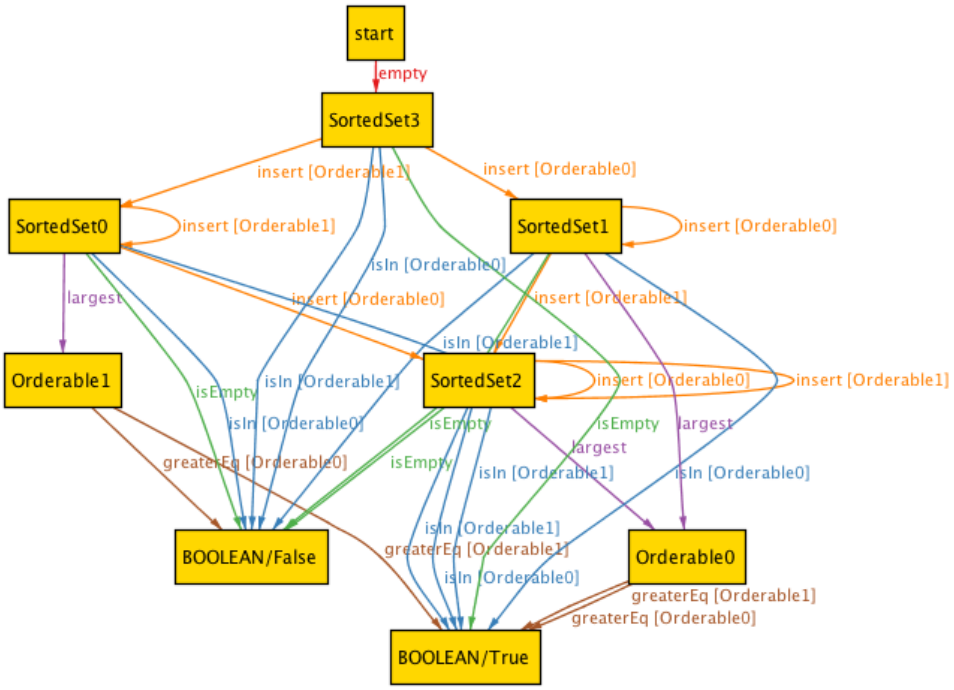


Figure 5: A structure satisfying the SortedSet specification module, found by the Alloy Analyzer.

3.3 Obtaining a collection of instances satisfying the specification

In order to obtain a collection of instances of the specification sorts that conform to our specification, Flasji capitalizes on the Alloy Analyzer tool [11] which, if such a finite collection exists, is capable of generating it (such a finite collection does not exist e.g. in the case of a specification of an unbounded stack).

The results presented in this paper assume Flasji asks the Alloy Analyzer to generate a structure with a fixed number of objects of the core type; work is under way to optimize the determination of the number of objects that should be considered of each type.

Figure 5 shows an example for the SortedSet specification. This collection consists of 2 Orderable instances and 4 SortedSet ones. The result of every applicable operation is defined for each of these instances (e.g. SortedSet2 contains the two Orderables, and it is obtained by inserting Orderable0 into SortedSet0 or else from inserting Orderable1 into SortedSet1).

These instances define correct, expected behaviour; in the next steps, Flasji will use the non-core sort ones (Orderable instances in the example) to create mock "abstract" objects that will represent well-

behaved objects, and uses the core sort ones (SortedSet instances in the example) to create mock "abstract" objects (`TreeSetMock` and "concrete" corresponding ones (`TreeSet` instances in the example) instances in the example) that will be compared. Let us see how.

3.4 Creating the test class

Flasji generates a test class containing instructions to:

1. create "abstract" objects corresponding to the objects composing the Alloy structure that conforms to the specification;
2. create "concrete" objects of the core type that correspond to the "abstract" ones (by using the corresponding concrete constructors); and
3. compare the behaviour of these "abstract" and "concrete" objects, by observing them in equal circumstances, that is, by applying corresponding methods and comparing the results.

By compiling and executing this test class, Flasji will be able to get information that it will interpret in order to find the faulty method, as explained ahead. First let us see how Flasji acomplishes this test class creation task.

3.4.1 Creating abstract objects

Listing 3 shows part of the generated test class, namely the creation of the "abstract" objects according to the Alloy structure defined in figure 5: two `OrderableMock` instances (lines 4 and 5) and four `TreeSetMock` ones (lines 11 to 14).

Lines 6 to 9 show `OrderableMock` objects being initialized – because the only method of this type is `greaterEq`, only the method `add_greaterEq` is invoked over each "abstract" object, for every possible value of its parameter, in order to give these objects the information about the expected, correct, results.

We postpone the initialization of the `TreeSetMock` objects because it implies previous creation of the corresponding concrete objects.

```
1  @Test
2  public void abstractVSconcreteTest () {
3    //IOrderable Mocks
4    OrderableMock orderable0 = new OrderableMock();
5    OrderableMock orderable1 = new OrderableMock();
6    orderable0.add_greaterEq(orderable0, true);
7    orderable0.add_greaterEq(orderable1, true);
8    orderable1.add_greaterEq(orderable0, false);
9    orderable1.add_greaterEq(orderable1, true);
10   //Abstract objects TreeSet
11   TreeSetMock ⟨OrderableMock⟩ sortedSet3 = new TreeSetMock ⟨OrderableMock⟩();
12   TreeSetMock ⟨OrderableMock⟩ sortedSet0 = new ...;
13   TreeSetMock ⟨OrderableMock⟩ sortedSet1 = new ...;
14   TreeSetMock ⟨OrderableMock⟩ sortedSet2 = new ...;
```

Listing 3: (Part of) the test class – building the "abstract" objects (incomplete).

3.4.2 Creating concrete objects

Flasji also builds "concrete" objects for the core sort. For each "abstract" object of the core sort there will exist a corresponding "concrete" one, which will be built using the corresponding constructor methods (see listing 4).

For example, according to the structure in figure 5, the sortedSet0 instance of sort SortedSet can be obtained by application of the creator constructor empty followed by application of the transformer constructor insert with parameter orderable1; complying with this (see lines 7 and 8), we build the corresponding "concrete" object `concSortedSet0` using the java constructor `TreeSet⟨OrderableMock⟩()`, which corresponds to the creator constructor empty, and apply to it the method `insert`, which corresponds to the transformer constructor insert, with parameter `orderable1`. Whenever there are several ways to build an object, the shortest path is chosen.

```
1 @Test
2 public void abstractVSconcreteTest () {
3   ...
4   //Create concrete objects TreeSet
5   TreeSet⟨OrderableMock⟩ concSortedSet0 = new TreeSet ⟨OrderableMock⟩();
6   concSortedSet0.insert(orderable1);
7   TreeSet⟨OrderableMock⟩ concSortedSet0_1 = new ...;
8   concSortedSet0_1.insert(orderable1);
9   ...
10  TreeSet⟨OrderableMock⟩ concSortedSet0_5 = new ...;
11  concSortedSet0_5.insert(orderable1);
12  // three more to go (concSortedSet3, 1 and 2)...
```

Listing 4: Continuing... building the "concrete" objects (incomplete).

Notice that, since methods will be applied to these "concrete" objects in order to verify their behaviour, as many copies of a given "concrete" object are created as methods applied to it, in order to cope with undesired side effects.

3.4.3 Back to abstract objects

Now that "concrete" objects are already created, we can initialize the `TreeSetMock` objects:

```
1 @Test
2 public void abstractVSconcreteTest () {
3   ...
4   //Initializing sortedSet abstract objects
5   sortedSet0.add_isEmpty(false);
6   sortedSet0.add_largest(orderable1);
7   sortedSet0.add_isIn(orderable0, false);
8   sortedSet0.add_isIn(orderable1, true);
9   sortedSet0.add_insert(orderable0, concSortedSet2);
10  sortedSet0.add_insert(orderable1, concSortedSet0);
11 // three more to go (sortedSet3, 1 and 2)...
```

Listing 5: Continuing... initializing `TreeSetMock` objects.

Lines 5 to 8 "feed" the `sortedSet0` object with the information that it represents a sorted set that is not empty, whose largest element is the `orderable1` object, and that it contains `orderable1` but not `orderable0`, as would be expected by inspection of the structure in figure 5. In the case of object `sortedSet3`, which is empty as can be seen in figure 5, the instruction invoking the method `add_largest()` over it would not be generated, since the operation `largest` is undefined for that object.

These informations will be used later on to obtain the values that are expected to be the results of the corresponding methods when applied to the "concrete" object corresponding to `sortedSet0`, which, by construction, is `concSortedSet0` (or any of its copies).

Line 9 "feeds" `sortedSet0` with the information about which "concrete" object should be expected after inserting `orderable0` in the "concrete" object that corresponds to `sortedSet0` (`concSortedSet0`

or any of its copies) – the expected result is `concSortedSet2`. In the same way, in line 10, the expected result of inserting `orderable1` in the "concrete" object that corresponds to `sortedSet0` is defined to be itself.

3.4.4 Comparing abstract and concrete objects

In a next step, Flasji generates instructions in the test class that invoke all possible operations over the "abstract" and "concrete" objects and compare the results:

```
1  @Test
2  public void abstractVSconcreteTest () {
3    ...
4    //Compare concrete with corresponding abstract
5    assertTrue(concSortedSet0_1.isEmpty() == sortedSet0.isEmpty());
6    assertTrue(concSortedSet0_2.largest() == sortedSet0.largest());
7    assertTrue(concSortedSet0_3.isIn(orderable1) == sortedSet0.isIn(orderable1));
8    assertTrue(concSortedSet0_4.isIn(orderable0) == sortedSet0.isIn(orderable0));
9    concSortedSet0_5.insert (orderable1);
10   assertTrue(concSortedSet0_5.equals(sortedSet0.insert (orderable1)));
11   //three more to go (concSortedSet3, 1 and 2)...
```

Listing 6: Continuing... comparing "abstract" and "concrete" objects (incomplete).

The JUnit method `assertTrue` is used to generate an `AssertionError` exception whenever the behaviour of the "concrete" objects is not as expected, that is, whenever the results of methods invoked over "concrete" objects are different from the ones indicated by their "abstract" counterparts.

Lines 5 to 8 show the comparison between the `sortedSet0` "abstract" object and its "concrete" counterpart `concSortedSet0` using each `TreeSet` method whose result type is not `TreeSet` nor **`void`**. Since all these results are of primitive types or of the parameter type `OrderableMock`, Flasji uses == to compare between `sortedSet0` and `concSortedSet0` results.

Lines 9 and 10 show the comparison between "abstract" and "concrete" objects using (the only) operation with a core result type – `insert`. As already referred, to verify whether a given operation whose result is of the core type is well implemented, we compare the "concrete" object the method returns, with the "concrete" object that it should return. Since `insert` is implemented with a **`void`** result type, we must first invoke the method using the "concrete" object as a target, and then we compare (using `equals`) its new state with the "concrete" object that, according to the "abstract" corresponding object, should be the correct result.

Since the ultimate goal of this test class is to find the method containing the fault, it should be possible to reason about the results of all these comparisons, so we must be able to test all the `assert` commands. Although we do not show it in this paper due to space limitations, enclosing each `assertTrue` invocation in a **`try`**–**`catch`** block that catches `AssertionError` exceptions, allows to collect all results which will help composing a final test diagnosis.

A final note before continuing: whenever the module of Congu specifications input to Flasji includes more than one non-parameter type, e.g., the case where the input includes a core sort C and one non-core, non-parameter sort N, the class implementing C is verified for faults considering that the class implementing N is correct. No mock class is built for N, hence no "abstract" `N` objects are created; only "concrete" `N` objects are. For methods that return `N` type results, "abstract" `C` objects are "fed" with the information about which `N` "concrete" object should be the expected result. Thus, comparison between actual and expected results relating these methods are achieved using `equals`. The running SortedSet example does not cover this kind of situation.

3.5 Running the test and interpreting the results

As soon as the test class is generated, Flasji compiles it and executes it. Then, it interprets the results of the tests. The interpretation is based upon the following observations:

1. whether several and varied observers (non-constructor operations) fail or only one fails – this is important to decide whether to blame a constructor or a given, specific, observer;
2. whether varied observers fail when applied to "concrete" objects created only by the constructor-creator, or when applied to objects that were also the target of non-creator constructors – this is important to decide which constructor is the faulty one.

The result interpretation algorithm inspects three data structures containing data collected during the execution of the test (whenever an `assertTrue` command fails):

- L_1 - Set of pairs $< obs; obj >$ that register that differences occurred between expected and actual behaviour, for given observer *obs* and object obj;
- L_2 - Set of pairs $< ncc; obj >$ that register that differences occurred between expected and actual behaviour, for given non-creator (transformer) constructor *ncc* and object obj;
- L_3 - Set of pairs $< cc; n >$ that register for every creator constructor *cc* the number of failed observations over objects uniquely built with *cc*;

If, when applied to concrete objects, more than one observer methods present results that are different from the ones expected ($(L_1 \cup L_2)$ contains pairs for more than one observer), we may infer that the method(s) used to build those concrete objects are ill-implemented, and that the problem does not come from some particular way of inspecting the objects. If the implementation of a given observer is wrong, one would not expect problems when inspecting the objects using the other observers, but only in the observations involving that particular one.

If a constructor-creator *cc* (in the running example, `TreeSet()` is the *cc* that implements the empty creator operation) is faulty, it is reasonable to think that the application of the other constructors over an object created with *cc* will most probably result in non-conformant objects, because the initial object is already ill-built. The information in L_3 allows us to focus on creator-constructors.

When no problems arise when observing a freshly created object, but they do arise when observing those objects after being affected by a given non-creator constructor *ncc* (`insert` in the running example), then one may point the finger to *ncc*.

if $(L_1 \cup L_2)$ contains pairs for more than 1 observer, **then**
  if there exists $< cc, i >$ in L_3 with $i > 0$, **then**
    if that pair $< cc, i >$ with $i > 0$ is unique, **then**
      cc is guilty;
    else
      inconclusive;
    endIf
  else
    for each non-creator constructor ncc_j **do**
      $L_{nccj} \leftarrow$ sub-set of L_2 containing only pairs from L_2 whose first element is ncc_j;
      Delete from L_{nccj} the pairs whose obj was not built using only ncc_j and a creator constructor;
      if L_{nccj} is not empty, **then**

```
          add ncc_j to the final set of suspects (FSS);
        endIf
      endFor
    endIf
    if #FSS = 1 then
      the guilty is the sole element of FSS;
    else
      inconclusive;
    endIf
  else
    if (L1 ∪ L2) is empty, then
      inconclusive;
    else
      the guilty is the sole observer in (L1 ∪ L2);
    endIf
  endIf
```

If the algorithm elects a guilty method in the end, then the user is given the identified method as the most probable guilty. In either case, the set FSS of (other) suspects is presented.

4 Evaluation

To evaluate the effectiveness of our approach, we applied it to two case studies – this paper's SortedSet running example, and a MapChain specification module and corresponding implementations. The Java classes implementing the designated sorts of both case studies where seeded with faults covering all the specification operations.

We put Flasji to run for every defective class, and registered the outputs.

We also tested those defective classes in the context of two existing fault-location tools – GZoltar [2, 17] and EzUnit4 [5, 18] –, that give as output a list of methods suspect of containing the fault, ranked by probability of being faulty. The tests suites we used were generated by the GenT [3, 4] tool (already referred to in this paper), from the ConGu specifications and refinement mappings; under given restrictions (e.g., the specification has finite models) GenT generates comprehensive test suites, that cover all specification axioms. GenT generated 20 test cases for the SortedSet case, and 17 for the MapChain one.

Finally we compared the three tools' results for every defective variation of each case study.

For each of the defective versions of the designated sorts implementations (for example, two different faults were seeded in SortedSet `isEmpty` method, three in MapChain `get` method, etc) table 1 shows:

- the number of tests (among the 20 JUnit tests that were generated by GenT for the SortedSet case, and 17 for the MapChain one) that failed when both GZoltar and EzUnit4 run them;
- whether the faulty method was ranked, by each tool, as most probable guilty (1_{st}), second most probable guilty (2_{nd}) or third or less probable (n_{th}). A fourth type of result – "No" – means the guilty method has not been ranked as suspect at all.

Flasji provided very accurate results in general (see also a summary in figure 6). The bad results in the three faults for method `get` of the MapChain case study (there were no suspects found whatsoever) are due to the fact that `equals` uses the `get` method, therefore becoming unreliable whenever method `get` is faulty. This case exemplifies the *oracle problem* (see section 5).

	Faulty method	failed tests	Faulty method ranked: Flasji	EzUnit4	GZoltar
SortedSet	isEmpty	5	1_{st}	n_{th}	2_{nd}
	isEmpty	1	1_{st}	1_{st}	2_{nd}
	isIn	1	1_{st}	n_{th}	1_{st}
	largest	7	1_{st}	1_{st}	1_{st}
	largest	1	1_{st}	1_{st}	2_{nd}
	private insert	2	No	n_{th}	2_{nd}
	public insert	5	1_{st}	n_{th}	2_{nd}
MapChain	get	4	No	n_{th}	1_{st}
	get	3	No	2_{nd}	1_{st}
	get	4	No	n_{th}	n_{th}
	isEmpty	2	1_{st}	2_{nd}	1_{st}
	isEmpty	1	1_{st}	n_{th}	1_{st}
	put	0	1_{st}	No	No
	put	1	1_{st}	1_{st}	2_{nd}
	put	2	1_{st}	1_{st}	2_{nd}
	remove	1	1_{st}	2_{nd}	2_{nd}
	remove	1	1_{st}	2_{nd}	2_{nd}

Table 1: Results of comparative experiments. "1_{st}", "2_{nd}" and "n_{th}" stand for first, second and third or worse, respectively. "No" means the faulty method has not been ranked as suspect.

Applying an alternative method of observation (see [16]) – one where the `equals` method is not used and, instead, only the outcomes of observers whose result is not of the core sort are used in comparisons – we obtain the right results for this case, i.e. `get` is ranked as prime suspect. However, the good results we had for the 3rd faulty `put` method and the 1st `remove` got worse – they are ranked second instead of first. These particular cases indicated `isEmpty` as prime suspect because the seeded fault of both those methods was the absence of change in the number of elements in the map whenever insertion/removal happens, which made `isEmpty` fail.

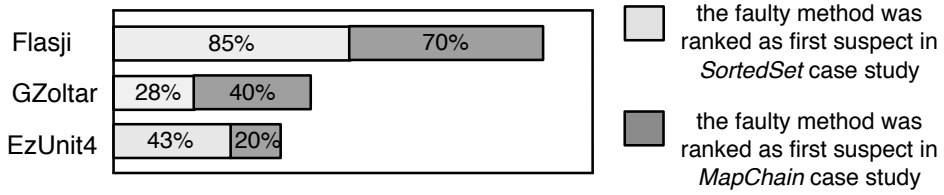


Figure 6: Summary of the evaluation experiment. The bars measure the success of each approach in ranking the faulty method as first suspect.

Another critical issue w.r.t. our approach is the one concerned with private methods. The fault in *private* method `insert` of the SortedSet case study, caused Flasji to rank the *public* `insert` method, instead, as the most probable suspect (in the particular implementation used, the public `insert` method is composed of one only statement which invokes the private `insert` method). As expected, private methods are not identified as suspects by Flasji because they do not directly refine any specification

operation (as defined in the refinement mapping from specifications to implementations); instead, the public, specified, methods that invoke them are identified.

A case worth mentioning is the one corresponding to the first seeded fault in the `put` method of the MapChain case study, where none of the seventeen GenT tests fail (the particular case that causes the error was not covered). As a consequence, neither EzUnit4 and GZoltar detected the fault; on the contrary, Flasji succeeded in detecting the guilty method.

5 Related work

The approach presented in this paper relies on the existence of structures satisfying the specification to supply the behaviour of objects to be used in tests. The structured nature of specifications, where functions and axioms are defined sort by sort, and where the latter are independently implemented by given Java types, is essential to the incremental integration style of Flasji.

Several approaches to testing implementations of algebraic specifications exist, that cover test generation ([6, 8, 9, 10, 12] to name a few), and many compare the two sides of equations where variables have been substituted by ground terms – differences exist in the way ground terms are generated, and in the way comparisons are made. The gap between algebraic specifications and implementations makes the comparison between concrete objects difficult, giving rise to what is known as the *oracle problem*, more specifically, the search for reliable decision procedures to compare results computed by the implementation. Whenever one cannot rely on the `equals` method, there should be another way to investigate equality between concrete objects. Several works have been proposed that deal with this problem, e.g. [9, 13, 19]. In [16] we tackle this issue by presenting an alternative way of comparing concrete objects, one that relies only in observers whose result is of a non-core sort. In some way this complies with the notion of observable contexts in [9] – all observers but the ones whose result is of the designated sort constitute observable contexts.

The unreliability of `equals` can also affect the effectiveness of the GenT tests [3] since this method is used whenever concrete objects of the same type are compared. One of the improvements we intend to make is to give Flasji the ability to test the `equals` method in order to make its use more reliable.

6 Conclusions

We presented Flasji, a technique whose goal is to test Java implementations of algebraic specifications and find the method that is responsible for some deviation of the expected behaviour.

Flasji capitalyzes on ConGu, namely using ConGu specification and refinement languages, and enriches it with the capability of finding faulty methods. It accomplishes the task through the generation of tests that are based on structures satisfying the specification. The behaviour of instances of the implementation is compared with the one expected, as given by those specification-compliant structures. The results of the comparisons are interpreted in order to find the method responsible for the fault.

An evaluation experiment was presented where Flasji results over two case studies, for which faults have been seeded in the implementing Java classes, are compared with two other tools' results when executed over comprehensive suites of tests. The encouraging results obtained in comparative studies led us to continue working on it, with the purpose of improving some negative aspects and weaknesses, some of which already identified and reported in this paper.

The following improvements, among others, are planned: (i) testing the implementation of the `equals` method, even if the specification module does not specify it, in order to be able to better rely on

its results, (ii) optimizing the determination of the number of objects of each type that an Alloy structure conforming to the specification should contain (the results here presented assumed Flasji asks the Alloy Analyzer to generate a structure with a fixed number of objects of the core type), and (iii) whenever there are several non-parameter types, apply the process several times, each considering one of them as the core type, and integrate the results (special cases as e.g. inter-dependent types, deserve attention).

References

[1] *Alloy Analyzer*. `http://alloy.mit.edu/alloy/`.

[2] R. Abreu, P. Zoeteweij & A.J.C. van Gemund (2009): *Spectrum-based Multiple Fault Localization*. In: *Proc. 24th IEEE/ACM ASE*, IEEE Computer Society, pp. 88–99, doi:10.1109/ASE.2009.25.

[3] F.R. Andrade, J.P. Faria, A. Lopes & A.C.R. Paiva (2012): *Specification-Driven Unit Test Generation for Java Generic Classes*. In: *IFM 2012*, *LNCS* 7321, Springer-Verlag, pp. 296–311, doi:10.1007/978-3-642-30729-4.

[4] F.R. Andrade, J.P. Faria & A. Paiva (2011): *Test Generation from Bounded Algebraic Specifications using Alloy*. In: *Proc. ICSOFT 2011*, 2, SciTePress, pp. 192–200.

[5] P. Bouillon, J. Krinke, N. Meyer & F. Steimann (2007): *EzUnit: A Framework for associating failed unit tests with potential programming errors*. In: *8th XP*, *LNCS* 4536, Springer, pp. 101–104, doi:10.1007/978-3-540-73101-6_14.

[6] K. Claessen & J. Hughes (2000): *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In: *Proc. of ICFP*, ACM SIGPLAN Notices, pp. 268–279, doi:10.1145/351240.351266.

[7] P. Crispim, A. Lopes & V. Vasconcelos (2011): *Runtime verification for generic classes with ConGu2*. In: *Proc. SBMF 2010*, *LNCS* 6527, Springer-Verlag, pp. 33–48, doi:10.1007/978-3-642-19829-8_3.

[8] R.K. Doong & P.G. Frankl (1994): *The ASTOOT Approach to Testing Object-Oriented Programs*. *ACM TOSEM* 3(2), pp. 101–130, doi:10.1145/192218.192221.

[9] M.C. Gaudel & P.L. Gall (2008): *Testing data types implementations from algebraic specifications*. *Formal Methods and Testing*, doi:10.1007/978-3-540-78917-8_7.

[10] M. Hughes & D. Stotts (1996): *Daistish: systematic algebraic testing for OO programs in the presence of side-effects*. In: *Proc. ISSTA96, ACM Press*, pp. 53–61, doi:10.1145/229000.226301.

[11] D. Jackson (2012): *Software Abstractions - Logic, Language, and Analysis, Revised Edition*. MIT Press.

[12] L. Kong, H. Zhu & B. Zhou (2007): *Automated Testing EJB Components Based on Algebraic Specifications*. In: *COMPSAC 2007*, pp. 717–722, doi:10.1109/COMPSAC.2007.82.

[13] P.D.L. Machado & D. Sanella (2002): *Unit testing for CASL architectural specifications*. In: *Proc. 27th MFCS*, *LNCS* 2420, Springer, pp. 506–518, doi:10.1007/3-540-45687-2_42.

[14] I. Nunes, A. Lopes & V. Vasconcelos (2009): *Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming*. In: *Runtime Verification*, *LNCS* 5779, Springer, pp. 115–131, doi:10.1007/978-3-642-04694-0_9.

[15] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu & L. Reis (2006): *Checking the conformance of Java classes against algebraic specifications*. In: *Proc. 8th ICFEM*, *LNCS* 4260, Springer, pp. 494–513, doi:10.1007/11901433_27.

[16] I. Nunes & F. Luís (2012): *A fault-location technique for Java implementations of algebraic specifications*. Technical Report 02, Faculty of Sciences of the University of Lisbon.

[17] A. Riboira & R. Abreu (2010): *The GZoltar Project: A Graphical Debugger Interface*. In L. Bottaci & G. Fraser, editors: *TAIC PART*, *LNCS* 6303, Springer, pp. 215–218. Available at `http://dx.doi.org/10.1007/978-3-642-15585-7_25`.

[18] F. Steimann & M. Bertschler (2009): *A simple coverage-based locator for multiple faults*. In: *IEEE ICST*, *LNCS* 4536, Springer, pp. 366–375, doi:10.1109/ICST.2009.24.

[19] H. Zhu (2003): *A note on test oracles and semantics of algebraic specifications*. In: *QSIC 2003*, IEEE Computer Society, pp. 91–98, doi:10.1109/QSIC.2003.1319090.

Decomposability in Input Output Conformance Testing

Neda Noroozi
Eindhoven University of Technology
Eindhoven, The Netherlands
n.noroozi@tue.nl

Mohammad Reza Mousavi
Eindhoven University of Technology
Eindhoven, The Netherlands
Center for Research on Embedded Systems (CERES)
Halmstad University, Sweden
m.r.mousavi@tue.nl

Tim A.C. Willemse
Eindhoven University of Technology
Eindhoven, The Netherlands
t.a.c.willemse@tue.nl

We study the problem of deriving a specification for a third-party component, based on the specification of the system and the environment in which the component is supposed to reside. Particularly, we are interested in using component specifications for conformance testing of black-box components, using the theory of input-output conformance (ioco) testing. We propose and prove sufficient criteria for decompositionality, i.e., that components conforming to the derived specification will always compose to produce a correct system with respect to the system specification. We also study the criteria for strong decomposability, by which we can ensure that only those components conforming to the derived specification can lead to a correct system.

1 Introduction

Enabling reuse and managing complexity are among the major benefits of using compositional approaches in software and systems engineering. This idea has been extensively adopted in several different subareas of software engineering, such as product-line software engineering. One of the cornerstones of the product-line approach is to reuse a common platform to build different products. This common platform should ideally comprise different types of artifacts, including test-cases, that can be re-used for various products of a given line. In this paper, we propose an approach to conformance testing, which allows to use a high-level specification and derive specifications for to-be-developed components (or subsystems) given the platform on which they are to be deployed. We call this approach *decompositional* testing and refer to the process of deriving specifications as *quotienting* (inspired by its counterpart in the domain of formal verification).

We develop our approach within the context of input-output conformance testing (**ioco**) [13], a model-based testing theory using formal models based on input-output labeled transition systems (IOLTSs). An implementation i is said to conform to a specification s, denoted by i **ioco** s, when after each trace in the specification, the outputs of the implementation are among those prescribed by the specifications.

For a given platform (environment) $\bar{e}$, whose behavior is given as an IOLTS, a quotient of a specification $\bar{s}$ by the platform $\bar{e}$, denoted by $\bar{s}_{/\bar{e}}$, is the specification that describes the system after filtering out the effect of $\bar{e}$. The structure of a system consisting of $\bar{e}$ and unknown component $\bar{c}$ is represented in Figure 1, whose behavior is described by a given specification $\bar{s}$. We would like to construct $\bar{s}_{/\bar{e}}$ such that it captures the behavior of any component $\bar{c}$ which, when deployed on $\bar{e}$ (put in parallel and possibly synchronize with $\bar{e}$) conforms to $\bar{s}$. Put formally, $\bar{s}_{/\bar{e}}$ is the specification which satisfies the following bi-implication:

A. Petrenko, H. Schlingloff (Eds.): Eighth Workshop on
Model-Based Testing (MBT 2013)
EPTCS 111, 2013, pp. 51–66, doi:10.4204/EPTCS.111.5

$$\forall \bar{c}, \bar{e}.\ \ \bar{c} \textbf{ ioco } \bar{s}_{/\bar{e}} \quad \Leftrightarrow \quad \bar{c} \| \bar{e} \textbf{ ioco } \bar{s}$$

The criteria for the implication from left to right, which is essential for our approach, are called *decomposability*. The criteria for the implication from right to left guarantee that quotienting produces the precise specification for the component and is called *strong decomposability*. We study both criteria in the remainder of this paper. Moreover, we show that strong decomposability can be combined with *on-the-fly* testing, thereby avoiding constructing the witness to the decomposability explicitly upfront.

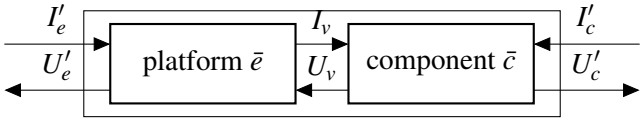


Figure 1: Strucure of a system composed of platform $\bar{e}$ and component $\bar{c}$ whose behavior is defined by a given specification $\bar{s}$. The language of platform $\bar{e}$ comprises $(I'_e \cup U_v) \cup (U'_e \cup I_v)$. Similarly, $(I'_c \cup I_v) \cup (U'_c \cup U_v)$ is the language of component $\bar{c}$. The platform $\bar{e}$ and component $\bar{c}$ interface via I_v and U_v which are hidden from the viewpoint of an external observer.

Related Work. The study of compositional and modular verification for various temporal and modal logics has attracted considerable attention and several compositional verification techniques have been proposed for such logics; see, e.g., [2, 7, 10, 6]. Decompositional reasoning aims at automatically decomposing the global property to be model checked into local properties of (possibly unknown) components, a technique that is often called quotienting. The notion of quotient introduced in the present paper is inspired by its corresponding notion in the area of (de)compositional model-checking, and is substantially adapted to the setting for input-output conformance testing, e.g., by catering for the distinction between input and output actions and taking care of (relative) quiescence of components. In the area of model-based testing, we are aware of a few studies dedicated to the issue of (de)composition [3, 5, 14], of which we give an overview below.

In [3] the compositionality of the **ioco**-based testing theory is investigated. Assuming that implementations of components conform to their specifications, the authors investigate whether the composition of these implementations still conforms to the composition of the specifications. They show that this is not necessarily the case and they establish conditions under which **ioco** is a compositional testing relation.

In [5], Frantzen and Tretmans study when successful integration of components by composing them in certain ways can be achieved. Successful integration is determined by two conditions: the integrated system correctly provides services, and interaction with other components is proper. For the former, a specification of the provided services of the component is assumed. Based on the **ioco**-relation, the authors introduce a new implementation relation called **eco**, which allows for checking whether a component conforms to its specification as well as whether it uses other components correctly. In addition, they also propose a bottom-up strategy for building an integrated systems.

Another problem closely related to the problem we consider in this paper is *testing in context*, also known as *embedded testing* [14]. In this setting, the system under test comprises a component $\bar{c}$ which is embedded in a context $\bar{u}$. Component $\bar{c}$ is isolated from the environment and all its interactions proceed through $\bar{u}$ (which is assumed to be correctly implemented). The implementation $\bar{i}$ and specification $\bar{s}$ of the system composed of $\bar{u}$ and $\bar{c}$, are assumed to be available. The problem of testing in context then entails generating a test suite that allows for detecting incorrect implementations $\bar{i}$ of component $\bar{c}$.

Although testing in context and decomposability share many characteristics, there are key differences between the two. We do not restrict ourselves to embedded components, nor do we assume the platforms to be fault-free. Contrary to the testing in context approach, decomposing a monolithic specification is the primary challenge in our work; testing in context already assumes the specification is the result of a composition of two specifications. Moreover, in testing in context, the component $\bar{c}$ is tested through context $\bar{u}$ whereas our approach allows for testing the component directly through its deduced specification. As a result, we do not require that the context is always available while testing the component, which is particularly important in case the platform is a costly resource.

For similar reasons, asynchronous testing [11, 8, 15], which can be considered as some form of embedded testing, is different from the work we present in this paper.

Structure. We give a cursory overview of **ioco**-based formal testing in Section 2. The notions of decomposability and strong decomposability are formalized in Section 3. We present sufficient conditions for determining whether a given specification is decomposable in Section 4 and whether it is strongly decomposable in Section 5. We conclude in Section 6. ***Additional examples and results, together with all proofs for the lemmata and theorems can be found in [9].***

2 Preliminaries

Conformance testing is about checking that the observable behavior of the system under test is included in the prescribed behavior of the specification. In order to formally reason about conformance testing, we need a model for reasoning about the behaviors described by a specification, and assume that we have a formal model representing the behaviors of our implementations, so that we can reason about their conformance mathematically.

In this paper, we use variants of the well-known Labeled Transition Systems as a behavioral model for both the specification and the system under test. The Labeled Transition System model assumes that systems can be represented using a set of states and transitions, labeled with events or *actions*, between such states. A tester can observe the events leading to new states, but she cannot observe the states. We assume the presence of a special action τ, which we assume is unobservable to the tester.

Definition 1 (IOLTS) *An input-output labeled transition system (IOLTS) is a tuple $\langle S, I, U, \rightarrow, \bar{s}\rangle$, where S is a set of states, I and U are disjoint sets of observable* inputs *and* outputs*, respectively, $\rightarrow \subseteq S \times (I \cup U \cup \{\tau\}) \times S$ is the transition relation (we assume $\tau \notin I \cup U$), and $\bar{s} \in S$ is the initial state. The class of IOLTSs ranging over inputs I and outputs U is denoted* $\mathsf{IOLTS}(I,U)$.

Throughout this section, we assume an arbitrary, fixed IOLTS $\langle S, I, U, \rightarrow, \bar{s}\rangle$, and we refer to this IOLTS by referring to its initial state $\bar{s}$. We write L for the set $I \cup U$. Let $s, s' \in S$ and $x \in L \cup \{\tau\}$. In line with common practice, we write $s \xrightarrow{x} s'$ rather than $(s,x,s') \in \rightarrow$. Furthermore, we write $s \xrightarrow{x}$ whenever $s \xrightarrow{x} s'$ for some $s' \in S$, and $s \not\xrightarrow{x}$ when not $s \xrightarrow{x}$. A *word* is a sequence over the input and output symbols. The set of all words over L is denoted L^*, and ε is the empty word. For words $\sigma, \rho \in L^*$, we denote the concatenation of σ and ρ by $\sigma\rho$. The transition relation is generalized to a relation over words by the following deduction rules:

$$\frac{}{s \overset{\varepsilon}{\Longrightarrow} s} \qquad \frac{s \overset{\sigma}{\Longrightarrow} s'' \qquad s'' \xrightarrow{x} s' \qquad x \neq \tau}{s \overset{\sigma x}{\Longrightarrow} s'} \qquad \frac{s \overset{\sigma}{\Longrightarrow} s'' \qquad s'' \xrightarrow{\tau} s'}{s \overset{\sigma}{\Longrightarrow} s'}$$

We adopt the notational conventions we introduced for $\rightarrow$ for $\Longrightarrow$. A state in the IOLTS $\bar{s}$ is said to *diverge* if it is the source of an infinite sequence of τ-labeled transitions. The IOLTS $\bar{s}$ is *divergent* if one of its reachable states diverges. Throughout this paper, we confine ourselves to non-divergent IOLTSs.

Definition 2 *Let $s' \in S$ and $S' \subseteq S$. The set of* traces, enabled actions *and* weakly enabled actions *for s and S' are defined as follows:*

- $\mathsf{traces}(s) = \{\sigma \in L^* \mid s \stackrel{\sigma}{\Rightarrow}\}$, *and* $\mathsf{traces}(S') = \bigcup_{s' \in S'} \mathsf{traces}(s')$.
- $\mathsf{init}(s) = \{x \in L \cup \{\tau\} \mid s \stackrel{x}{\rightarrow}\}$, *and* $\mathsf{init}(S') = \bigcup_{s' \in S'} \mathsf{init}(s')$.
- $\mathsf{Sinit}(s) = \{x \in L \mid s \stackrel{x}{\Rightarrow}\}$, *and* $\mathsf{Sinit}(S') = \bigcup_{s' \in S'} \mathsf{Sinit}(s')$.

Quiescence and Suspension Traces. Testers often not only have the power to observe events produced by an implementation, they can also observe the *absence* of events, or *quiescence* [13]. A state $s \in S$ is said to be *quiescent* if it does not produce outputs and it is *stable*. That is, it cannot, through internal computations, evolve to a state that *is* capable of producing outputs. Formally, state s is quiescent, denoted $\delta(s)$, whenever $\mathsf{init}(s) \subseteq I$. In order to formally reason about the observations of inputs, outputs and quiescence, we introduce the set of *suspension traces*. To this end, we first generalize the transition relation over words to a transition relation over suspension words. Let L_δ denote the set $L \cup \{\delta\}$.

$$\frac{s \stackrel{\sigma}{\Longrightarrow} s'}{s \stackrel{\sigma}{\Longrightarrow}_\delta s'} \qquad \frac{\delta(s)}{s \stackrel{\delta}{\Rightarrow}_\delta s} \qquad \frac{s \stackrel{\sigma}{\Longrightarrow}_\delta s'' \qquad s'' \stackrel{\rho}{\Longrightarrow}_\delta s'}{s \stackrel{\sigma\rho}{\Longrightarrow}_\delta s'}$$

The following definition formalizes the set of suspension traces.

Definition 3 *Let $s \in S$ and $S' \subseteq S$. The set of* suspension traces *for s, denoted $\mathsf{Straces}(s)$ is defined as the set $\{\sigma \in L_\delta^* \mid s \stackrel{\sigma}{\Rightarrow}_\delta\}$; we set $\mathsf{Straces}(S') = \bigcup_{s' \in S'} \mathsf{Straces}(s')$.*

Input-Output Conformance Testing with Quiescence. Tretmans' **ioco** testing theory [13] formalizes black box conformance of implementations. It assumes that the behavior of implementations can always be described adequately using a class of IOLTSs, called *input output transition systems*; this assumption is the so-called *testing hypothesis*. Input output transition systems are essentially plain IOLTSs with the additional assumption that inputs can always be accepted.

Definition 4 (IOTS) *Let $\langle S, I, U, \rightarrow, \bar{s}\rangle$ be an IOLTS. A state $s \in S$ is* input-enabled *iff $I \subseteq \mathsf{Sinit}(s)$; the IOLTS $\bar{s}$ is an* input output transition system *(IOTS) iff every state $s \in S$ is input-enabled. The class of input output transition systems ranging over inputs I and outputs U is denoted $\mathsf{IOTS}(I,U)$.*

While the **ioco** testing theory assumes input-enabled implementations, it does not impose this requirement on specifications. This facilitates testing using partial specifications, *i.e.*, specifications that are under-specified. We first introduce the main concepts that are used to define the family of conformance relations of the **ioco** testing theory.

Definition 5 *Let $\langle S, I, U, \rightarrow, \bar{s}\rangle$ be an IOLTS. Let $s \in S$, $S' \subseteq S$ and let $\sigma \in L_\delta^*$.*

- s after $\sigma = \{s' \in S \mid s \stackrel{\sigma}{\Rightarrow}_\delta s'\}$, *and* S' after $\sigma = \bigcup_{s' \in S'} s'$ after σ.
- $\mathsf{out}(s) = \{x \in L_\delta \setminus I \mid s \stackrel{x}{\Rightarrow}_\delta\}$, *and* $\mathsf{out}(S') = \bigcup_{s' \in S'} \mathsf{out}(s')$.

The family of conformance relations for **ioco** are then defined as follows, see also [13].

Definition 6 (ioco) *Let $\langle R, I, U, \rightarrow, \bar{r}\rangle$ be an IOTS representing a realization of a system, and let IOLTS $\langle S, I, U, \rightarrow, \bar{s}\rangle$ be a specification. Let $F \subseteq L_\delta^*$. We say that $\bar{r}$ is* input output conform *with specification $\bar{s}$, denoted $\bar{r}$ **ioco**$_F$ $\bar{s}$, iff*

$$\forall \sigma \in F : \ \mathsf{out}(\bar{r} \text{ after } \sigma) \subseteq \mathsf{out}(\bar{s} \text{ after } \sigma)$$

The **ioco**$_F$ conformance relation can be specialized by choosing an appropriate set F. For instance, in a setting with $F = \mathsf{Straces}(s)$, we obtain the **ioco** relation originally defined by Tretmans in [12]. The latter conformance relation is known to admit a sound and complete test case generation algorithm, see, *e.g.*, [12, 13]. Soundness means, intuitively, that the algorithm will never generate a test case that, when executed on an implementation, leads to a *fail* verdict if the test runs are in accordance with the specification. Completeness is more esoteric: if the implementation has a behavior that is not in line with the specification, then there is a test case that, in theory, has the capacity to detect that non-conformance.

Suspension automata. The original test case generation algorithm by Tretmans for the **ioco** relation relied on an automaton derived from an IOLTS specification. This automaton, called a *suspension automaton*, shares many of the characteristics of an IOLTS, except that the observations of quiescence are encoded explicitly as outputs: δ is treated as an ordinary action label which can appear on a transition. In addition, Tretmans assumes these suspension automata to be *deterministic*: any word that could be produced by an automaton leads to exactly one state in the automaton.

Definition 7 (Suspension automaton) *A suspension automaton(SA) is a deterministic IOLTS $\langle S, I, U \cup \{\delta\}, \rightarrow, \bar{s}\rangle$; that is, for all $s \in S$ and all $\sigma \in L^*$, we have $|s \text{ after } \sigma| \leq 1$.*

Note that determinism implies the absence of τ transitions. In [12], a transformation from ordinary IOLTSs to suspension automata is presented; the transformation ensures that trace-based testing using the resulting suspension automaton is exactly as powerful as **ioco**-based testing using the original IOLTS.

The transformation is essentially based on the subset construction for determinizing automata. Given an IOLTS, the transformation Δ defined below converts any IOLTS into an SA.

Definition 8 *Let $\langle S, I, U, \rightarrow, \bar{s}\rangle \in \mathsf{IOLTS}(I, U)$. The SA $\Delta(\bar{s}) = \langle Q, I, U \cup \{\delta\}, \rightarrow, \bar{q}\rangle$ is defined as:*

- $Q = \mathbb{P}(S) \setminus \{\emptyset\}$.
- $\bar{q} = \bar{s}$ after ε.
- $\rightarrow \subseteq Q \times L_\delta \times Q$ *is the least relation satisfying:*

$$\frac{x \in L \qquad q \in Q}{q \stackrel{x}{\rightarrow} \{s' \in S \mid \exists s \in q \bullet s \stackrel{x}{\Rightarrow} s'\}} \qquad \frac{q \in Q}{q \stackrel{\delta}{\rightarrow} \{s \in q \mid \delta(s)\}}$$

Example 1 *Consider the IOLTS $\bar{s}$ depicted in Figure 2 on page 57. The IOLTS $\bar{s}$ is a specification of a malfunctioning vending machine which sells tea for one euro coin (c). After receiving money, it either delivers tea (t), refunds the money (r) or does nothing. Its suspension automaton $\Delta(\bar{s})$, with initial state $\bar{q}$, is depicted next to it. Note that the suspension traces of $\bar{s}$ and the traces of suspension automaton $\Delta(\bar{s})$ are identical.*

In general, a suspension automaton may not represent an actual IOLTS; for instance, in an arbitrary suspension automaton, it is allowed to observe quiescence, followed by a proper output. This cannot happen in an IOLTS. In [16], the set of suspension automata is characterized for which a transformation to an IOLTS *is* possible. Such suspension automata are called *valid*. Proposition 1 of [16] states that for any IOLTS $\bar{s}$, the suspension automaton $\Delta(\bar{s})$ is valid. Conversely, Theorem 2 of [16] states that any valid suspension automaton has the same testing power (with respect to **ioco**) as *some* IOLTS. This essentially means that the class of valid suspension automata can be used safely for testing purposes.

Parallel Composition. A software or hardware system is usually composed of subunits and modules that work in an orchestrated fashion to achieve the desired overall behavior of the software or hardware system. In our setting, we can formalize such compositions using a special operator $\_\|\_$ on IOLTSs: two IOLTSs can interact by connecting the outputs sent by one IOLTS to the inputs of the other IOLTS. We assume that such inputs and outputs are taken from a shared alphabet of actions. For the non-common actions the behavior of both IOLTSs is interleaved.

Definition 9 (parallel composition) *Let $\langle S_1, I_1, U_1, \rightarrow_1, \bar{s}_1\rangle$ and $\langle S_2, I_2, U_2, \rightarrow_2, \bar{s}_2\rangle$ be two IOLTSs with disjoint sets of input labels I_1 and I_2, and disjoint sets of output labels U_1 and U_2. The parallel composition of $\bar{s}_1$ and $\bar{s}_2$, denoted $\bar{s}_1 \| \bar{s}_2$ is the IOLTS $\langle Q, I, U, \rightarrow, \bar{s}_1 \| \bar{s}_2\rangle$, where:*

- $Q = \{s_1 \| s_2 \mid s_1 \in S_1, s_2 \in S_2\}$.
- $I = (I_1 \cup I_2) \setminus (U_1 \cup U_2)$ *and* $U = U_1 \cup U_2$.
- $\rightarrow \subseteq Q \times (L \cup \{\tau\}) \times Q$ *is the least relation satisfying:*

$$\frac{s_1 \xrightarrow{x}_1 s_1' \qquad x \notin L_2}{s_1 \| s_2 \xrightarrow{x} s_1' \| s_2} \qquad \frac{s_2 \xrightarrow{x}_2 s_2' \qquad x \notin L_1}{s_1 \| s_2 \xrightarrow{x} s_1 \| s_2'}$$

$$\frac{s_1 \xrightarrow{x}_1 s_1' \qquad s_2 \xrightarrow{x}_2 s_2' \qquad x \neq \tau}{s_1 \| s_2 \xrightarrow{x} s_1' \| s_2'}$$

The interaction *between* components is typically intended to be unobservable by a tester. This is not enforced by the parallel composition, but can be specified by combining parallel composition with a *hiding* operator, which is formalized below.

Definition 10 (hiding) *Let $\langle S, I, U, \rightarrow, \bar{s}\rangle$ be an IOLTS, and let $V \subseteq U$. The IOLTS resulting from hiding events from the set V, denoted by* $\mathbf{hide}[V]\,\mathbf{in}\, s$ *is the IOLTS $\langle S, I, U \setminus V, \rightarrow', \bar{s}\rangle$, where $\rightarrow'$ is defined as the least relation satisfying:*

$$\frac{s \xrightarrow{x} s' \qquad x \notin V}{\mathbf{hide}[V]\,\mathbf{in}\, s \xrightarrow{x}{}' \mathbf{hide}[V]\,\mathbf{in}\, s'} \qquad \frac{s \xrightarrow{x} s' \qquad x \in V}{\mathbf{hide}[V]\,\mathbf{in}\, s \xrightarrow{\tau}{}' \mathbf{hide}[V]\,\mathbf{in}\, s'}$$

Note that the hiding operator may turn non-divergent IOLTSs into divergent IOLTSs. As divergence is excluded from the **ioco** testing theory, we must assume such divergences are not induced by composing two implementations in parallel and hiding all successful communications. Since implementations are assumed to be input enabled, this can only be ensured whenever components that are put in parallel never produce infinite, uninterrupted runs of outputs over their alphabet of shared output actions. Implementations adhering to these constraints are referred to as *shared output bounded* implementations. From hereon, we assume that all the implementions considered are shared output bounded.

3 Decomposibility

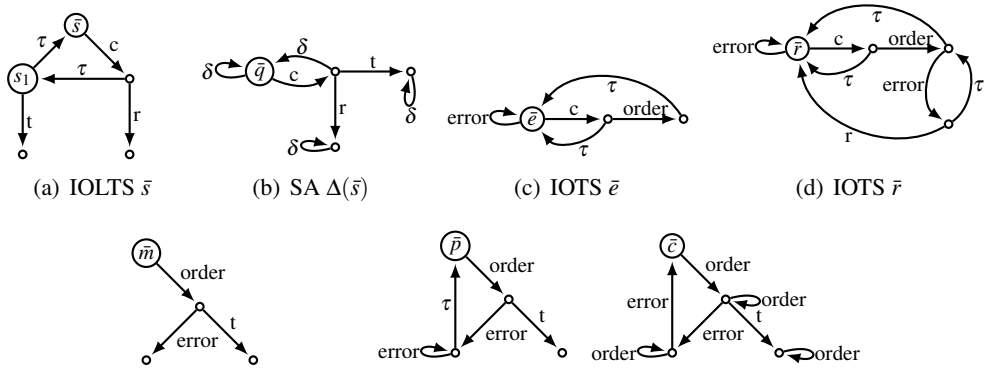
Software can be constructed by decomposing a specification of the software in specifications of smaller complexity. Reuse of readily available and well-understood platforms or environments can steer such a decomposition. Given the prevalence of such platforms, the software engineering and associated testing problem thus shifts to finding a proper specification of the system from which the platform behavior has been factored out. Whether this is possible, however, depends on the specification; if so, we say that a specification is *decomposable*.

The decomposability problem requires known action alphabets for both the specification and the platform. Hence, we first fix these alphabets and illustrate how these are related. Hereafter, L_s denotes the action alphabet of the specification $\bar{s}$ and L_e denotes the action alphabet of the platform $\bar{e}$. The actions of L_e not exposed to $\bar{s}$ are contained in action alphabet L_v, *i.e.*, we have $L_v = L_e \setminus L_s$. The action alphabet of the quotient will be denoted by L, *i.e.* $L = (L_s \setminus L_e) \cup L_v$. The relation between the above alphabets is illustrated in Figure 1 in the introduction.

Definition 11 (Decomposability) *Let $\bar{s} \in \mathsf{IOLTS}(I_s, U_s)$ be a specification, and let $\bar{e} \in \mathsf{IOTS}(I_e, U_e)$ be an implementation. Let $L_v = I_v \cup U_v$ be a set of actions of $\bar{e}$ not part of $\bar{s}$. Specification $\bar{s}$ is said to be* decomposable *for IOTS $\bar{e}$ iff there is some specification $\bar{s}' \in \mathsf{IOLTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v)$ for which both:*

- $\exists \bar{c} \in \mathsf{IOTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v) \bullet \bar{c}$ ***ioco*** $\bar{s}'$*, and*
- $\forall \bar{c} \in \mathsf{IOTS}((I_e \setminus I_e) \cup I_v, (U_e \setminus U_e) \cup U_v) \bullet \bar{c}$ ***ioco*** $\bar{s}' \Longrightarrow$ **hide**$[L_v]$ **in** $\bar{c} \| \bar{e}$ ***ioco*** $\bar{s}$

Decomposability of a specification $\bar{s}$ essentially ensures that a specification $\bar{s}'$ for a subcomponent exists that guarantees that every **ioco**-correct implementation of it is also guaranteed to work correctly in combination with the platform.



(a) IOLTS $\bar{s}$ (b) SA $\Delta(\bar{s})$ (c) IOTS $\bar{e}$ (d) IOTS $\bar{r}$

(e) IOLTS $\bar{m}$ (f) IOLTS $\bar{p}$ (g) IOTS $\bar{c}$

Figure 2: A specification of a vending machine ($\bar{s}$), two behavioral models of an implemented money component ($\bar{e}$ and $\bar{r}$) and two specifications for a drink component ($\bar{m}$ and $\bar{p}$) with the behavioral model of an implementation of the drink component ($\bar{c}$).

Example 2 *Consider IOLTSs depicted in Figure 2. The IOTS $\bar{e}$ 2(c) presents the behavioral model of an environment which after receiving a coin (c) either orders drink (order) or does nothing. Upon receiving an error signal (error), never refunds the money (r). Component $\bar{e}$ interacts with another component*

through actions 'order' and 'error'; together, the components implement a vending machine for which IOLTS $\bar{s}$ 2(a) is the specification. The IOLTS $\bar{m}$ 2(e) is a specification of a drink component which delivers tea after receiving a drink order. If it encounters a problem in delivering the drink, it signals an error. Specification $\bar{m}$ guarantees that the combination of component $\bar{e}$ with any drink component implementation conforming to $\bar{m}$, also conforms to $\bar{s}$.

It may, however, be the case that an implementation, in combination with a given platform, perfectly adheres to the overall specification $\bar{s}$, and, yet fails to pass the conformance test for $\bar{s}'$. As a consequence, non-conformance of an implementation to $\bar{s}'$ may not by itself be a reason to reject the implementation.

Example 3 *Consider IOLTSs in Figure 2. The IOLTS $\bar{m}$ 2(e) is a witness for decomposability of IOLTS $\bar{s}$2(a) for platform $\bar{e}$2(c). Thus, any compound system built of IOTS $\bar{e}$ and a component conforming to $\bar{m}$ is guaranteed to be in conformance with IOLTS $\bar{s}$. Now, consider IOTS $\bar{c}$ 2(g) which incorrectly implements the functionality specified in IOLTS $\bar{m}$ 2(e), as it sends 'error' twice. Observe that, nevertheless,* $\mathbf{hide}[\{error, order\}]\,\mathbf{in}\,\bar{c}\|\bar{e}$ *still conforms to $\bar{s}$.*

It is often desirable to consider specifications $\bar{s}'$ for which one *only* has to check whether an implementation $\bar{c}$ adheres to $\bar{s}'$, *i.e.*, specifications for which it is guaranteed that a failure of an implementation $\bar{c}$ to comply to $\bar{s}'$ also guarantees that the combination $\bar{c}\|\bar{e}$ will violate the original specification $\bar{s}$. We can obtain this by considering a stronger notion of decomposability.

Definition 12 (Strong Decomposability) *Let $\bar{s} \in \mathsf{IOLTS}(I,U)$ be a specification, and let $\bar{e} \in \mathsf{IOTS}(I_e,U_e)$ be an implementation. Let $L_v = I_v \cup U_v$ be a set of actions of $\bar{e}$ not part of $\bar{s}$. Specification $\bar{s}$ is said to be* strongly decomposable *for IOTS $\bar{e}$ iff there is some specification $\bar{s}' \in \mathsf{IOLTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v)$ for which both:*

- $\exists \bar{c} \in \mathsf{IOTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v)\ \bullet\ \bar{c}\ \boldsymbol{ioco}\ \bar{s}'$, *and*
- $\forall \bar{c} \in \mathsf{IOTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v)\ \bullet\ \bar{c}\ \boldsymbol{ioco}\ \bar{s}' \iff \mathbf{hide}[L_v]\,\mathbf{in}\,\bar{c}\|\bar{e}\ \boldsymbol{ioco}\ \bar{s}$

Example 4 *Consider the IOLTSs $\bar{p}$ and $\bar{e}$ in Figure 2; specification $\bar{p}$ is such that the combination of component $\bar{e}$ with any* shared output bounded *component that does not conform to $\bar{p}$, fails to comply to $\bar{s}$.*

4 Sufficient Conditions for Decomposibility

Checking whether a given specification is decomposable is a difficult problem. However, knowing that a specification is decomposable in itself hardly helps a design engineer. Apart from the question whether a specification is decomposable, one is typically interested in a witness for the decomposed specification, or *quotient*. Our approach to the decomposability problem is therefore constructive: we define a quotient and we identify several conditions that ensure that the quotient we define is a witness for the decomposability of a given specification.

One of the problems that may prevent a specification from being decomposable for a given platform $\bar{e}$ is that the latter may exhibit some behavior which unavoidably violates the specification $\bar{s}$. We shall therefore only consider platforms for which such violations are not present. We formalize this by checking whether the behavior of $\bar{e}$ is *included* in the behavior of $\bar{s}$; that is, we give conditions that ensure that $\bar{e}$ in itself cannot violate the given specification $\bar{s}$. Moreover, we assume that the input-enabled specification of $\bar{e}$ is available.

Assuming that the behavior of $\bar{e}$ is included in the behavior of the given specification $\bar{s}$, we then propose a quotient $\bar{s}'$ of $\bar{s}$ for $\bar{e}$ and prove sufficient conditions that guarantee that $\bar{s}$ is indeed decomposable and $\bar{s}'$ is a witness to that.

4.1 Inclusion relation

We say that the behavior of a given platform $\bar{e}$ is included in a specification $\bar{s}$ if the outputs allowed by $\bar{s}$ subsume all outputs that can be produced by $\bar{e}$. For this, we need to take possible communications between $\bar{e}$ and the to-be-derived quotient over the action alphabet L_v into account. Another issue is that we are dealing with two components, each of which may be quiescent. If component $\bar{e}$ is quiescent, its quiescence may be masked by outputs from the component with which it is supposed to interact. We must therefore consider a refined notion of quiescence. We say state s in specification $\bar{s}$ is *relatively quiescent* with respect to alphabet L_e, denoted by $\delta_{\bar{e}}(s)$, if s produces no output of L_e, *i.e.* $\mathsf{out}(s) \cap L_e = \emptyset$. Analogous to δ, the suspension traces of $\bar{s}$ can be enriched by adding the rule $s \stackrel{\delta_{\bar{e}}}{\Rightarrow}_{\delta} s$ for $\delta_{\bar{e}}(s)$ to be able to formally reason about the possibility of being relatively quiescent with respect to L_e. We write $\mathsf{Straces}_{\bar{e}}(\bar{s})$ to denote this enriched set of suspension traces of $\bar{s}$.

Since the suspension traces of $\bar{s}$ and $\bar{e}$ differ as a result of different alphabets, we introduce a *projection operator* which allows us to map the suspension traces of $\bar{s}$ to suspension traces of $\bar{e}$. The operator $_{-\downarrow_{L_e}}$ is defined as $(x\sigma)_{\downarrow_{L_e}} = x\sigma_{\downarrow_{L_e}}$ if $x \in L_e$; $(x\sigma)_{\downarrow_{L_e}} = \delta(\sigma_{\downarrow_{L_e}})$, if $x \in \{\delta, \delta_{\bar{e}}\}$; otherwise, $(x\sigma)_{\downarrow_{L_e}} = \sigma_{\downarrow_{L_e}}$.

Definition 13 *Let IOTS $\langle S_e, I_e, U_e, \rightarrow, \bar{e}\rangle$ be an implementation. Let IOLTS $\langle S_s, I_s, U_s, \rightarrow, \bar{s}\rangle$ be a specification. We say the behavior of $\bar{e}$ is included in $\bar{s}$, denoted by $\bar{e}$* **incl** *$\bar{s}$ iff*

$$\forall \sigma \in \mathsf{Straces}_{\bar{e}}(\bar{s}) : \mathsf{out}(\mathbf{hide}[L_v]\,\mathbf{in}\,\bar{e} \text{ after } \sigma_{\downarrow_{L_e}}) \subseteq \mathsf{out}(\bar{s} \text{ after } \sigma)$$

Example 5 *Consider the IOLTSs in Figure 2. We have $\bar{e}$* **incl** *$\bar{s}$. Consider the IOLTS $\bar{r}$ which has the same functionality with IOLTS $\bar{e}$ except that upon receiving an error signal (error), it may or may not refund the money (r). The behavior of $\bar{r}$ is not included in $\bar{s}$, because of observing the output r in $\bar{r}$ after executing $(ct)_{\downarrow_{L_e}}$ while $\bar{s}$ after execution of ct reaches to a quiescent state.*

4.2 Quotienting

We next focus on deriving a quotient of the specification $\bar{s}$, factoring out the behavior of the platform $\bar{e}$. A major source of complexity in defining such a quotient is the possible non-determinism that may be present in $\bar{s}$ and $\bar{e}$. We largely avert this complexity by utilizing the suspension automata underlying $\bar{s}$ and $\bar{e}$.

Another source of complexity is the fact that we must reason about the states of two systems running in parallel; such a system synchronizes on shared actions and interleaves on non-shared actions. We tame this conceptual complexity by formalizing an executes operator which, when executing a shared or non-shared action, keeps track of the set of reachable states for the (suspension automata) of $\bar{s}$ and $\bar{e}$. Formally, the executes operator is defined as follows.

Definition 14 *Let $\langle Q_s, I_s, U_s \cup \{\delta\}, \rightarrow_s, \bar{q}_s\rangle$ be a suspension automaton underlying specification IOLTS $\bar{s}$, and let $\langle Q_e, I_e, U \cup \{\delta\}, \rightarrow_e, \bar{q}_e\rangle$ be a suspension automaton underlying platform IOLTS $\bar{e}$. Let $q \in \mathbb{P}(Q_s \times Q_e)$ be a non-empty collection of sets and let $x \in L_s \setminus (L_e \setminus L_v)$.*

$$q \text{ executes } x = \begin{cases} \bigcup_{\sigma \in L_e^*} \bigcup_{(s,e) \in q} \{(q'_s, q'_e) \mid s \xrightarrow{\sigma}_s q'_s \text{ and } e \xrightarrow{\sigma x}_e q'_e\} & \text{if } x \in L_v \\ \bigcup_{\sigma \in L_e^*} \bigcup_{(s,e) \in q} \{(q'_s, q'_e) \mid s \xrightarrow{\sigma x}_s q'_s \text{ and } e \xrightarrow{\sigma}_e q'_e\} & \text{if } x \notin L_v \\ \bigcup_{\sigma \in L_e^*} \bigcup_{(s,e) \in q} \{(q'_s, q'_e) \mid s \xrightarrow{\sigma \delta}_s q'_s \text{ and } e \xrightarrow{\sigma \delta}_e q'_e\} & \text{if } x = \delta \end{cases}$$

Using the executes operator, we have an elegant construction of an automaton, called a *quotient automaton*, see below, which allows us to define sufficient conditions for establishing the decomposability of a given specification.

Definition 15 (Quotient Automaton) *Let $\langle Q_s, I_s, U_s \cup \{\delta\}, \rightarrow_s, \bar{q}_s \rangle$ be a suspension automaton underlying specification $\bar{s}$, and let $\langle Q_e, I_e, U_e \cup \{\delta\}, \rightarrow_e, \bar{q}_e \rangle$ be a suspension automaton underlying platform $\bar{e}$. The quotient of $\bar{s}$ by $\bar{e}$, denoted by $\bar{s}_{/\bar{e}}$ is a suspension automaton $\langle Q, I, U \cup \{\delta\}, \rightarrow, \bar{q} \rangle$ where:*

- $Q = (\mathbb{P}(Q_s \times Q_e) \setminus \{\emptyset\}) \cup Q_\delta$, *where* $Q_\delta = \{q_\delta \mid q \in \mathbb{P}(Q_s \times Q_e), q \neq \emptyset\}$; *for* $q \notin Q_\delta$, *we set* $q^{-1} = q$ *and for* $q_\delta \in Q_\delta$, *we set* $q_\delta^{-1} = q$.
- $\bar{q} = \{(\bar{q}_s, \bar{q}_e)\}$.
- $I = (I_s \setminus I_e) \cup (U_e \setminus U_s)$ *and* $U = (U_s \setminus U_e) \cup \{\delta\} \cup (I_e \setminus I_s)$.
- $\rightarrow \subseteq Q \times L \times Q$ *is the least set satisfying:*

$$\frac{a \in I \quad q^{-1} \text{ executes } a \neq \emptyset}{q \xrightarrow{a} q^{-1} \text{ executes } a} \; [I_1] \qquad \frac{x \in U_v \quad q \notin Q_\delta \quad q^{-1} \text{ executes } x \neq \emptyset}{q \xrightarrow{x} q^{-1} \text{ executes } x} \; [U_1]$$

$$\frac{x \in U \setminus U_v \quad \forall (s,e) \in q, \sigma \in \mathsf{traces}(s) \cap \mathsf{traces}(e) \cap (L_\delta^* \setminus L_\delta^* \delta) : \; x \in \mathsf{out}(s \text{ after } \sigma)}{q \xrightarrow{x} q^{-1} \text{ executes } x} \; [U_2]$$

$$\frac{\forall (s,e) \in q^{-1}, \sigma \in \mathsf{traces}(s) \cap \mathsf{traces}(e) : \; \delta \in \mathsf{out}(s \text{ after } \sigma)}{q \xrightarrow{\delta} q^{-1} \text{ executes } \delta} \; [\delta_1]$$

We briefly explain the construction of a quotient automaton. A *non-shared input action* is added to a state in the quotient automaton $\bar{s}_{/\bar{e}}$ if an execution of the corresponding state in $\bar{e}$ leads to a state in $\bar{s}$ at which that action is enabled (I_1, in combination with the second case in Definition 14). A *shared input action* obeys the same rule except that a state of $\bar{e}$ has to be reachable where that input action is taken (I_1, in combination with the first case in Definition 14). Note that a shared input action of $\bar{s}_{/\bar{e}}$ is an output action from the viewpoint of $\bar{e}$. In contrast, a *non-shared output action* is allowed at a state of $\bar{s}_{/\bar{e}}$ only if it is allowed by $\bar{s}$ after any possible execution of $\bar{e}$ (U_2) and a similar rule is applied to quiescence (δ_1). Analogous to the shared input actions, a *shared output action* is considered as an action of a state whenever a valid execution of the correspondent states in $\bar{e}$ leads to a state at which that output action is enabled (U_1). Because the shared actions are hidden in $\bar{s}$, a shared output action, in $\bar{s}_{/\bar{e}}$, may also be enabled at a state reached by δ transitions. Such a sequence of events is invalid due to the definition of quiescence. The observed problem is solved by adding a special set of states Q_δ to the states of the quotient automaton. These states represent quiescent states corresponding to the reachable states after executing δ in $\bar{s}_{/\bar{e}}$. Moreover, no shared output action is added to these states.

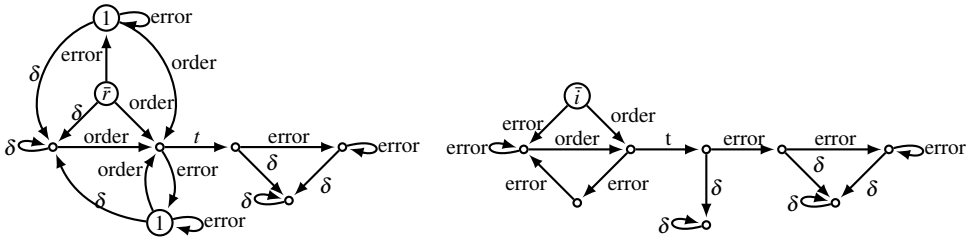


Figure 3: Two quotient automata derived using Definition 15

The quotient automaton derived from specification $\bar{s}$ and platform $\bar{e}$ is a suspension automaton: it is deterministic and it has explicit δ labels. Yet, the quotient automata we derive are not necessarily valid suspension automata. (As we recalled in Section 2, only *valid* suspension automata have the same testing power as ordinary IOLTSs.) We furthermore observe that there some quotient automata that are valid suspension automata but nevertheless only admit non-shared output bounded implementations as implementations that conform to the quotient. As observed earlier, such implementations unavoidably give rise to divergent systems when composed in parallel with the platform.

Example 6 *Consider SAs depicted in Figure 3, IOLTSs $\bar{s}$ and $\bar{e}$ in Figure 2 and IOLTS $\bar{l}$ derived by removing the internal transition from state s_1 to the initial state in $\bar{s}$. SA $\bar{r}$ is the quotient of $\bar{s}$ by $\bar{e}$. Likewise, SA $\bar{i}$ is the quotient of $\bar{l}$ by $\bar{e}$. Suspension automata $\bar{r}$ and $\bar{i}$ are valid SA regarding the definition of validity of suspension automata presented in [16] . Assume an arbitrary shared output bounded IOTS $\bar{c}$ whose length of the longest sequence on the shared output is n,* i.e. $\mathsf{out}(\bar{c}\ \mathsf{after}\ \sigma) \subseteq \{tea, \delta\}$ *for* $\sigma = \{error\}^n$. *Clearly, $\bar{c}$* ~~**ioco**~~ *$\bar{i}$, because* $\mathsf{out}(\bar{i}\ \mathsf{after}\ \sigma) = \{error\}$. *However, for any $n \geq 0$, there is always a shared output bounded IOTS that conforms to $\bar{r}$.*

In view of the above, we say that a quotient automaton is *valid* if it is a valid suspension automaton and strongly non-blocking.

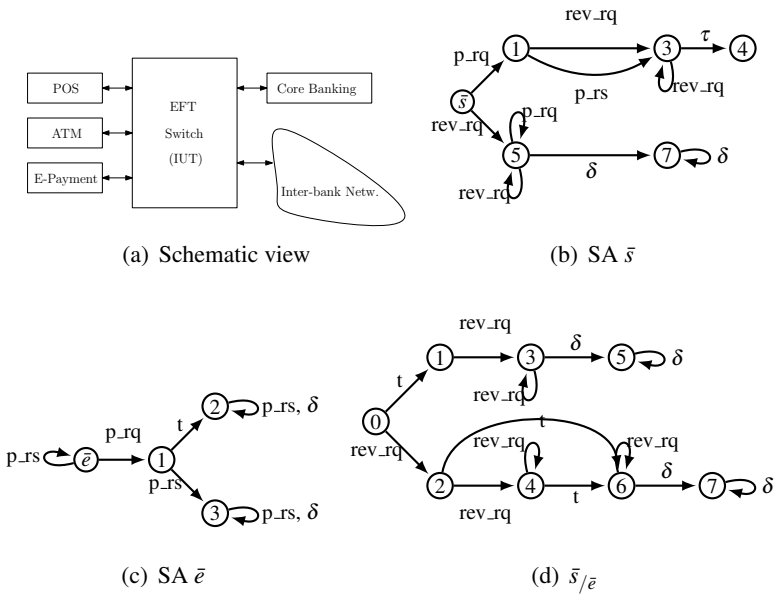
Definition 16 *Let $\bar{s}_{/\bar{e}}$ be a quotient automaton derived from a specification $\bar{s}$ and an environment $\bar{e}$. We say that $\bar{s}_{/\bar{e}}$ is* valid *iff both:*

- *$\bar{s}_{/\bar{e}}$ is a valid suspension automaton, and*
- *$\bar{s}_{/\bar{e}}$ is strongly non-blocking,* i.e. $\forall q \in \bar{s}_{/\bar{e}} \bullet \mathsf{out}(q) \cap ((U \setminus U_v) \cup \{\delta\}) \neq \emptyset$.

Strongly non-blocking ensures that the quotient automaton always admits a shared output bounded implementation that conforms to it. Furthermore, valid quotient automata are, by definition, also valid suspension automata. Since every valid suspension automaton underlies at least one IOLTS, we therefore have established a sufficient condition for the decomposability of a specification.

Theorem 1 *Let $\bar{s} \in \mathsf{IOLTS}(I_s, U_s)$ be a specification and let $\bar{e} \in \mathsf{IOTS}(I_e, U_e)$ be an environment. Then $\bar{s}$ is decomposable for $\bar{e}$ if $\bar{s}_{/\bar{e}}$ is a valid quotient automaton and $\bar{e}$* **incl** *$\bar{s}$.*

Note that the IOLTS underlying the quotient automaton is a witness to the decomposability of the specification; we thus not only have a sufficient condition for the decomposability of a specification but also a witness for the decomposition.



(a) Schematic view (b) SA $\bar{s}$

(c) SA $\bar{e}$ (d) $\bar{s}_{/\bar{e}}$

Figure 4: A schematic view of the EFT Switch, a suspension automata of simplified behavioral models of the EFT switch $\bar{s}$ and an implementation of the financial component $\bar{e}$, and the quotient of $\bar{s}$ w.r.t. $\bar{e}$

4.3 Example

To illustrate the notions introduced so far, we treat a simplified model of an Electronic Funds Transfer (EFT) switch, which we have studied and tested using ioco-based techniques [1]. A schematic view of this example is depicted in Figure 4(a). An EFT switch provides a communication mechanism among different components of a card-based financial system. On one side of the EFT switch, there are components, with which the end-user deals, such as Automated Teller Machines (ATMs), Point-of-Sale (POS) devices and e-Payment applications. On the other side, there are banking systems and the inter-bank network connecting the switches of different financial institutions.

The various involving parties in every transaction performed by an EFT switch in conjunction with the variety of financial transactions complicate the behavioral model of the EFT switch. Similar to any other complex software system, the EFT switch comprises many different components, some of which can be run individually.

A part of the simplified communication model of the EFT switch with a banking system in the purchase scenario is depicted in Figure 4(b). The scenario starts by receiving a purchase request from a POS; this initial part of the scenario is removed from the model, for the sake of brevity. Subsequently, the EFT switch sends a purchase request (p_rq) to the banking system. The EFT switch will reverse (rev_rq) the sent purchase request if the corresponding response (p_rs) is not received within a certain amount of time (e.g, an internal time-out occurs, denoted by τ). Due to possible delays in the network layer of the EFT switch, an external observer (tester) may observe the reverse request of a purchase even before the purchase request which is pictured in Fig 4(b).

The EFT switch is further implemented in terms of two components, namely, the financial component and the reversal component. A simplified behavioral model of the financial component is given in Figure 4(c). Comparing the two languages of $\bar{s}$ and $\bar{e}$, t action (representing time-out) is considered as an internal

interface between $\bar{e}$ and a to-be-developed implementation of the reversal component. Observe that for every sequence σ in $\{p\_rq(\delta_e|rev\_rq)^*, p\_rq(\delta_e|rev\_rq)^* rev\_rq(\delta|\delta_e)^*, p\_rq\ p\_rs(\delta_e|rev\_rq)^*(\delta|\delta_e)^*, (\delta_e|rev\_rq)^*, (\delta_e|rev\_rq)^* rev\_rq(rev\_rq|p\_rq)^*(\delta|\delta_e)^*\}$, it holds that $\mathsf{out}(\mathbf{hide}[t]\,\mathbf{in}\,\bar{e}\ \ \mathsf{after}\ \sigma_{\downarrow_{L_e}}) \subseteq \mathsf{out}(\bar{s}\ \mathsf{after}\ \sigma)$; thus, the behavior of $\bar{e}$ is included in $\bar{s}$. We next instigate investigate decomposability of $\bar{s}$ with $\bar{e}$, by constructing the quotient $\bar{s}_{/\bar{e}}$. Note that t is the only shared action which is an input action from the view point of $\bar{s}_{/\bar{e}}$. The resulting quotient automaton, obtained by applying Definition 15 to $\bar{s}$ and $\bar{e}$ is depicted in Figure 4(d). We illustrate some steps in its derivation. The initial state of the quotient automaton is defined as the $\{(\bar{s},\bar{e})\}$. Below, we illustrate which of the rules of Definition 15 are possible from this initial state; doing so repeatedly for all reached states will ultimately produce the reachable states of the quotient automaton.

1. We check the possibility of adding input transitions to the initial state, *i.e.* $q_0 = \{(\bar{s},\bar{e})\}$. Following q_0 executes $t = \{(s_1,e_2)\}$ and deduction rule I_1 in Definition 15, the transition $q_0 \xrightarrow{t} q_1$ is added to the transition relation of $\bar{s}_{/\bar{e}}$ where $q_1 = \{(s_1,e_2)\}$ (state 1 in Figure 4(d)).
2. We check the possibility of adding output transitions to $q_0 = \{(\bar{s},\bar{e})\}$. We observe that $rev\_rq \in \mathsf{out}(\bar{s}\ \mathsf{after}\ \sigma)$ for every $\sigma \in \{\varepsilon, p\_rq, p\_rq\ \ p\_rs\}$. Regarding deduction rule U_2, the transition $q_0 \xrightarrow{rev\_rq} q_2$ is added to the transition relation of $\bar{s}_{/\bar{e}}$ where $q_2 = \{(s_5,\bar{e}),(s_2,e_1),(s_2,e_3)\}$ (state 2 in Figure 4(d)).
3. Following deduction rule δ_1 and $\delta \notin \mathsf{out}(\bar{s}\ \mathsf{after}\ \varepsilon)$, δ-labeled transition is not added to q_0.

The constructed quotient automaton $\bar{s}_{/\bar{e}}$ is valid: it is both a valid suspension automaton and strongly non-blocking. As a result, $\bar{s}$ is decomposable with respect to $\bar{e}$ and $\bar{s}_{/\bar{e}}$ is a witness to that.

5 Strong Decomposibility

It is a natural question whether the quotient automaton that we defined in the previous section, along with the sufficient conditions for decomposability of a specification provide sufficient conditions for strong decomposability. The proof of Theorem 1 gives some clues to the contrary. A main problem is in the notion of quiescence, and, in particular in the notion of *relative* quiescence, which is unobservable in the standard **ioco** theory. More specifically, the platform $\bar{e}$ may mask the (unwanted) lack of outputs of the quotient automaton.

A natural solution to this is to consider a subclass of implementations called *internal choice* IOTSs, studied in [8, 15]: such implementations *only* accept inputs when reaching a quiescent state. The proposition below states that strong decomposability can be achieved under these conditions.

Theorem 2 *Let $\bar{s} \in \mathsf{IOLTS}(I_s,U_s)$ be a specification and let $\bar{e} \in \mathsf{IOTS}(I_e,U_e)$ be an environment. If $\bar{s}$ is decomposable and $\bar{e}$ is an internal choice IOTS then $\bar{s}$ is* strongly decomposable *and $\bar{s}_{/\bar{e}}$ is a witness to this.*

As a result of the above theorem, testing whether the composition of a component $\bar{c}$ and a platform $\bar{e}$ conforms to specification $\bar{s}$ reduces to testing for the conformance of $\bar{c}$ to $\bar{s}_{/\bar{e}}$. This can be done using the standard **ioco** testing theory [13].

A problem may arise when trying this approach in practice. Namely, the amount of time and memory needed for derivation of the **ioco** test suit increases exponentially in the number of transitions in the specification due to the nondeterministic nature of the test-case generation algorithm. We avoid these complexities by presenting an on-the-fly testing algorithm inspired by [4]. Algorithm 1 describes the

on-the-fly testing algorithm in which sound test cases are generated without constructing the quotient automaton upfront. We partially explored the quotient automaton during test execution. We use the extended version of executes operator in Algorithm 1 which is defined on ordinary IOLTSs; the underlying IOLTSs of suspension automata is used to avoid the complexity of constructing suspension automata, *i.e.* executes $: \mathbb{P}(\mathbb{P}(S_s) \times \mathbb{P}(S_e)) \times L_\delta \times \mathbb{P}(\mathbb{P}(S_s) \times \mathbb{P}(S_e))$.

Algorithm 1 *Let $\bar{s} \in \mathsf{IOLTS}(I_s, U_s)$ be a specification and let $\bar{e} \in \mathsf{IOTS}(I_e, U_e)$ be an environment. Let $\bar{c} \in \mathsf{IOTS}(L_I, L_U)$ be an implementation tested against $\bar{s}$ with respect to $\bar{e}$ by application of the following rules, initializing S with $(\{(\bar{s} \text{ after } \varepsilon), (\bar{e} \text{ after } \varepsilon)\})$ and verdict V with* None*:*
while ($V \notin \{\text{Fail}, \text{Pass}\}$)
{ *apply one of the following case:*

1. *(*provide an input*) Select an $a \in \{a \in L_I \mid S \text{ executes } a \neq \emptyset\}$, then $S = S$ executes a and provide $\bar{c}$ with a*
2. *(*accept quiescence*) If no output is generated by $\bar{c}$ (quiescence situation) and $\forall (s,e) \in S, \sigma \in \mathsf{Straces}(s) \cap \mathsf{Straces}(e) : \ \delta \in \mathsf{out}(s \text{ after } \sigma)$, then $S = S$ executes δ*
3. *(*fail on quiescence*) If no output is generated by $\bar{c}$ (quiescence situation) and $(\exists (s,e) \in S, \sigma \in \mathsf{Straces}(s) \cap \mathsf{Straces}(e) : \ \delta \notin \mathsf{out}(s \text{ after } \sigma))$, then $V =$ Fail*
4. *(*accept a shared output*) If $x \in U_v$ is produced by $\bar{c}$ and S executes $x \neq \emptyset$, then $S = S$ executes x*
5. *(*fail on a shared output*) If $x \in U_v$ is produced by $\bar{c}$ and S executes $x = \emptyset$, then $V =$ Fail*
6. *(*accept an output*) If $x \in U \setminus U_v$ is produced by $\bar{c}$ and $\forall (s,e) \in S, \sigma \in \mathsf{Straces}(s) \cap \mathsf{Straces}(e) \cap (L_\delta^* \setminus L_\delta^* \delta) : \ x \in \mathsf{out}(s \text{ after } \sigma)$, then $S = S$ executes x*
7. *(*fail on an output*) If $x \in U \setminus U_v$ is produced by $\bar{c}$ and $\exists (s,e) \in S, \sigma \in \mathsf{Straces}(s) \cap \mathsf{Straces}(e) \cap (L_\delta^* \setminus L_\delta^* \delta) : \ x \notin \mathsf{out}(s \text{ after } \sigma)$, then $V =$ Fail*
8. *(*nondeterministically terminate*) $V =$ Pass* }

Termination of the above algorithm with $V =$ *Fail* implies that the composition of the implementation under test with $\bar{e}$ does not conform to $\bar{s}$.

Theorem 3 *Let $\bar{s} \in \mathsf{IOLTS}(I_s, U_s)$ be a specification and let $\bar{e} \in \mathsf{IOLTS}(I_e, U_e)$ be an internal choice IOTS environment whose behavior is included in $\bar{s}$. Let V be the verdict upon termination of Algorithm 1 when executed on an implementation $\bar{c}$. If* **hide**$[L_v]$ **in** $\bar{c} \| \bar{e}$ ***ioco*** $\bar{s}$ *then $V =$* Pass.

6 Conclusions

We investigated the property of *decomposability* of a specification in the setting of Tretmans' **ioco** theory for formal conformance testing [12]. Decomposability allows for determining whether a specification can be met by some implementation running on a given platform. Based on a new specification, to which we refer to as the *quotient*, and which we derived from the given one by factoring out the effects of the platform, we identified three conditions (two on the quotient and one on the platform) that together guarantee the decomposability of the original specification.

Any component that correctly implements the quotient is guaranteed to work correctly on the given platform. However, failing implementations provide no information on the correctness of the cooperation between the component and the platform. We therefore studied *strong decomposability*, which

further strengthens the decomposability problem to ensure that only those components that correctly implement the quotient are guaranteed to work correctly on the given platform, meeting the overall specification. This ensures that testing a component against the quotient provides all information needed to judge whether it will work correctly on the platform and meet the overall specification's requirements. However, the complexity of computing the quotient is an exponential problem. We propose an on-the-fly test case derivation algorithm which does not compute the quotient explicitly. Components that fail such a test case provably fail to work on the platform, meeting the overall specification, too.

Checking the inclusion relation of a platform may be expensive in practice. As for future work, we would like to merge the two steps of checking the correctness of the platform and driving the quotient and investigate whether the constraints on the platform can be relaxed by ensuring that the derived quotient masks some of the unwanted behavior of the platform.

References

[1] H. R. Asaadi, R. Khosravi, M. R. Mousavi & N. Noroozi (2011): *Towards Model-Based Testing of Electronic Funds Transfer Systems*. In: *FSEN*, *LNCS* 7141, pp. 253–267. Available at `http://dx.doi.org/10.1007/978-3-642-29320-7_17`.

[2] S. Berezin, S. Campos & E.M. Clarke (1998): *Compositional Reasoning in Model Checking*. In: *Compositionality: The Significant Difference*, *LNCS* 1536, Springer, pp. 81–102. Available at `http://dx.doi.org/10.1007/3-540-49213-5_4`.

[3] M. van der Bijl, A. Rensink & J. Tretmans (2003): *Compositional Testing with ioco*. In: *FATES*, *LNCS* 2931, Springer, pp. 86–100. Available at `http://dx.doi.org/10.1007/978-3-540-24617-6_7`.

[4] R.G. de Vries & J. Tretmans (2000): *On-the-fly Conformance Testing using SPIN*. *STTT* 2(4), pp. 382–393. Available at `http://dx.doi.org/10.1007/s100090050044`.

[5] L. Frantzen & J. Tretmans (2006): *Model-Based Testing of Environmental Conformance of Components*. In: *FMCO*, *LNCS* 4709, Springer, pp. 1–25. Available at `http://dx.doi.org/10.1007/978-3-540-74792-5_1`.

[6] D. Giannakopoulou, C. S. Pasareanu & H. Barringer (2005): *Component Verification with Automatically Generated Assumptions*. *Autom. Softw. Eng.* 12(3), pp. 297–320. Available at `http://dx.doi.org/10.1007/s10515-005-2641-y`.

[7] O. Kupferman & M. Vardi (1998): *Modular model checking*. In: *Compositionality: The Significant Difference*, *LNCS* 1536, Springer, pp. 81–102. Available at `http://dx.doi.org/10.1007/3-540-49213-5_4`.

[8] N. Noroozi, R. Khosravi, M.R. Mousavi & T. A. C. Willemse (2011): *Synchronizing Asynchronous Conformance Testing*. In: *SEFM*, *LNCS* 7041, Springer, pp. 334–349. Available at `http://dx.doi.org/10.1007/978-3-642-24690-6_23`.

[9] N. Noroozi, M.R. Mousavi & T.A.C. Willemse (2013): *Decomposability in Formal Conformance Testing*. Technical Report CSR-13-02, TU/Eindhoven.

[10] C. S. Pasareanu, M. B. Dwyer & M. Huth (1999): *Assume-Guarantee Model Checking of Software: A Comparative Case Study*. In: *Theoretical and Practical Aspects of SPIN Model Checking*, *LNCS* 1680, Springer, pp. 168–183. Available at `http://dx.doi.org/10.1007/3-540-48234-2_14`.

[11] A. Simão & A. Petrenko (2011): *Generating asynchronous test cases from test purposes*. *Information & Software Technology* 53(11), pp. 1252–1262. Available at `http://dx.doi.org/10.1016/j.infsof.2011.06.006`.

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

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

[14] T. Villa, N. Yevtushenko, R.K. Brayton, A. Mishchenko, A. Petrenko & A. Sangiovanni-Vincentelli (2012): *The Unknown Component Problem, Theory and Applications*. Springer, doi:10.1007/978-0-387-68759-9.

[15] M. Weiglhofer & F. Wotawa (2009): *Asynchronous Input-Output Conformance Testing*. In: *COMPSAC*, IEEE Computer Society, pp. 154–159. Available at `http://dx.doi.org/10.1109/COMPSAC.2009.194`.

[16] T. A. C. Willemse (2006): *Heuristics for ioco -Based Test-Based Modelling*. In: *FMICS/PDMC*, *LNCS* 4346, Springer, pp. 132–147. Available at `http://dx.doi.org/10.1007/978-3-540-70952-7_9`.

Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces

Mikhail Chupilko Alexander Kamkin
Institute for System Programming of the Russian Academy of Sciences (ISPRAS)
109004, Russia, Moscow, Alexander Solzhenitsyn st., 25.
{chupilko,kamkin}@ispras.ru

Runtime verification is checking whether a system execution satisfies or violates a given correctness property. A procedure that automatically, and typically on the fly, verifies conformance of the system's behavior to the specified property is called a monitor. Nowadays, a variety of formalisms are used to express properties on observed behavior of computer systems, and a lot of methods have been proposed to construct monitors. However, it is a frequent situation when advanced formalisms and methods are not needed, because an executable model of the system is available. The original purpose and structure of the model are out of importance; rather what is required is that the system and its model have similar sets of interfaces. In this case, monitoring is carried out as follows. Two "black boxes", the system and its reference model, are executed in parallel and stimulated with the same input sequences; the monitor dynamically captures their output traces and tries to match them. The main problem is that a model is usually more abstract than the real system, both in terms of functionality and timing. Therefore, trace-to-trace matching is not straightforward and allows the system to produce events in different order or even miss some of them. The paper studies on-the-fly conformance relations for timed systems (i.e., systems whose inputs and outputs are distributed along the time axis). It also suggests a practice-oriented methodology for creating and configuring monitors for timed systems based on executable models. The methodology has been successfully applied to a number of industrial projects of simulation-based hardware verification.

1 Introduction

Verification has long been recognized as one of the integral parts of software and hardware design processes [15, 22]. Generally, it is an activity intended to check whether a system or its part meets a *specification* (set of functional and timing requirements). Verification techniques can be divided into two main groups, namely *formal verification* and *testing* (also known as *simulation-based verification* in the hardware engineering domain) [14]. Formal methods are aimed at rigorous proving or disproving the correctness of a formal model of a system with respect to a formal specification. Such approaches exhaustively examine all possible executions of a given system – either explicitly (by enumerating all reachable states) or implicitly (by using symbolic techniques). In contrast, testing deals with a finite number of executions and estimates the system's behavior in a finite number of situations (so-called *test situations*). *Runtime verification* is a common point of both. Like testing, it works with concrete executions of a system, but does it in a formal way.

In runtime verification, a correctness property is typically expressed in a formal language, which makes it possible to automatically translate the property into a *monitor*. Such a monitor is then used to check a system execution with respect to that property [5]. The idea is similar to *specification-based testing*, where a formal specification serves as a basis for generating a *test oracle*, which, like monitor, determines whether an observed behavior is consistent with the specification [11, 12]. But, as opposed to testing, it is not a scope of runtime verification to construct test sequences and apply them to the

A. Petrenko, H. Schlingloff (Eds.): Eighth Workshop on Model-Based Testing (MBT 2013)
EPTCS 111, 2013, pp. 67–81, doi:10.4204/EPTCS.111.6

system under test. The task is to passively observe inputs and outputs of the system and to check their conformance – that is why it is also called *passive testing* [3]. Formally, when $\mathcal{L}(\varphi)$ denotes the set of valid system executions given by property φ, runtime verification is aimed at checking whether a concrete execution w is an element of $\mathcal{L}(\varphi)$. In this sense, runtime verification deals with the *word problem*, i.e., identifying whether a given word is included in some language [5].

Correctness properties in runtime verification may be expressed using a variety of different formalisms, including *extended regular expressions* [21], *contract specifications* [12] and *rule-based approaches* [4]. *Temporal logic*, which is well-known from *model checking* [10], is also very popular in runtime verification, especially variants of *linear temporal logic*, such as LTL and TLTL (a natural counterpart of LTL in the timed setting) [5]. There are also a lot of methods for generating effective monitors (or test oracles) from formal specifications. However, sophisticated formalisms and methods are not always suitable for industrial practice. For example, many hardware design companies use *executable software models* for design space exploration and architecture modeling; it is quite natural to *reuse* those models for verification and monitoring. High reusability within a project is important to complete verification within the timeline [19]. Moreover, reusable models ensure conceptual integrity of the project and accelerate the knowledge interchange.

Runtime verification based on executable models is carried out in the following way. A reference model is *co-executed* with the target system and applied with the same inputs as the system under verification. The outputs of two "black boxes" are given to the monitor that matches them and decides whether they are consistent. Aside from minor technical difficulties on organizing co-execution and transforming interfaces, there is a conceptual problem relating to *model abstractness*. As a rule, a model (tending to be as simple as possible) does not specify the system's behavior accurately, which makes the output matching awkward. If the model produces some outputs in some order, it does not necessarily mean that the system should do it in the same manner – the order may differ and some of the ouputs may be omitted. Before using a model for monitoring one has to specify a priori information on its abstractness and give it to the monitor. One of the contributions of this paper is an approach that allows easy adaptation of monitors for models represented in different abstraction levels.

We consider *timed systems*, which react on inputs distributed in time and emit outputs at dedicated time points. Formally, it means that each event is paired together with a time stamp, identifying when exactly the event happened. For the *discrete-time model*, timed sequences of events can be easily transformed into ordinary ones by removing time stamps and inserting a special *tick* event in proper positions of the original sequence (as many times as necessary) [2]. Nevertheless, even in that case, it is convenient to suppose that each event is tagged with a time stamp. Executions of a system and its model are described by *timed sequences* over the same alphabet. Assumptions on the model abstractness allow dynamical generalization of linear sequences into the *partially ordered multisets* consisting of events and *time intervals* associated with them. In general terms, the monitor checks on the fly that an implementation trace is a linearization of the generalized specification trace (subset of the trace) and all implementation events satisfy the corresponding time interval constraints.

The rest of the paper is organized as follows. Section 2 introduces the basic mathematical notions used in the work such as a *timed word*, *trace* and *pomset*. Section 3 is the main part of the paper, in which the suggested method for timed trace matching is described. The section formalizes implementation and specification behavior and defines a conformance relation between implementations and specifications. It also describes a monitoring approach in detail and states its correctness. Section 4 outlines our experience in using the proposed approach for simulation-based verification of industrial hardware designs. Section 5 is a brief survey of the related work. Section 6 concludes the paper and discusses some of our future research directions.

2 Preliminaries

For the rest of the paper, Σ denotes a finite alphabet of *events*, while $\mathbb{T}$ denotes a *time domain*. An event might be considered as a set of propositions that identify a situation when the event happens. A time domain is a totally ordered set with no upper bound, typically, $\mathbb{N}$ (discrete-time model) or $\mathbb{R}^{\geq 0}$ (continuous-time model). Sequences of events are called *words* (the *empty word* is denoted by ε). Symbols Σ^* and Σ^ω stand for the sets of *finite* and *infinite* words over Σ, respectively. The length of a word w is denoted by $|w|$. If u and v are two words over the same alphabet and u is finite, then uv denotes their *concatenation*. For $w = uv$ we say that w is a *continuation* of u with v.

Sometimes, it is useful to structurize events by dividing them into *inputs* and *outputs* ($\Sigma = I \cup O$) and by introducing a notion of *port* [17]. Let $P = \{1, 2, ..., k\}$ and $\mathsf{port} : \Sigma \to P$. Then, the tuple $\langle \Sigma_1, ..., \Sigma_k \rangle$, where $\Sigma_p = \mathsf{port}^{-1}(p)$, is called a *distributed alphabet*.

Definition 1 (Timed word – Alur and Dill [2]) *A timed word w over the alphabet Σ and the time domain $\mathbb{T}$ is a sequence $(a_0,t_0)(a_1,t_1)...$ of timed events $(a_i,t_i) \in \Sigma \times \mathbb{T}$, satisfying the following constraints:*

1. *for each $i \geq 0$, $t_i < t_{i+1}$ holds (monotonicity);*
2. *if the sequence is infinite, for every $t \in \mathbb{T}$ there is some $i \geq 0$, such that $t_i > t$ (progress).* □

Strict monotonicity in the definition above can be weaken to monotonicity (i.e., it can be required that $t_i \leq t_{i+1}$ for all $i \geq 0$) [2]. $(\Sigma \times \mathbb{T})^*$ and $(\Sigma \times \mathbb{T})^\omega$ denote the sets of finite and infinite timed words, respectively. Note that port partitioning implies an additional constraint on a timed word:

3. for all $i, j \geq 0$, such that $i \neq j$ and $t_i = t_j$, $\mathsf{port}(e_i) \neq \mathsf{port}(e_j)$ (*sequentiality*).

In concurrent systems, the concept of *independence* is often in use. Two events are considered as *independent* if they cannot be causally related (i.e., they may happen concurrently). Events on different ports are usually independent, while those on the same port are dependent. Concurrent execution can be modeled by partially ordered traces of events, where incomparable events are supposed to occur in indeterminate order or in parallel [6]. This intuition underlies two formal models of non-interleaving concurrency: (1) *Mazurkiewicz's trace model* [18] and (2) *Pratt's pomset model* [20]. The definitions and their extensions for the timed case are given below.

Definition 2 (Trace – Mazurkiewicz [18]) *An independence relation over the alphabet Σ is a symmetric and irreflexive relation $\mathcal{I} \subset \Sigma \times \Sigma$. Given an independence relation $\mathcal{I}$, a pair $\langle \Sigma, \mathcal{I} \rangle$ is called a concurrent alphabet. Two words u and v are called Mazurkiewicz equivalent ($u \equiv_\mathcal{I} v$) iff u can be transformed to v by a finite number of exchanges of adjacent, independent events. A Mazurkiewicz trace (or, simply, a trace) is an equivalence class of words by the Mazurkiewicz equivalence relation.* □

The set of traces over the concurrent alphabet $\langle \Sigma, \mathcal{I} \rangle$ is denoted as $\mathbb{M}(\Sigma, \mathcal{I})$. Given an independence relation $\mathcal{I}$, the relation $\mathcal{D} = (\Sigma \times \Sigma) \setminus \mathcal{I}$ is called the *dependence relation*. The *length* of a trace τ (denoted by $|\tau|$) is the length of any of its representatives. If w is a word, $[w]_\mathcal{I}$ is the trace that includes w as a representative. A *concatenation* of traces over the same concurrent alphabet $\langle \Sigma, \mathcal{I} \rangle$ is defined by the equality $[u]_\mathcal{I}[v]_\mathcal{I} = [uv]_\mathcal{I}$. A trace σ is called a *prefix* of τ ($\sigma \sqsubseteq \tau$) iff there exists γ, such that $\sigma\gamma = \tau$.

Example 1 (Traces) *Let $\Sigma = \{a,b,c,d\}$ and $\mathcal{I} = \{(a,b),(b,a),(c,d),(d,c)\}$. Then, some traces are as follows:*

$$\begin{aligned} [\varepsilon]_\mathcal{I} &= \{\varepsilon\} \\ [ad]_\mathcal{I} &= \{ad\} \\ [ab]_\mathcal{I} &= \{ab, ba\} \\ [abcd]_\mathcal{I} &= \{abcd, bacd, abdc, badc\} \end{aligned}$$

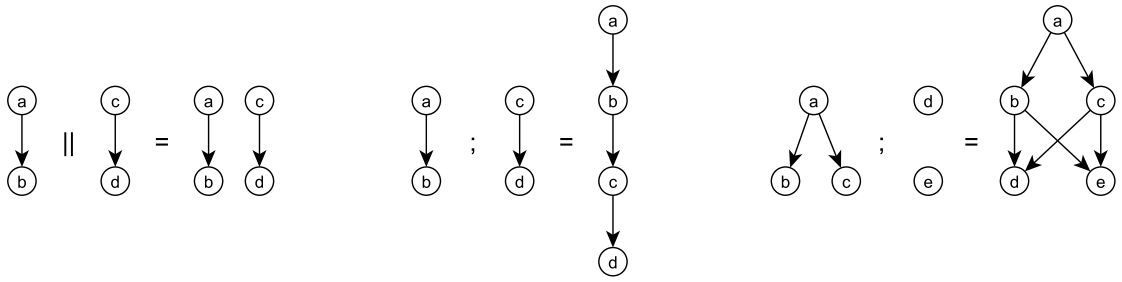


Figure 1: Sequential and parallel composition of simple pomsets

Definition 3 (Pomset (partially ordered multiset) – Pratt [20]) *A Σ-labeled partial order is a tuple $\langle V, \preceq, \lambda \rangle$, where V is a finite set of vertices and $\lambda : V \to \Sigma$ is the labeling function. Two Σ-labeled partial orders are called equivalent iff they are order- and label-isomorphic (i.e., they are either equal or differ only in the names of vertices). A pomset over the alphabet Σ is an isomorphism class of Σ-labeled partial orders.* □

Note that words are equivalent to pomsets with the total order, while multisets are equivalent to pomsets whose partial order is the equality. For convenience, we will use a concrete representative (a labeled partial order) to denote the pomset. There is a number of operations on pomsets, including *parallel* and *sequential composition*. Let $\sigma = \langle V, \preceq, \lambda \rangle$ and $\gamma = \langle V', \preceq', \lambda' \rangle$ are pomsets over the same alphabet, such that $V \cap V' = \varnothing$. Define the pomsets $(\sigma \parallel \gamma)$ and $(\sigma \,;\, \gamma)$ as follows:

$$\begin{array}{rcl} (\sigma \parallel \gamma) & = & \langle V \cup V', \preceq \cup \preceq', \lambda \cup \lambda' \rangle \\ (\sigma \,;\, \gamma) & = & \langle V \cup V', \preceq \cup \preceq' \cup (V \times V'), \lambda \cup \lambda' \rangle \end{array}$$

Example 2 (Pomsets) *Examples of pomsets in the form of Hasse diagrams (i.e., drawings of the partial order transitive reduction), may be found in Figure 1.*

A *linearization* of a pomset $\langle V, \preceq, \lambda \rangle$ is a total labelled order $\langle V, \leq, \lambda \rangle$, where $\preceq \subseteq \leq$. The set of linearizations of a pomset σ is denoted by $\mathsf{lin}(\sigma)$. A designation $x \perp y$ means that neither $x \preceq y$ nor $y \preceq x$. We say that $x \in V$ *immediately precedes* $y \in V$ and write $x \dot{\prec} y$ iff $x \prec y$ and there is no such $z \in V$ that $x \prec z \prec y$. A *history* of $x \in V$ is the set $\downarrow x = \{y \in V \mid y \preceq x\}$ (for $X \subseteq V$, $\downarrow X = \bigcup_{x \in X} \downarrow x$).

It can be shown that each trace can be represented as a pomset. The opposite is true only for a restricted class of pomsets [6]. Let $\langle \Sigma, \mathcal{I} \rangle$ be a concurrent alphabet and $\sigma = \langle V, \prec, \lambda \rangle$ be a pomset, such that

- for each $x \in V$, $\downarrow x$ is a finite set;
- for all $x, y \in V$, if $x \perp y$, then $\big(\lambda(x), \lambda(y)\big) \in \mathcal{I}$;
- for all $x, y \in V$, if $x \dot{\prec} y$, then $\big(\lambda(x), \lambda(y)\big) \in \mathcal{D}$.

Then, $\mathsf{lin}(\sigma)$ is a trace over $\langle \Sigma, \mathcal{I} \rangle$ and $\sigma = \mathsf{pom}(\mathsf{lin}(\sigma))$ [6]. Further, we will represent traces as pomsets satisfying the conditions above. The same consideration is done in [16, 8].

Definition 4 (Timed trace – Chieu and Hung [8]) *A timed trace over the concurrent alphabet $\langle \Sigma, \mathcal{I} \rangle$ and the time doman $\mathbb{T}$ is a quadruple $\langle V, \preceq, \lambda, \theta \rangle$, where $\langle V, \preceq, \lambda \rangle$ is a trace over $\langle \Sigma, \mathcal{I} \rangle$ and $\theta : V \to \mathbb{T}$ is a time function satisfying the following conditions:*

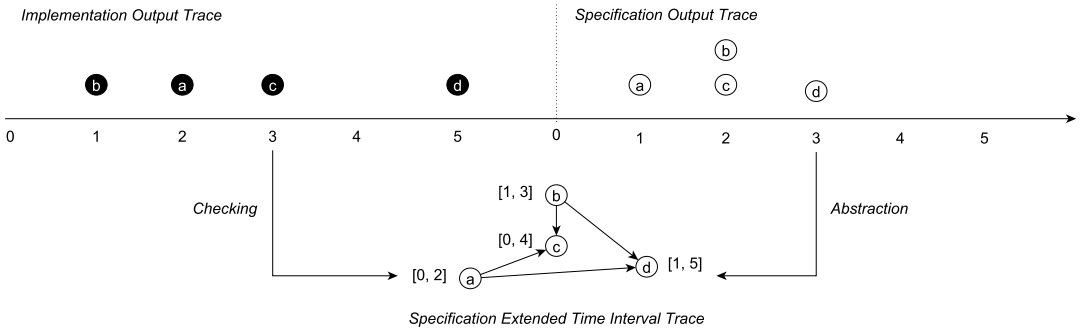


Figure 2: Scheme for checking conformance between implementation and specification

1. *for all $x,y \in V$, if $x \prec y$, then $\theta(x) < \theta(y)$ (causality);*
2. *if the trace is infinite, then for every $t \in \mathbb{T}$ there is a cut $C \subseteq V$, such that $\min_{x \in C}\{\theta(x)\} \geq t$ (progress).* $\square$

The set of timed traces over the concurrent alphabet $\langle \Sigma, \mathcal{I} \rangle$ and the time domain $\mathbb{T}$ is denoted as $\mathbb{M}_\theta(\Sigma, \mathcal{I}, \mathbb{T})$. Note that timed words are a particular case of timed traces. Given a non-empty timed trace $\sigma = \langle V, \preceq, \lambda, \theta \rangle$, $\mathsf{begin}(\sigma) = \min_{x \in V}\{\theta(x)\}$ and $\mathsf{end}(\sigma) = \max_{x \in V}\{\theta(x)\}$ (if σ is infinite, $\mathsf{end}(\sigma) = \infty$); $\sigma_{[t,t+\Delta t]}$ is a sub-trace of σ consisting of $x \in V$, such that $\theta(x) \in [t, t+\Delta t]$. Let $\mathcal{TI}(\mathbb{T})$ be the set of time intervals over the time domain $\mathbb{T}$ (i.e., $\mathcal{TI}(\mathbb{T}) = \{[t, t+\Delta t] \mid t, t+\Delta t \in \mathbb{T}\}$).

Definition 5 (Time interval trace) *A time interval trace over the concurrent alphabet $\langle \Sigma, \mathcal{I} \rangle$ and the time doman $\mathbb{T}$ is a quadruple $\sigma = \langle V, \preceq, \lambda, \delta \rangle$, where $\langle V, \preceq, \lambda \rangle$ is a trace over $\langle \Sigma, \mathcal{I} \rangle$ and $\delta : V \to \mathcal{TI}(\mathbb{T})$ is a function that associates a time interval to a vertex. The language of the time interval trace σ is the set $\mathcal{L}(\sigma) = \{\langle V, \preceq, \lambda, \theta \rangle \in \mathbb{M}_\theta(\Sigma, \mathcal{I}, \mathbb{T}) \mid \forall x \in V \,.\, \theta(x) \in \delta(x)\}$.* $\square$

The set of time interval traces over the concurrent alphabet $\langle \Sigma, \mathcal{I} \rangle$ and the time domain $\mathbb{T}$ is denoted as $\mathbb{M}_\delta(\Sigma, \mathcal{I}, \mathbb{T})$. Futher we will deal with pairs consisting of a timed trace σ and a time interval trace σ_δ, such that $\sigma \in \mathcal{L}(\sigma_\delta)$. Such a pair can be expressed as a quintuple $\langle V, \preceq, \lambda, \theta, \delta \rangle$ and is referred to as an *extended time interval trace*. The set of such traces is denoted as $\mathbb{M}_{\theta\delta}(\Sigma, \mathcal{I}, \mathbb{T})$.

3 Runtime Verification with Executable Models

A timed word (more precisely, a timed trace with an empty partial order) describes a concrete execution of the *implementation* under verification, while an extended time interval trace being more general can be considered as a *specification* behavior. Our goal is to check whether an implementation timed word $w_I \in (\Sigma \times \mathbb{T})^{*(\omega)}$ is conforming to a specification trace $\sigma_S \in \mathbb{M}_{\theta\delta}(\Sigma, \mathcal{I}, \mathbb{T})$. Note that we are interested in *on-the-fly* checking, which means that a monitor "lives" in time and matches two traces in an *event-driven* fashion. *Trace acceptance* (*verdict*) at a given time point has a three-valued semantics [5]: (1) $false$ (an inconsistency has been detected), (2) $true$ (the implementation execution has been completed and its trace is conforming to the specification trace) and (3) *inconclusive* (the monitoring is in progress and no inconsistency has been found).

To make it clear where a specification trace comes from, an additional explanation should be provided. As it was said in the introduction, a system specification is represented in the *executable* form.

Hence, it can be executed and its executions (as ones of the implementation) are represented as timed words. The straightforward testing of the equality of two timed words is often inadequate and makes sense only for *time-accurate* specifications. Specifications are usually more abstract than implementations, especially in terms of *event ordering* and *timing*. Assumptions on the specification abstractness generalize a concrete timed word to the extended time interval trace softening the conformance checking. Formally, *abstraction* is a map $\mathcal{A} : (\Sigma \times \mathbb{T})^{*(\omega)} \to \mathbb{M}_{\theta\delta}(\Sigma, \mathcal{I}, \mathbb{T})$, such that w is conforming to $\mathcal{A}(w)$ for every $w \in (\Sigma \times \mathbb{T})^{*(\omega)}$. A specification timed word w_S is mapped into the extended time interval trace $\mathcal{A}(w_S) = \sigma_S$. Then, it is checked whether an implementation word w_I is conforming to the constructed specification trace σ_S. This scheme is illustrated in Figure 2. Technical details can be found in Section 4.

3.1 Conformance Relation

The next definition formalizes system executions in terms of timed traces. It also singles out *input* and *output sequences* as particular cases of traces corresponding to stimuli to a system and its reactions, respectively. System behavior is then abstractly defined as a map of inputs to outputs.

Definition 6 (Execution trace) *An execution trace over the concurrent alphabet $\langle \Sigma, \mathcal{I} \rangle$ and the time domain $\mathbb{T}$ is a timed trace with the empty partial order (i.e., a trace of the kind $\langle V, \varnothing, \lambda, \theta \rangle$). If $\Sigma = I \cup O$, then execution traces over the alphabet I are called input sequences, while execution traces over the alphabet O are referred to as output sequences.* □

Note that the empty partial order in execution traces reflects a fact that an implementation is a "black box", and, therefore, the cause-effect relation between its events is unknown. The sets of input and output sequences are designated by $\mathbb{I}_\theta(\Sigma, \mathbb{T})$ and $\mathbb{O}_\theta(\Sigma, \mathbb{T})$, respectively. Hereinafter, we will use the shortened notations: $\mathbb{I} = \mathbb{I}_\theta(\Sigma, \mathbb{T})$ and $\mathbb{O} = \mathbb{O}_\theta(\Sigma, \mathbb{T})$.

Definition 7 (Behavior) *Deterministic timed behavior (or, simply, behavior) over the alphabet Σ and the time domain $\mathbb{T}$ is a (partial) map $\mathcal{B} : \mathbb{I} \times \mathbb{T} \to \mathbb{O}$ satisfying the following constraints:*

- *for every $w \in \mathbb{I}$ and $t \in \mathbb{T}$, $\mathsf{end}\big(\mathcal{B}(w,t)\big) \le t$ holds (future uncertainty);*
- *for every $w \in \mathbb{I}$ and $t \in \mathbb{T}$, $\mathcal{B}(w,t) = \mathcal{B}(w_{[0,t]}, t)$ holds (time directivity);*
- *for every $w \in \mathbb{I}$ and every $t \in \mathbb{T}$, there exists $wv \in \mathbb{I}$, a continuation of w, and $\Delta t \ge 0$, such that $\mathsf{end}\big(\mathcal{B}(wv, t + \Delta t)\big) \ge t$ (liveness).* □

The idea behind the concept is clear. Behavior describes how an input sequence is transformed into the output sequence taking into account an observation time point. Usually, when an input sequence is applied, then after a finite number of time units (counting from the last input time) the output sequence is fully observed and is ready to be checked. Such post-mortem analysis is not however what we are interested in. There are two reasons for that: (1) to ease the analysis, an execution should be terminated as soon as a failure is detected; (2) storing long sequences in memory is costly. Providing that a reference model is available, consider how it can be used for checking implementation behavior in runtime. Let us extend the definition above by allowing a specification to return extended time interval traces over the outputs (not concrete sequences as it is required). Denote the set of such traces as $\mathbb{O}_{\theta\delta}$.

Given an output trace $\langle V, \preceq, \lambda, \theta, \delta \rangle \in \mathbb{O}_{\theta\delta}$, define two functions, $\Delta t^{\pm}$, such that for every $x \in V$, $\delta(x) = [\theta(x) - \Delta t^{-}(x), \theta(x) + \Delta t^{+}(x)]$. Assume that functions $\Delta t^{\pm}$ are bounded (i.e., there exist constants $\Delta T^{\pm} > 0$, such that $|\Delta t^{\pm}(x)| \le \Delta T^{\pm}$ for all $x \in V$). Assume also that values $\Delta t^{\pm}(x)$ depend only on the event not on the vertex itself (i.e., $\Delta t^{\pm}(x) = \Delta t^{\pm}(\lambda(x))$). Let $\mathcal{I}$ and $\mathcal{S}$ be an *implementation* and *specification behavior*, respectively. Given an input sequence $w \in \mathbb{I}$, a time point $t \in \mathbb{T}$, let us consider

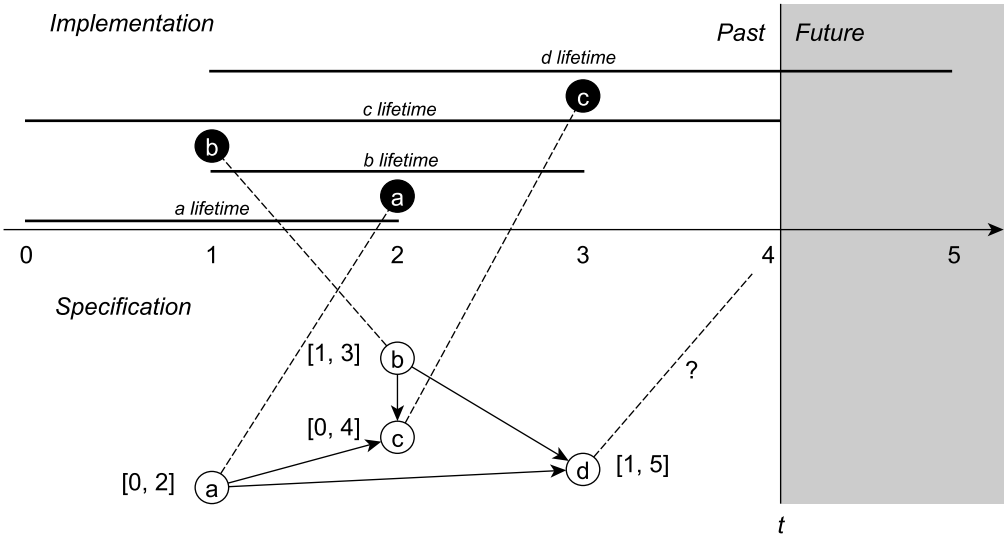


Figure 3: Conformance between implementation and specification

implementation and specification outputs: $\mathcal{I}(w,t) = \langle V_\mathcal{I}, \varnothing, \lambda_\mathcal{I}, \theta_\mathcal{I} \rangle$ and $\mathcal{S}(w,t) = \langle V_\mathcal{S}, \preceq_\mathcal{S}, \lambda_\mathcal{S}, \theta_\mathcal{S}, \delta_\mathcal{S} \rangle$. Let us introduce the following notations:

$$\begin{array}{rcl}
\mathsf{past}^{\Delta t}_\mathcal{I}(w,t) & = & \{y \in \mathcal{I}(w,t) \mid \theta_\mathcal{I}(y) \le \big(t - \Delta t^-(y)\big)\}; \\
\mathsf{past}_\mathcal{I}(w,t) & = & \{y \in \mathcal{I}(w,t) \mid \theta_\mathcal{I}(y) \le t\}; \\
\mathsf{past}^{\Delta t}_\mathcal{S}(w,t) & = & \{x \in \mathcal{S}(w,t) \mid \theta_\mathcal{S}(x) \le \big(t - \Delta t^+(x)\big)\}; \\
\mathsf{past}_\mathcal{S}(w,t) & = & \{x \in \mathcal{S}(w,t) \mid \theta_\mathcal{S}(x) \le t\}; \\
\mathsf{match}(x,y) & = & \big(\lambda_\mathcal{I}(y) = \lambda_\mathcal{S}(x)\big) \wedge \big(\theta_\mathcal{I}(y) \in \delta_\mathcal{S}(x)\big).
\end{array}$$

Definition 8 (Conformance relation) *The implementation behavior $\mathcal{I}$ is said to be conforming to the specification behavior $\mathcal{S}$ iff* $\mathsf{dom}\mathcal{I} = \mathsf{dom}\mathcal{S}$ *and for all $w \in \mathsf{dom}\mathcal{S}$ and $t \in \mathbb{T}$, there is a relation $\mathcal{M}(w,t) \subseteq \{(x,y) \in \mathsf{past}_\mathcal{S}(w,t) \times \mathsf{past}_\mathcal{I}(w,t) \mid \mathsf{match}(x,y)\}$ (called a matching relation), such that:*

1. *$\mathcal{M}(w,t)$ is a one-to-one relation;*
2. *for each $x \in \mathsf{past}^{\Delta t}_\mathcal{S}(w,t)$, there is $y \in \mathsf{past}_\mathcal{I}(w,t)$, such that $(x,y) \in \mathcal{M}(w,t)$;*
3. *for each $y \in \mathsf{past}^{\Delta t}_\mathcal{I}(w,t)$, there is $x \in \mathsf{past}_\mathcal{S}(w,t)$, such that $(x,y) \in \mathcal{M}(w,t)$;*
4. *for all $(x,y),(x',y') \in \mathcal{M}(w,t)$, if $x \prec x'$, then $\theta_\mathcal{I}(y) \le \theta_\mathcal{I}(y')$.*

If for some $w \in \mathbb{I}$ and $t \in \mathbb{T}$ the abovementioned properties are violated, then $\mathcal{I}$ is said to be not conforming to $\mathcal{S}$, and $w_{[0,t]}$ is referred to as a counterexample. □

Figure 3 illustrates the conformance relation definition for a particular input sequence (being unimportant it is not shown in the picture) and observation time ($t = 4$). The upper part of the figure is a drawing of the implementation outputs (black circles with white labels: *b*, *a* and *c*). The lower part depicts the specification outputs (white circles with black labels: *a*, *b*, *c* and *d*). Let us denote the trace

vertices (i.e., circles themselves) by y_b, y_a and y_c (for the implementation) and x_a, x_b, x_c and x_d (for the specification). The implementation vertices are not causally related to each other, while the specification vertices are partially ordered (the precedence relation is drawn by arrows: $x_a \prec x_c$, $x_b \prec x_c$, $x_a \prec x_d$ and $x_b \prec x_d$) and are tagged with time intervals ($\delta(x_a) = [0,2]$, $\delta(x_b) = [1,3]$, $\delta(x_c) = [0,4]$ and $\delta(x_d) = [1,5]$). Matchings are depicted by intermittent lines connecting the implementation vertices with the specification ones ((x_a, y_a), (x_b, y_b) and (x_c, y_c)). It is easy to see that this relation fits the matching relation definition: (1) it is one-to-one relation; (2 & 3) it includes all events whose lifetime has been exhausted; (4) is preserves the specification ordering:

- $\big(x_a \prec x_c\big)$ and $\big(\theta(y_a) = 2 \leq 3 = \theta(y_c)\big)$;
- $\big(x_b \prec x_c\big)$ and $\big(\theta(y_b) = 1 \leq 3 = \theta(y_c)\big)$.

And, certainly, this relation satisfies the matching condition:

- $\big(\lambda(x_a) = \lambda(y_a) = a\big)$ and $\big(\theta(y_a) = 2 \in [0,2] = \delta(x_a)\big)$;
- $\big(\lambda(x_b) = \lambda(y_b) = b\big)$ and $\big(\theta(y_b) = 1 \in [1,3] = \delta(x_b)\big)$;
- $\big(\lambda(x_c) = \lambda(y_c) = c\big)$ and $\big(\theta(y_c) = 3 \in [0,4] = \delta(x_c)\big)$.

The next section describes a procedure that automatically and dynamically constructs a matching relation between implementation and specification outputs. If it fails to create such a relation, it reports the reason, which can interpreted as a failure type: a *missing* or *unexpected* implementation output.

3.2 On-the-Fly Trace Matching

A monitor that matches implementation and specification traces and checks their conformance is co-executed with the implementation and specification and reacts on their outputs. Formally, the monitor can be expressed as a *timed automaton* [2] with two types of input ports: (1) ports for receiving *specification outputs* and (2) ports for receiving *implementation outputs*. When the automaton detects inconsistency between implementation and specification traces, it goes into a dedicated state informing that the implementation is not conforming to the specification.

A formal description of the on-the-fly trace matcher is given below. It is represented as a system of *guarded actions*. Each *action* is atomic and is executed as soon as the *guard* is $true$. The actions and their guards depend on an external variable t reflecting the current *simulation time* and outputs produced by the specification and implementation in response to the same input sequence (S and I, respectively). The value of t is monotonically increasing in *real time* (simulation time may coincide with real time). The writing $y \in I[t]$ means that at time t the implementation omits an output y. The description is based on two functions: (1) the *primary arbiter* ($arbiter_S$) and (2) the *secondary arbiter* ($arbiter_I$), which are defined as follows:

$$arbiter_S(X) = \begin{cases} \mathsf{min}_{\preceq}(X) & \text{if } X \neq \varnothing, \\ \phi & \text{otherwise } (\phi \notin \Sigma); \end{cases}$$

$$arbiter_I(y, X) = \begin{cases} \arg\,\mathsf{min}_{x \in X.\mathsf{match}(x,y)}\{\theta_{\mathbb{S}}(x)\} & \text{if there is } x \in X, \text{ such that } \mathsf{match}(x,y), \\ \phi & \text{otherwise.} \end{cases}$$

Action 1 $onSpecOutput[x], x \in S[t]$

Guard: $true$
Input: x
$past_S \Leftarrow past_S \cup \{x\}$
if $x \in arbiter_S(past_S)$ **then**
for all $y \in past_I$ [in ascending of $\theta_{\mathbb{I}}(y)$] **do**
if $x = arbiter_I(y, \{x\})$ **then**
$past_S \Leftarrow past_S \setminus \{x\}$
$past_I \Leftarrow past_I \setminus \{y\}$
$match \Leftarrow match \cup \{\langle x, y\rangle\}$
trace($\langle x, y\rangle$, "Conforming output")
break
end if
end for
end if

Action 2 $onImplOutput[y], y \in I[t]$

Guard: $true$
Input: y
$past_I \Leftarrow past_I \cup \{y\}$
$x \Leftarrow arbiter_I(y, arbiter_S(past_S))$
if $x \neq \phi$ **then**
$past_S \Leftarrow past_S \setminus \{x\}$
$past_I \Leftarrow past_I \setminus \{y\}$
$match \Leftarrow match \cup \{\langle x, y\rangle\}$
trace($\langle x, y\rangle$, "Conforming output")
end if

Action 3 $onSpecTimeout[x], x \in past_S$

Guard: $\left(\theta_{\mathbb{S}}(x) + \Delta t^+(x)\right) \leq t$
Input: x
$past_S \Leftarrow past_S \setminus \{x\}$
$verdict \Leftarrow false$‘
trace($\langle x, \phi\rangle$, "Missing output")
terminate

Action 4 $onImplTimeout[y], y \in past_I$

Guard: $\left(\theta_{\mathbb{I}}(y) + \Delta t^-(y)\right) \leq t$
Input: y
$past_I \Leftarrow past_I \setminus \{y\}$
$verdict \Leftarrow false$
trace($\langle \phi, y\rangle$, "Unexpected output")
terminate

Action 5 $onInitialize$

Guard: $t = 0$
Input: $\varnothing$
$past_S \Leftarrow \varnothing$
$past_I \Leftarrow \varnothing$
$match \Leftarrow \varnothing$

Action 6 $onFinalize$

Guard: $\left(\mathbf{end}(S) + \Delta T^+\right) \leq t \wedge \left(\mathbf{end}(I) + \Delta T^-\right) \leq t$
Input: $\varnothing$
$verdict \Leftarrow true$
terminate

Given a time point, the timeout actions (*onSpecTimeout* and *onImplTimeout*), if they are activated, are called after the output reception actions (*onSpecOutput* and *onImplOutput*). Otherwise, there might be a *false negative*. E.g., when the implementation sends an output y at time t and there is $x \in past_S$, such that $\lambda_{\mathbb{S}}(x) = \lambda_{\mathbb{I}}(y)$ and $\left(\theta_{\mathbb{S}}(x) + \Delta t^+(x)\right) = t$ (thus, $\theta_{\mathbb{I}}(y)$ is a boundary point of $\delta_{\mathbb{S}}(x)$), calling *onSpecTimeout* before *onImplOutput* would lead to the undesirable failure. If there are two specification outputs x and x', such that $\theta_{\mathbb{S}}(x) = \theta_{\mathbb{S}}(x')$ and $x \prec x'$, calling $onSpecOutput[x]$ should precede calling $onSpecOutput[x']$. The initialization action (*onInitialize*) comes first, while the finalization action (*onFinalize*) is the last action within a time slot. The order between the timeout actions as the order between the output reception actions is insufficient and may be arbitrary. The sequence for checking guards and activating actions within a time slot t is as follows:

1. initialization (*onInitialize*);
2. output reception ($onSpecOutput[x]$ and $onImplOutput[y]$, $x \in S[t]$ and $y \in I[t]$);

3. timeouts ($onSpecTimeout[x]$ and $onImplTimeout[y]$, $x \in past_S$ and $y \in past_I$);
4. finalization ($onFinalize$).

Note that when we say that some property φ holds at time t, we mean that φ holds after all of the actions activated at time t have completed. For a multi-port system, the monitor can be decomposed into a number of loosely connected sub-monitors serving individual ports. If the specification abstracts away from the inter-port dependencies, the sub-monitors are fully independent and can work in parallel.

Statement 1 (Monitor correctness) *An input sequence w is a counterexample for $\mathcal{I}$ being conforming to $\mathcal{S}$ iff the monitor terminates with verdict $= false$.* □

Rigorously speaking, the termination condition $\big(\mathsf{end}(I) + \Delta T^-\big) \le t$ cannot be checked for "black-box" implementations (a monitor is not able to identify whether the implementation is *quiescent* or *active*). However, for some types of systems (in particular, systems with *covergent behavior*) the condition can be approximated with a checkable one.

Definition 9 (Convergent behavior) *The behavior $\mathcal{B} : \mathbb{I} \times \mathbb{T} \to \mathbb{O}$ is called convergent iff the following conditions are met:*

- *for every finite $w \in \mathbb{I}$, there exists $T(w) \in \mathbb{T}$, called the stabilization time, such that for any $t \ge T(w)$, $\mathcal{B}(w,t) = \mathcal{B}\big(w, T(w)\big)$ ($\mathcal{B}(w)$ denotes $\mathcal{B}\big(w, T(w)\big)$);*
- *for every $t \in \mathbb{T}$, $\mathcal{B}(\varepsilon, t) = \varepsilon$ holds (the initial state is quiescent);*
- *for every finite $w, v \in \mathbb{I}$, such that $v \ne \varepsilon$ and $t_0 = \mathsf{begin}(v) > T(w)$, $t \ge t_0$ and $\Delta t \in \mathbb{T}$,*

$$\begin{cases} \mathcal{B}\big(w(v+\Delta t), t+\Delta t\big)_{[t_0+\Delta t, t+\Delta t]} = \mathcal{B}(wv, t)_{[t_0, t]} + \Delta t, \\ t_0 \le \mathsf{begin}(\mathcal{B}(wv)_{[T(w),\infty)}), \text{ if } \mathcal{B}(\cdot)_{[\cdot)} \ne \varepsilon; \end{cases}$$

where $w + \Delta t$ denotes the sequence constructed from w by adding Δt to each time stamp of w (quiescent states are stable). □

Assuming that the implementation under verification is convergent, the termination condition may be expressed as follows:

$$\Big(T(w) \le t\Big) \wedge \Big(\big(\mathsf{end}(I_{[0,T(w)]}) + \Delta T^-\big) \le t\Big).$$

3.3 Specifications with Optional Outputs

There are systems where operations in some situations terminate other operations, conflicting with them and of a lower priority. For example, a write operation can be cancelled by another write operation targeted at the same location and started right after the previous one. Due to abstractness, a specification is not able to express precisely under what conditions operations are cancelled and their output is not sent outside. Taking into account such problems, the definition of the specification behavior should be extended. Assume there is an unary relation $\Diamond \subseteq V_{\mathcal{S}}$ marking cancellable outputs (the complement of $\Diamond$ is denoted by $\Box$): if $\Diamond x$, then the output is *optional* (it might be cancelled, but the cancellation condition is unknown or inexpressible in specification terms); if $\Box x$, then the output is *obligatory* (it cannot be cancelled). Note that if some action is cancelled, then all dependent actions are cancelled either.

Definition 10 (Conformance relation for specifications with optional outputs) *The implementation behavior $\mathcal{I}$ is said to be conforming to the specification behavior with optional outputs $\mathcal{S}$ iff* $\mathbf{dom}\mathcal{I} = \mathbf{dom}\mathcal{S}$ *and for all $w \in \mathbf{dom}\mathcal{S}$ and $t \in \mathbb{T}$, there is a relation $\mathcal{M}(w,t) \subseteq \{(x,y) \in \mathsf{past}_{\mathcal{S}}(w,t) \times \mathsf{past}_{\mathcal{I}}(w,t) \mid \mathsf{match}(x,y)\}$, such that:*

1. $\mathcal{M}(w,t)$ *is a one-to-one relation;*
2. *for each* $x \in \mathsf{past}_{\mathbb{S}}^{\Delta t}(w,t)$,
   - *if* $\Box x$, *then there is* $y \in \mathsf{past}_{\mathbb{J}}(w,t)$, *such that* $(x,y) \in \mathcal{M}(w,t)$;
   - *if* $\Diamond x$, *then either there is* $y \in \mathsf{past}_{\mathbb{J}}(w,t)$, *such that* $(x,y) \in \mathcal{M}(w,t)$, *or for each* $x' \in \mathsf{past}_{\mathbb{S}}(w,t)$, *if* $x \preceq x'$, *then there is no* $y \in \mathsf{past}_{\mathbb{J}}(w,t)$, *such that* $(x',y) \in \mathcal{M}(w,t)$.
3. *for each* $y \in \mathsf{past}_{\mathbb{J}}^{\Delta t}(w,t)$, *there is* $x \in \mathsf{past}_{\mathbb{S}}(w,t)$, *such that* $(x,y) \in \mathcal{M}(w,t)$;
4. *for all* $(x,y),(x',y') \in \mathcal{M}(w,t)$, *if* $x \prec x'$, *then* $\theta_{\mathbb{J}}(y) \leq \theta_{\mathbb{J}}(y')$. $\Box$

Checking conformance to specifications with optional outputs can be done with a few modifications of the monitor described above. In *onSpecTimeout*, it should be checked whether an event x is optional (the action fails only if x is obligatory). The most difficult part is to track that all events dependent on the cancelled one are also cancelled. Assume that there is $\Delta T_{dep} \in \mathbb{T}$, such that for all $x,x' \in V_{\mathbb{S}}$, if $|\theta_{\mathbb{S}}(x) - \theta_{\mathbb{S}}(x')| > \Delta T_{dep}$, then $x \perp x'$. To describe the monitor, let us introduce a predicate $cancelled_S(x) = \big(\exists x' \in term_S \,.\, x' \preceq x\big)$ and a modified version of the primary arbiter: $arbiter_S(X) = \mathsf{min}_{\preceq}(X \setminus term_S)$.

Action 7 *onSpecOutput*$[x]$, $x \in S[t]$

Guard: *true*
Input: x
  $past_S \Leftarrow past_S \cup \{x\}$
  if $cancelled(x)$ **then**
    $term_S \Leftarrow term_S \cup \{x\}$
  else
    ...
  end if

Action 8 *onSpecTimeout*$[x]$, $x \in (past_S \setminus term_S)$

Guard: $(\theta_{\mathbb{S}}(x) + \Delta t^{+}(x)) \leq t$
Input: x
  if $\Box x$ **then**
    ...
  else
    $term_S \Leftarrow term_S \cup \{x\}$
  end if

Action 9 *onTermTimeout*$[x]$, $x \in term_S$

Guard: $\big(\theta_{\mathbb{S}}(x) + \Delta T_{dep}\big) \leq t$
Input: x
  $past_S \Leftarrow past_S \setminus \{x\}$
  $term_S \Leftarrow term_S \setminus \{x\}$

Action 10 *onInitialize*

Guard: $t = 0$
Input: $\varnothing$
  $term_S \Leftarrow \varnothing$
  ...

4 Tool Support and Experience

The proposed approach to runtime verification has been implemented in a C++ library named *C++TESK Testing ToolKit* [1]. The library provides users with classes and macros for automated development of test system components, including reference models, monitors (test oracles), stimuli generators, coverage trackers, etc. C++TESK supports testing and monitoring of both hardware and software systems but has been mainly used for hardware designs (namely, for simulation-based verification of microprocessor units). Note that hardware is usually developed in *hardware description languages* (*HDLs*), like Verilog and VHDL, and can be executed (simulated) in a special environment, called *HDL simulator*. The C++TESK facilities for developing reference models of hardware designs (and, consequently, runtime monitors) include means for sending and receiving data packages, forking and joining concurrent threads, modeling time delays and specifying order between data packages. Some of the primitives (the most important within the scope of the paper) are as follows (the syntax here differs from the original one, used in the toolkit):

- `delay(n)` — models a time delay (as an observable outcome, it increments the current time value by `n` time units);
- `recv(in):pkg` — waits until an input package is received at a given input port (`in`); then, returns that package (`pkg`);
- `send(out, pkg, opt)` — sends an output package (`pkg`) via a given output port (`out`) specifying whether the package is obligatory or optional (`opt`);
  (Note that every time a package is sent outside, it is tagged with time interval $[t-\Delta t^-, t+\Delta t^+]$, where t is the sending start time and $\Delta t^{\pm}$ are user-defined parameters of the transmission port.)
- `depends(pkg1, pkg2)` — states that an output package (`pkg1`) depends on or causally related to some other package (`pkg2`), input or output.
  (This probably answers the question raised in the beginning of Section 3 of where a specification trace, namely partial ordering of its events, is taken from.)

Differences in hardware complexity, verification purposes and amounts of resources lead to the variety of model types and model abstraction levels. Abstraction is a well-known way for fighting complexity and facilitating model development. Though the verification quality is likely to be lower in case of simpler reference models, if there is a strict deadline (and it is often so), there is no other way out. Event ordering and timing are the main subjects for abstraction in hardware designs and other concurrent time-dependent systems. We use the following classification of the reference models according to the time modeling accuracy: (1) *untimed models* (represent only general information on the cause-effect relation of their inputs and outputs, while the timing is not modelled at all: $\Delta t^{\pm} = \infty$), (2) *time-approximate models* (contain the detailed specification of the event ordering, including some internal arbitration schemes, but the timing is approximate: $\Delta t^{\pm} \leq T$, where T has a value of several tens of time units) and (3) *time-accurate models* (implement the exact, or almost exact, event ordering and timing: $\Delta t^{\pm} \leq 1$).

The proposed methodology has been used for verification of a number of units of different industrial microprocessors. Our experience was originally presented in [9], and since then we have verified a table lookup unit, an l2-cache bank controller and an instruction buffer. Also, testbenches and monitors for several previously tested components (a north bridge data switch and a memory access unit) required improvement according to the modifications of the units. The newest information of the approach application is shown in Table 1. As it can be seen from the table, the methodology supports runtime verification by means of abstract models (being available at early design stages) and, at the same time, by means of up to time-accurate models (being available typically at finishing design stages). Moreover, the approach allows reusing reference models across the hardware development cycle, which is really important in the industrial settings.

The first version of C++TESK supported only accurate reference models (it was required that a model knows the exact ordering of events on each of the output ports). Having received feedback from C++TESK users (everyone is welcome to join the community), the toolkit has been modified. Mostly, it concerns a problem of lack of unit-level specifications even for almost finished hardware designs. It is impossible to create an accurate model without detailed knowledge of the unit functionality and timing. Regular interviewing of engineers takes a lot of time and is inconvenient. Two major solutions of the problem have been proposed besides forcing the developers to write the specifications. The first solution is to reuse parts of a more complicated system-level model (emulating behavior of the whole microprocessor). Though such parts are rather abstract (as a rule, system-level models are developed in an untimed manner), they are really useful for early-stage verification. The second solution is to develop approximate reference models by means of C++TESK and to refine them if necessary.

Microprocessor Unit	Development Stage	Model Abstraction Level	
		From	To
Translation lookaside buffer	Late / finishing	Time-approximate model	Time-accurate model
Floating point unit	Late / finishing	Untimed model	—
Non-blocking L2-cache	Middle / late	Time-approximate model	—
North bridge data switch	Middle / late / finishing	Time-approximate model	Time-accurate model
Memory access unit	Early / middle	Untimed model	Time-accurate model
System interrupt controller	Early / middle	Untimed model	Time-approximate model
Table lookup unit	Late	Time-approximate model	—
L2-cache bank controller	Late	Time-accurate model	—
Instruction buffer	Late / finishing	Time-accurate model	—

Table 1: Experience of the approach application

5 Related Work

There are several works on model-based testing and monitoring that have similarities with our approach. Some of them are mentioned below.

In [7], a *partial order input/output automaton* (*POIOA*), where each transition is associated with an almost arbitrary ordered set of inputs and outputs, is used to represent the expected behavior. The key idea is to obtain two POIOAs (representing behavior of specification and implementation) and to check their conformance. There is a way to derive a test suite that guarantees fault detection defined by a POIOA-specific fault model: *missing output faults*, *unspecified output faults*, *weaker precondition faults*, *stronger precondition faults* and *transfer faults*. If the following assumptions are satisfied: an unspecified input is detectable, specified ordering of outputs can be observed, response time is bounded, and each specification transition can be modeled as a single implementation transition, then it is possible to set up conformance between two POIOAs. Comforming implementation accepts any input compatible with the specification (and may accept more) and produces outputs defined by the specification in an order compatible with the specification. If the POIOAs are not conforming, it is considered as wrong behavior of the implementation according to the fault model. The main difficulty in the approach, in our opinion, is to represent behavior of specification and implementation by the proposed formalism.

In [3], the approach to passive testing based on *invariants* is presented. Invariants are used as a means of representing the most relevant expected properties of the implementation, which should be exhibited in response to the corresponding test sequences. Two types of invariants are of usage: (1) *timed consequent invariants* and (2) *timed observational invariants*. The first type is used to check that an event happens (within certain time bounds) after a given trace of events. The second type is used to check that a given sequence of events always occur (within certain time bounds) between two given events. The correctness of the implementation behavior is verified in two steps. The first step is to check the correctness of the invariants with respect to a given specification. The second step is to check the correctness of a trace, recorded from the implementation, with respect to the invariants. We think, that this approach is applicable to monitoring of complex timed systems, but it is not clear how to maintain the sets of invariants (which might be huge) during the system life cycle.

The approach proposed in [13] allows usage of implicitly defined *asynchronous finite state machines* (*AFSMs*) for model-based testing of complex distributed systems. The implementation behavior is verified only in *quiescent states* of the FSM model. Thus, it is required that there is a predicate identifying such kind of states. The testing step is done as follows. First, all outputs are collected and their partial

order is determined. Then, all possible linearizations of the events are enumerated and checked. If all of them fail (with respect to the specification), then the implementation is not conforming to the specification. As checking is performed in quiescent states only, the approach is hardly applicable to runtime monitoring (where there may be arbitrary input sequences, and such states are rarely visited).

6 Conclusion

On-the-fly analysis of system behavior is an integral part of dynamic verification of software and hardware systems. A lot of formalisms have been proposed to express correctness properties for systems of different types, and a great number of methods have been suggested to check whether system executions are conforming to the specified properties. None of them is perfect, we think, but all together they cover a vast spectrum of verification and monitoring tasks. Among the variety of specification approaches, executable models, written in high-level programming languages, have a significant niche. First of all, such models are rather universal and allow expressing a broad range of behavioral and structural properties. Besides, programming languages (especially general-purpose languages, like C and C++) are widely spread in the engineering community.

Our work focuses on using executable models for runtime verification of reactive systems, including, in particular, time-dependent systems. The problem is not as simple as it looks at first sight. The naive checking that a system and its model produce the same outputs at the same time is inadequate in the majority of cases. The model may abstract away from many features implemented in the system under verification such as event ordering and accurate timing (at least it should be abstracted from the implementation bugs). We suppose that conformance relations used for runtime verification can be configured in several ways: (1) by introducing an independency relation over the model events, (2) by extending time points of the model outputs to time intervals and, finally, (3) by marking some of the model outputs as being optional.

Basing on this idea, we have developed a method for system monitoring and proved its correctness. The formalization is based on the theory of traces and partially ordered multisets. The method has been implemented in C++TESK, an open-source toolkit for hardware modeling, analysis and verification, and has been successfully used in about 10 projects on simulation-based verification of microprocessor units. Our future research is aiming at failure diagnostics, which is a deeper analysis of specification and implementation traces being carried out offline. The goal is to explain what in particular went wrong during the monitoring and give a hint to developers where the bugs are localized.

7 Bibliography

References

[1] *C++TESK Homepage*. Available at `http://forge.ispras.ru/projects/cpptesk-toolkit/`.

[2] R. Alur & D.L. Dill (1994): *A Theory of Timed Automata*. *Theoretical Computer Science* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[3] C. Andrés, M.G. Merayo & M. Núñez (2012): *Formal Passive Testing of Timed Systems: Theory and Tools*. *Software Testing, Verification & Reliability* 22(6), pp. 365–405, doi:10.1002/stvr.1464.

[4] H. Barringer, D. Rydeheard & K. Havelund (2007): *Rule Systems for Run-Time Monitoring: From Eagle to RuleR*. In: *Proceedings of 7th International Workshop on Runtime Verification. Revised Selected Papers*, pp. 111–125, doi:10.1007/978-3-540-77395-5_10.

[5] A. Bauer, M. Leucker & C. Schallhart (2011): *Runtime Verification for LTL and TLTL*. ACM Transactions on Software Engineering and Methodology 20(4), pp. 14:1–14:64, doi:10.1145/2000799.2000800.

[6] B. Bloom & M. Kwiatkowska (1991): *Trade-offs in True Concurrency: Pomsets and Mazurkiewicz Traces*. Technical Report TR 91-1223, Cornell University.

[7] G. von Bochmann, S.Haar, C.Jard & G.-V. Jourdan (2008): *Testing Systems Specified as Partial Order Input/Output Automata*. In: Proceedings of the 20^{th} IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems: 8^{th} International Workshop, TestCom '08 / FATES '08, Springer-Verlag, Berlin, Heidelberg, pp. 169–183, doi:10.1007/978-3-540-68524-1-13.

[8] D.V. Chieu & D.V. Hung (2012): *Timed Traces and Their Applications in Specification and Verification of Distributed Real-time Systems*. In: Proceedings of the Third Symposium on Information and Communication Technology, pp. 31–40, doi:10.1145/2350716.2350723.

[9] M. Chupilko & A. Kamkin (2011): *A TLM-Based Approach to Functional Verification of Hardware Components at Different Abstraction Levels*. In: Proceedings of the 12^{th} Latin-American Test Workshop, pp. 1–6, doi:10.1109/LATW.2011.5985902.

[10] E.M. Clarke, O. Grumberg & D.A. Peled (1999): *Model Checking*. The MIT Press.

[11] R.M. Hierons, K. Bogdanov, J.P.Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A.J.H. Simons, S. Vilkomir, M.R. Woodward & H. Zedan (2009): *Using Formal Specifications to Support Testing*. ACM Computing Surveys 41(2), pp. 9:1–9:76, doi:10.1145/1459352.1459354.

[12] V.P. Ivannikov, A.S. Kamkin, A.S. Kossatchev, V.V. Kuliamin & A.K. Petrenko (2007): *The Use of Contract Specifications for Representing Requirements and for Functional Testing of Hardware Models*. Programming and Computer Software 33(5), pp. 272–282, doi:10.1134/S0361768807050039.

[13] V. Kuliamin, A. Petrenko, N. Pakoulin, A. Kossatchev & I. Bourdonov (2003): *Integration of Functional and Timed Testing of Real-Time and Concurrent Systems*. In M. Broy & A. Zamulin, editors: Perspectives of System Informatics, Lecture Notes in Computer Science 2890, Springer Berlin Heidelberg, pp. 450–461, doi:10.1007/978-3-540-39866-0-45.

[14] W.K. Lam (2005): *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall.

[15] J. Laski & W. Stanley (2009): *Software Verification and Analysis: An Integrated, Hands-On Approach*. Springer.

[16] M. Leucker (2000): *On Model Checking Synchronised Hardware Circuits*. In: Proceedings of the 6^{th} Asian Computing Science Conference, Lecture Notes in Computer Science 1961, Springer, pp. 182–198, doi:10.1007/3-540-44464-5_14.

[17] G. Luo, R. Dssouli, G. von Bochmann, P. Venkataram & A. Ghedamsi (1993): *Generating Synchronizable Test Sequences Based On Finite State Machine with Distributed Ports*. In: Proceedings of the IFIP Sixth International Workshop on Protocol Test Systems, pp. 53–68.

[18] A. Mazurkiewicz (1987): *Trace Theory*. In: Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency, Springer-Verlag New York, Inc., New York, NY, USA, pp. 279–324, doi:10.1007/3-540-17906-2_30.

[19] B. Patel (2010): *A Monitor-Based Approach to Verification*. EE Times.

[20] V.R. Pratt (1984): *The Pomset Model of Parallel Processes: Unifying the Temporal and the Spatial*. In: Seminar on Concurrency, pp. 180–196, doi:10.1007/3-540-15670-4_9.

[21] K. Sen & G. Rosu (2003): *Generating Optimal Monitors for Extended Regular Expressions*. Electronic Notes in Theoretical Computer Science 89(2), pp. 162–181, doi:10.1016/S1571-0661(04)81051-X.

[22] B. Wile, J. Goss & W. Roesner (2005): *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann.

Top-Down and Bottom-Up Approach for Model-Based Testing of Product Lines

Stephan Weißleder
Berlin, Germany
Fraunhofer-Institute FOKUS
stephan.weissleder@fokus.fraunhofer.de

Hartmut Lackner
Berlin, Germany
Fraunhofer-Institute FOKUS
hartmut.lackner@fokus.fraunhofer.de

Systems tend to become more and more complex. This has a direct impact on system engineering processes. Two of the most important phases in these processes are requirements engineering and quality assurance. Two significant complexity drivers located in these phases are the growing number of product variants that have to be integrated into the requirements engineering and the ever growing effort for manual test design. There are modeling techniques to deal with both complexity drivers like, e.g., feature modeling and model-based test design. Their combination, however, has been seldom the focus of investigation. In this paper, we present two approaches to combine feature modeling and model-based testing as an efficient quality assurance technique for product lines. We present the corresponding difficulties and approaches to overcome them. All explanations are supported by an example of an online shop product line.

1 Introduction

Today, users of most kinds of products are not satisfied by unique standard solutions, but desire the tailoring of products to their specific needs. As a consequence, the products have to support different kinds of optional features and, thus, tend to become more and more complex. At the same time, a high level of quality is expected by the users and has to be guaranteed for all of these product variants. One example is the German car industry where each car configuration is produced only once on average. Summing up, system engineering processes often face challenges that are focused at requirements engineering for product lines and quality assurance, e.g., by testing, at the same time. This paper deals with the combination of these challenges. Today, engineering processes are supported by model-driven techniques. Models are often used to present only essential information, to allow for easy understanding, and to enable formal description and automatic processing. Models can also be used to describe the features of product lines and the test object as a basis for automatic test design. Such an approach is also used in this paper.

Product lines (multi-variant systems) are sets of products with similar features, but differences in appearance or price [19]. There are two important aspects of product lines: First, users recognize the single product variants as members of the same product line because of their resemblance. For instance, we recognize cars from a certain manufacturer or certain smart phones although we don't know internal details like, e.g., the power of the engine or the used processors. Second, the vendors of product lines cannot afford to build every product variant from scratch, but have to strive for reusing components for several product variants. The product line managers have to try to bring together these two aspects. For this, they have to know about and manage the variation points of the product line and the relation of variation points and reusable system components. Feature models can be used to express these variation points and their relations. They help in making the corresponding engineering process manageable.

A. Petrenko, H. Schlingloff (Eds.): Eighth Workshop on
Model-Based Testing (MBT 2013)
EPTCS 111, 2013, pp. 82–94, doi:10.4204/EPTCS.111.7

Quality assurance is the part of system engineering responsible for ensuring high-quality products, a positive end user experience, and the prevention of damage in safety-critical systems. Testing is an important aspect of quality assurance. Since testing is focused on several levels like, e.g., components and their integration, it can be more complex and costly than development. Because of the afore described growing complexity of systems, it is necessary to reduce the effort for testing without reducing the test quality. Model-based test design automation is a powerful approach to reach this goal. It can be used to automatically derive test cases from models. There are several experience reports to substantiate the success of this technique [6, 9].

In this paper, we present two approaches to apply automatic model-based test design for the quality assurance of product lines. All descriptions are supported by an online shop example, i.e., a product line of online shops. This paper is structured as follows. In the next section, we describe the example and use it to introduce feature modeling and automatic model-based test design. In Section 3, we present the two approaches for model-based testing of product lines together with an evaluation of their advantages and challenges. In this paper, we focus on theoretical considerations instead of applying complete tool chains. Some parts of the projected tool chain, however, can already be used and were applied for our example. Section 4 contains the related work. In Section 5, we summarize, discuss threats to validity, and present our intended future work.

2 Fundamentals

In this section, we define an online shop product line as our running example. For this example, we assume that we are a provider of online shops. We offer to install and maintain online shops with different capabilities. The price depends on the supported set of capabilities. All of our shops include a catalog that lists all the available products and at least one payment method. In our example, we allow payment via bank transfer, with ecoins, and by credit card. The shops can have either a high or a low security level that determine the security of communication. For instance, using a credit card requires a high security level. Furthermore, we offer comfort search functions in the shop to support product selection. Our customers can select a subset of these features for integration into their specific product.

In the following, we use this example to introduce feature models for describing the features, i.e., the variation points of a product line and their relations. Furthermore, we also use it to introduce state machine models and how to use them for automatic model-based test design. Finally, we show how to link elements of feature models to elements of other kinds of models.

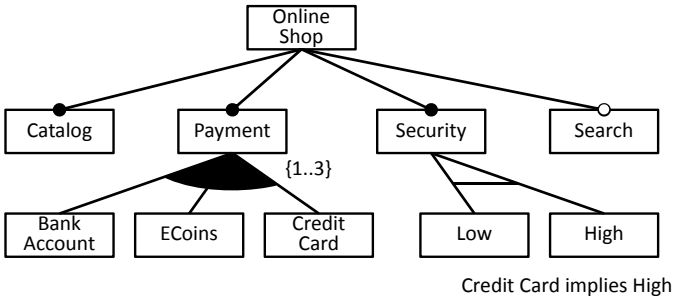


Figure 1: Feature model for online shops.

2.1 Feature Models

Models are used as directed abstractions of all kinds of artifacts. The selection of the contained information is only driven by the model's intended use. Thus, models are often used to reduce complexity and support the user in understanding the described content. In the following, we present feature models that help in describing all the aforementioned features of our online shop product line.

A feature model is a tree structure of features depicted as rectangles and relations between them depicted as arcs that connect the rectangles. Figure 1 depicts a feature model that contains information about our online shops. The topmost feature contains the name of the product line. Four features are connected to it: The features *Catalog*, *Payment*, and *Security* are connected to the top-most feature by arcs with filled circles at their end, which describe that these three features are mandatory, i.e., exist in every product variant. The *Search* feature is optional, which is depicted by using an arc that ends with an empty circle. This hierarchy of features is continued. For instance, the feature *Payment* contains the three subfeatures *Bank Account*, *ECoins*, and *Credit Card*, from which at least one has to be selected for each product variant. The subfeatures *High* and *Low* of the feature *Security* are alternative, which means that exactly one of them has to be chosen for each product variant. Furthermore, there is a textual condition that states that credit cards can only be selected if the provided security level is high.

Summing up, feature models are a simple way to describe the variation points of a product line and their relations at an abstract level that is easy to understand. Their semantics, however, only consist of rectangles and arcs with no links to system engineering-relevant aspects such as requirements or architecture models. The importance of feature models for the system engineering process only becomes real if they are integrated into the existing tool chain. This integration is done by linking the features of the feature model to other artifacts like, e.g., requirements in DOORS [8]. There also exists corresponding tool support [7, 18]. In our approach, we link features to elements of state machines of the Unified Modeling Language (UML) [13] to steer automatic model-based test design for product lines.

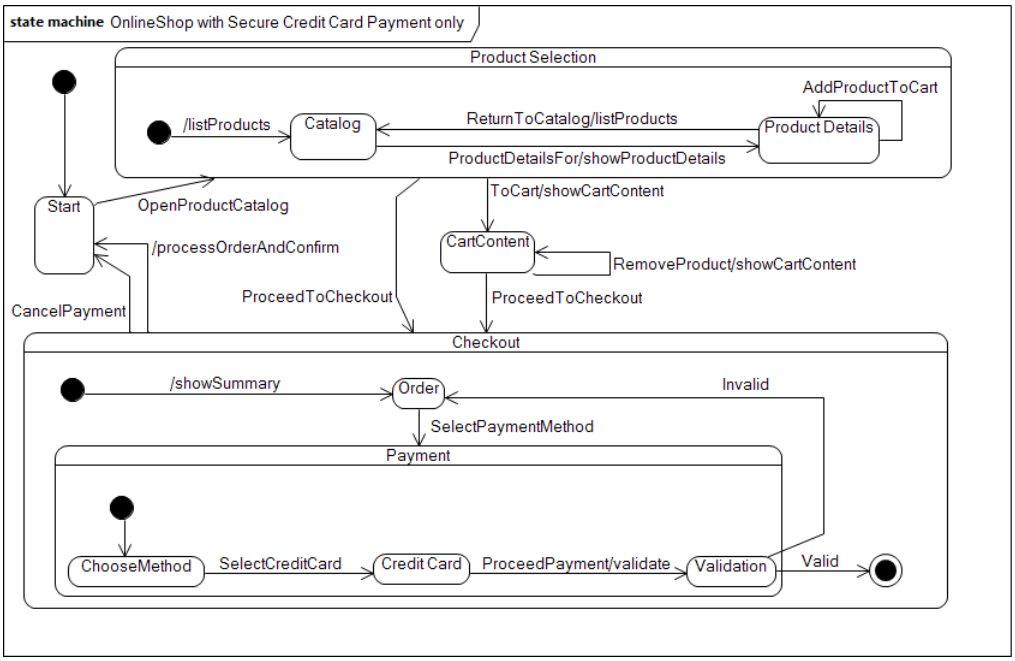


Figure 2: Online shop state machine diagram for one product variant.

2.2 Automatic Model-Based Test Design

Models can also be used for testing. The corresponding technique is called model-based testing (MBT) and there are many different approaches to it [21]. Several kinds of models are applicable for MBT like, e.g. system models or environment models [24]. Furthermore, different modeling paradigms are applicable like, e.g., state charts, petri nets, or classification trees.

For our online shop system, we focus on the automatic derivation of test cases based on structural coverage criteria that are applied to state machines. Figure 2 shows such a state machine. The behavior depicted in this state machine corresponds to one product variant of our online shop product line that only allows to pay per credit card and does not include the search function: A user of the online shop can open the product catalog (*OpenProductCatalog*). In this state, the user can select products and have a look at their details (*ProductDetailsFor*). In the detail view, the user can decide to add the product to his shopping cart (*AddProductToCart*). After the user selected products, he can decide to remove some selected elements again (*ToCart*, *RemoveProduct*) or to finish the transaction (*ProceedToCheckOut*). For paying, the user first has to select a payment method (*SelectPaymentMethod*). For the depicted shop variant, the user is only allowed to select the credit card payment method (*SelectCreditCard*). Afterwards, the system validates the entered user data and if they are valid (*Valid*), then the order is processed and a confirmation message is shown to the user. Finally, the user is forwarded to the initial page of the shop. Like depicted in the state machine, the user has the option to cancel the process and return to the start page during the checkout process.

This model can be used for automatic test design. As stated above, there are several ways to do so. A widely used approach is to apply coverage criteria like, e.g., All-Transitions [20] to the state machine. A test generator then tries to create paths on the state machine that cover all transitions of the state machine. These paths can be executed by using the sequence of triggers that correspond to the path transition sequence. Afterwards, the created paths are translated into test cases of the desired target language that can be used, e.g., for documentation or test execution. There are several automatic model-based test generators available like, e.g., the Conformiq Designer [5] or ParTeG [22].

Automatic model-based test design for single product variants is well-known. In this paper, we do focus on how to use this technique for product lines.

2.3 Linking Feature Models and Models for Test Generation

In order to apply model-based test generation to product lines, the model for test generation has to be linked to the feature model. One straight forward approach is to also describe all variation points in the state machine, i.e., the possible behavior of the system under test, and to link the features of the feature model to these variation points. Figure 3 depicts a model that contains the behavior of all product variants. Because more than one variant is described, such models are called 150% models. As one can see, the depicted state machine contains elements that correspond to the aforementioned variation points of the online shop product line. However, it is not a complete model of our system as it lacks the information from the feature model like, e.g., the relations of the features and the corresponding information about the validity of feature selections. To resolve this, we connect the features of the feature model to the 150% state machine with logical expressions.

Mapping features to other model elements can require complex logical expressions and, thus, can become complex. For reasons of simplicity, we link the models by a mapping model that links features of the feature model to one or more model elements of the 150% model. The application of a configuration to the state machine results in a 100% model by deleting all model elements that are not associated to

that particular configuration. Figure 4 depicts the mapping of the feature model to the 150% model. Using this mapping, it is possible to select valid variants or sets of them, to derive corresponding 100% state machines, and to automatically derive test cases from them. As described in the related work in Section 4, there are already some approaches that head into the same direction. For instance, the product-by-product approach creates all valid product variants, derives the corresponding 100% models, and applies model-based test generation to each model. However, this approach corresponds to a brute force approach, which is infeasible for larger systems. There are several approaches of how to design test cases for such linked models. In the following, we present and compare two more mature test design approaches for product lines.

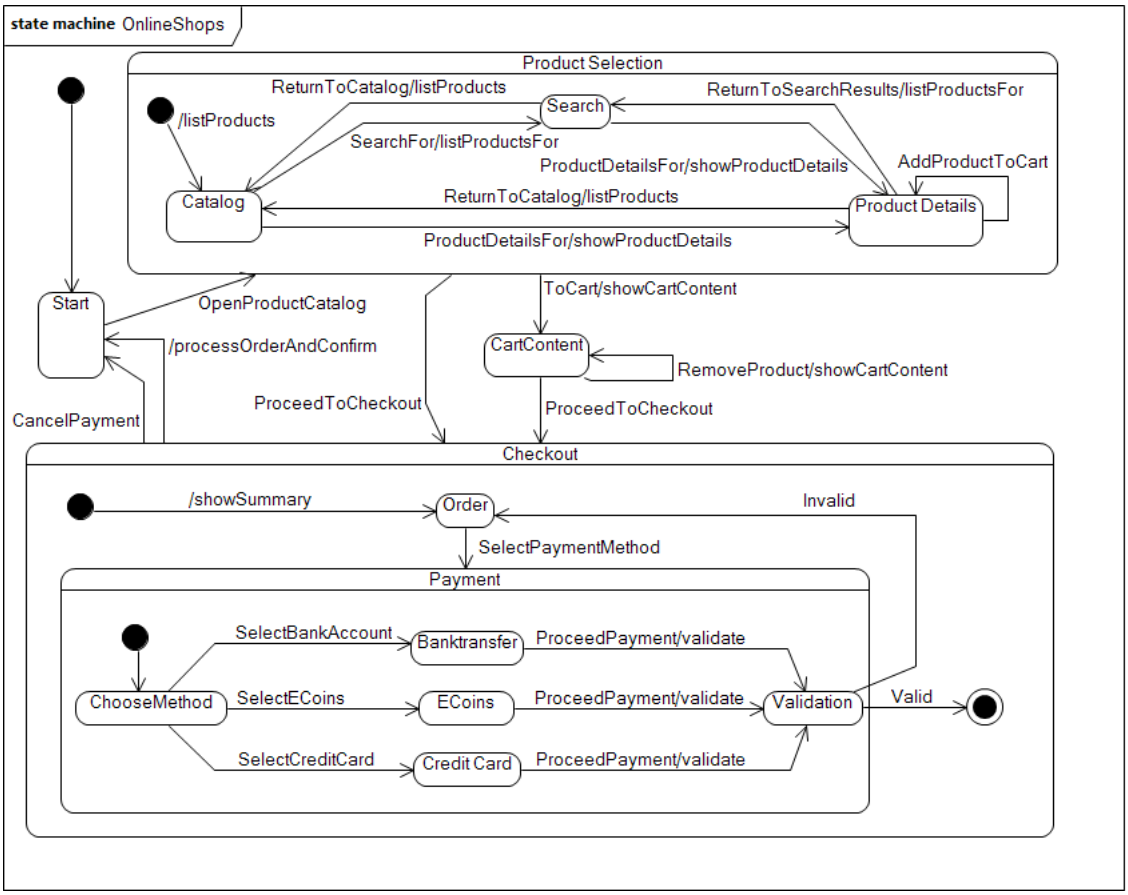


Figure 3: Online shop 150% state machine.

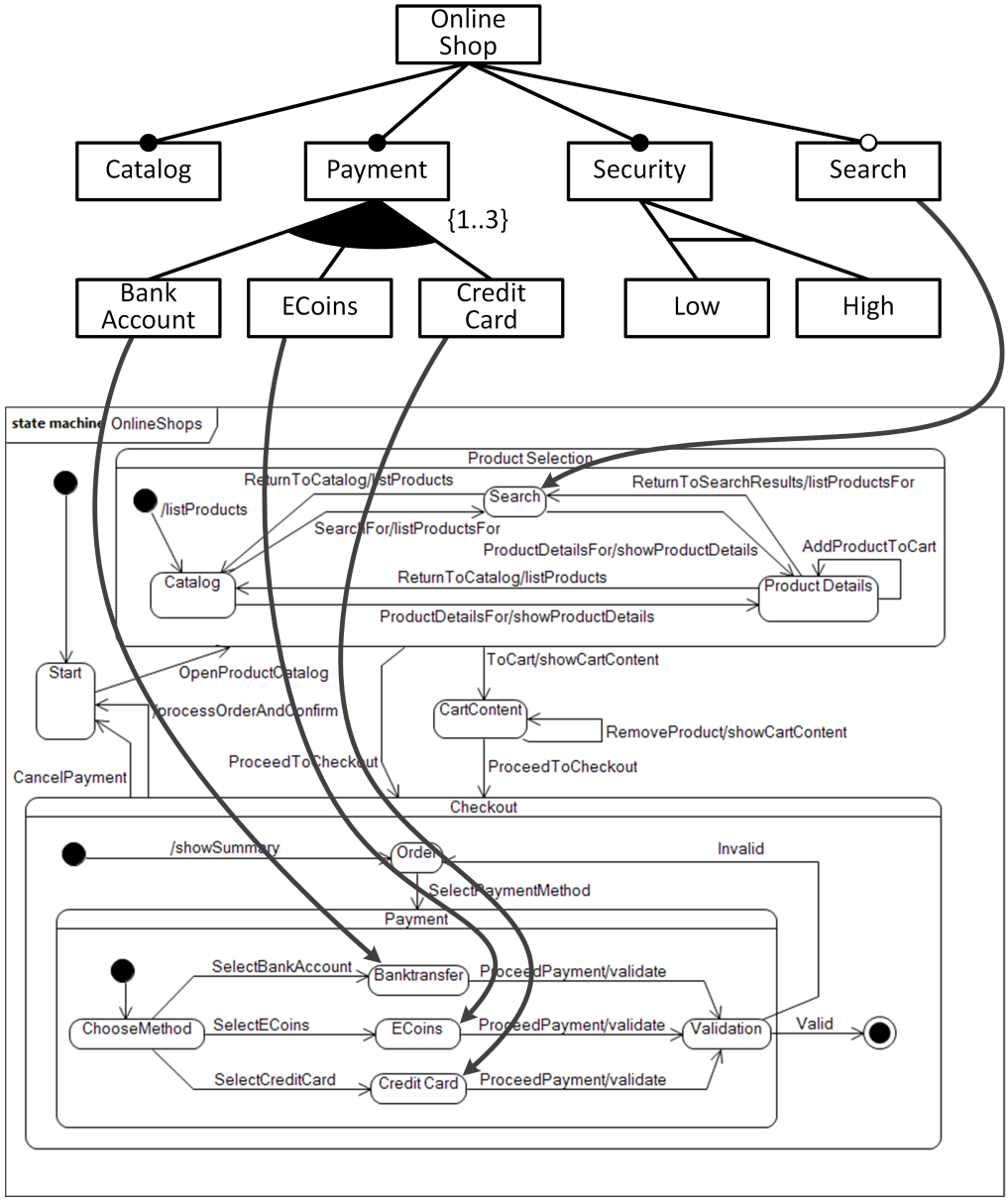


Figure 4: Mapping the feature model to the 150% state machine.

3 Applying Model-Based Test Design to Product Lines

In the previous section, we described how to use feature models to describe variation points of product lines, how to use state machines for automatically designing test cases, and how to link feature models and state machines. By this, we provided the infrastructure for automatic model-based test design for product lines. There are, however, several processes of actually deriving test cases from the combination of these two kinds of models. In the following, we are going to present two different approaches and to evaluate their pros and cons using the described example.

3.1 Top-Down Approach

In the top-down approach, we first derive a set of product variants from the feature model, derive the set of corresponding 100% models, and apply standard model-based testing to each 100% model. Automatic model-based test generation is often driven by applying coverage criteria to models. This approach as presented for state machines can also be applied to feature models. The coverage criteria are used to measure to which degree the product variants represent the product line, i.e., the set of all possible product variants. They can also be used to automatically derive a representative set of product variants [16]. Using the links between feature model and 150% state machine allows for automatic derivation of the corresponding 100% state machines for each generated product variant. For each of these 100% state machines, the presented standard approach of test generation based on structural coverage criteria can be applied. Afterwards, the generated test suites can be executed for the corresponding product variants. Figure 5 depicts this approach.

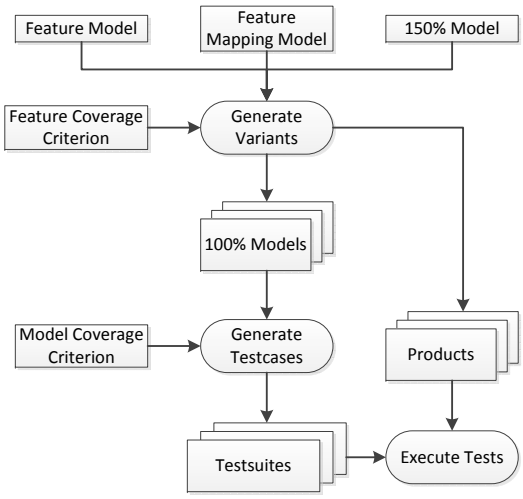


Figure 5: Top-down approach for test generation.

For evaluating the strengths and weaknesses of the top-down approach, we consider two aspects: **a)** To which degree and with which efficiency are product variants covered? **b)** To which degree and with which efficiency is the system behavior covered? For **a**, the coverage criteria applied to the feature model directly determine the coverage of the feature model. The answer to **b** additionally depends on the relative strength of the coverage criterion that is applied to each 100% model. Furthermore, there will be overlap between the behavior of the variants, which is going to be tested twice. So the presumption

is that the generated test cases will not be very efficient, i.e., several parts will be tested twice without additional gain of information.

For our example of the online shop product line, we run test generation with the following combination of coverage criteria: We apply the two coverage criteria *All-Features-Selected* and *All-Features-Unselected* to the feature model to derive variants. Then, we derive the corresponding 100% state machines using the mapping from the feature model to the 150% state machine and generate tests using the coverage criterion *All-Transitions* [20] on every generated 100% state machine. To the best of our knowledge, the research on coverage criteria on feature models is still in an immature state and, thus, references to such coverage criteria are rare [10]. In contrast to existing work on coverage criteria, it is also important to focus on covering features by not selecting them. The mentioned two coverage criteria are correspondingly focused on selecting and not selecting all features of a feature model, respectively. The two coverage criteria *All-Features-Selected* and *All-Features-Unselected* can be satisfied by two product variants in which the following optional features are selected: **(i)** Credit Card Payment and High Security; **(ii)** Bank Transfer, ECoins, Low Security, and Search. Figures 6 and 7 show the corresponding variant models.

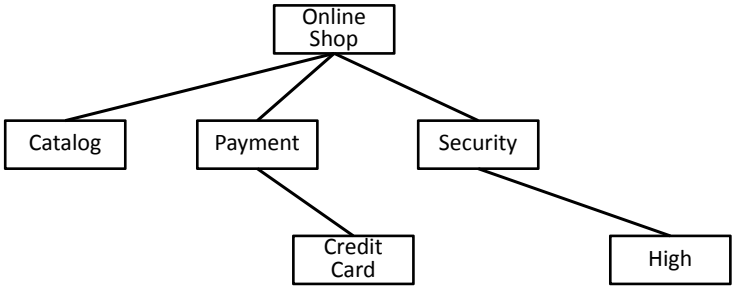


Figure 6: Product variant (i).

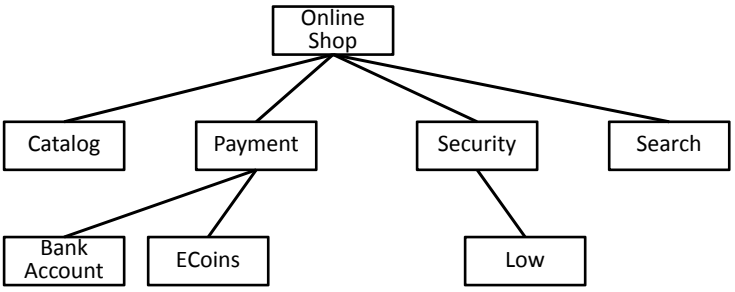


Figure 7: Product variant (ii).

For each variant, one test case is enough to cover all transitions of the corresponding 100% state machine.

For **(i)**, the sequence of events to trigger the test case is as follows (see 100% model in Figure 2): *OpenProductCatalog; ProductDetailsFor; AddProductToCart; AddProductToCart; ReturnToCatalog; ToCart; RemoveProduct; ProceedToCheckout; CancelPayment; OpenProductCatlog; ProceedToCheckout; SelectPaymentMethod; SelectCreditCard; ProceedPayment; Invalid; SelectPaymentMethod; SelectCreditCard; ProceedPayment; Valid.*

For **(ii)**, the sequence is as follows (no corresponding 100% model depicted - please refer to the corresponding parts of the 150% model in Figure 3):

OpenProductCatalog; SearchFor; ProductDetailsFor; AddProductToCart; AddProductToCart; ReturnToSearch; ReturnToCatalog; ToCart; RemoveProduct; ProceedToCheckout; CancelPayment; OpenProductCatalog; ProductDetailsFor; AddProductToCart; ReturnToCatalog; ProceedToCheckout; SelectPaymentMethod; SelectBankAccount; ProceedPayment; Invalid, SelectPaymentMethod; SelectECoins; ProceedPayment; Valid.

All features have been selected as well as deselected. Both test cases together cover all 22 explicitly triggered transitions of the 150% model. They have a total length of (i:19 + ii:24) 43 event calls, which is almost twice the size of the lower boundary. Since we created two product variants for testing instead of the 20 possible ones, however, this approach is still far more efficient than the brute force approach.

Our tool chain that supports the described test generation approach is currently under construction. However, we already manually created the two 100% state machines and used the Conformiq Designer [5] to automatically design tests. The used coverage criterion is All-Transitions. For variant **i**, the test generator created seven test cases that comprise altogether 28 event calls. In case of variant **ii**, eleven test cases with 42 event calls were generated. Since the Conformiq Designer is not tailored to find as many test steps per test case as possible, this deviation from our theoretical considerations are no surprise. After all, all transitions were covered for both cases.

3.2 Bottom-Up Approach

The idea of the bottom-up approach is contrary to the top-down approach. Here, we create test cases based on the 150% state machine and match the resulting sequences to single product variants, afterwards. The idea is simple, but the generated paths of the state machine cover elements that are linked to different features and the state machine does not provide means to check if all these features can be combined in one valid product variant. As a result, one has to include the conditions that are expressed in the feature model into the 150% state machine. This is done by expressing the selection and deselection of a feature to a variable with the value 1 and 0, respectively. Figure 8 depicts such an enrichment on

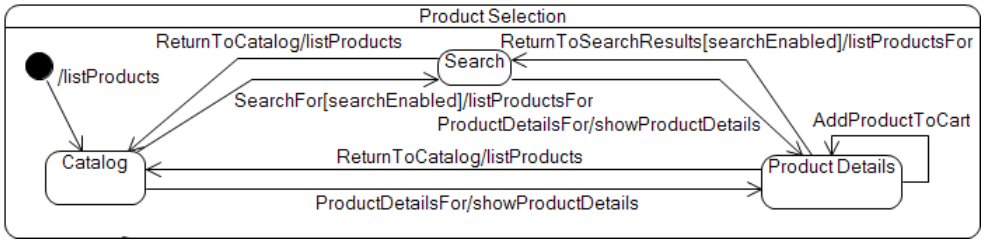


Figure 8: Enriched part of the 150% state machine for automatic test generation of only valid product variants.

an excerpt of the 150% state machine. The shown composite state was enriched with information from the feature model by adding a guard to all transitions leading to the state *Search*, which corresponds to the search feature. Now, the tests cover the search function only if the corresponding guard is set to true. Setting the guard variable to a value is possible only once at the beginning of the state machine. As a consequence, the variable and the feature selection will be consistent for the whole test case. Relations between features can also be expressed in the guard, e.g., by stating that the value of the variable corresponding to an alternative feature has a different value.

This enables the generator to choose from any valid configuration for finding a new test case. Since we did not generate the product variants from the feature model, we had to retrieve the necessary product

variants for test execution from the test cases. Conformiq supports this task because all initial variable values can be stored into the prolog of a test case. Figure 9 shows the workflow for the bottom-up approach.

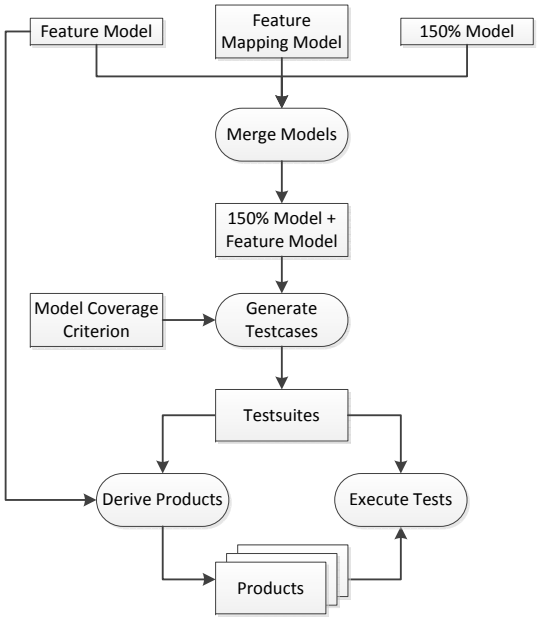


Figure 9: Bottom-up approach for test generation.

In the following, we present the input sequence for a test case that covers all transitions in the enriched 150% model:
OpenProductCatalog, ProductDetailsFor, ReturnToCatalog, SearchFor, ProductDetailsFor, AddProductToCart, AddProductToCart, ReturnToSearchResults, ReturnToCatalog, ToCart, RemoveProduct, ProceedToCheckout, SelectPaymentMethod, SelectBankAccout, ProceedPayment, Invalid, SelectPaymentMethod, SelectECoins, ProceedPayment, Invalid, CancelPayment, OpenProductCatalog, ProceedToCheckout, SelectPaymentMethod, SelectCreditCard, ProceedPayment, Valid.
This test case has 27 test steps, covers all 22 event calls of the 150% model, and requires only one product variant for test execution in which all features except *Low* (Security) have been selected.

Again, we use the Conformiq Designer for test generation focused on covering all transitions. The result of the test generation are twelve test cases with overall 59 event calls.

3.3 Comparison

Here, we summarize the first evaluations of both approaches and compare them to each other.

Concerning our theoretical considerations and the manually created test cases, the top-down approach results in two test cases that use 43 event calls and the bottom-up approach results in one test case with 27 event calls. From the perspective of redundancy, the bottom-up approach seems to be more efficient than the top-down approach. On the one hand, this always depends on the used coverage criteria for the feature model. For instance, a weaker coverage criterion that is satisfied by only one product variant, can lead to a more efficient result for the top-down approach. On the other hand, this is no generally applicable solution because the importance of single variants for the behavior is not easy to determine. The bottom-

up approach abstracts from this issue because one does not have to define the coverage criterion on the feature model in the first place, but it does not necessarily cover all features of the feature model. As a result, it seems that the personal notion and the importance of the covered aspects is important: If feature coverage is important, the top-down approach is more suitable. If efficiency and covered behavior is more important, the bottom-up approach is more suitable.

The application of the Conformiq Designer shows partly similar though different results. For the two 100% models, 18 test cases with 70 event calls were generated. For the 150% model, twelve test cases with 59 event calls were generated. The main reason for the deviation to the manually created test cases is the breadth-first search approach of Conformiq and our approach of finding the minimal number of test cases. Furthermore, the Conformiq Designer created test cases for two product variants for the bottom-up approach. Interestingly, both variants include high security and differ only in the selection of the feature *Credit Card*. This distinction is unnecessary and would have been avoided by a human designer. This issue leaves room for future improvements.

4 Related Work

In this section, we present the related work. We present standard approaches to model-based testing, cite work about feature modeling, and name approaches to combining both.

Testing is one of the most important quality assurance techniques in industry. Since testing often consumes a high percentage of project budget, there are approaches to automate repeating activities like, e.g., regression tests. Some of these approaches are data-driven testing, keyword-driven testing, and model-based testing. There are many books that provide surveys of conventional standard testing [1,2,12] and model-based testing [3,20,25]. In this paper, we use model-based testing techniques and apply them to product lines. Modeling languages like the Unified Modeling Language (UML) [13] have been often used to create test models for testing. For instance, Abdurazik and Offutt [14] automatically generate test cases from state machines. We also apply state machines of the UML.

Feature models are commonly used to describe the variation points in product lines. There are several approaches to apply feature models in quality assurance. For instance, Olimpiew and Gomaa [15] deal with test generation from product lines and sequence diagrams. In contrast to that, we focus on UML state machines and describe different approaches for combining both. In contrast to sequence diagrams, state machines are commonly used to describe a higher number of possible behaviors, which make the combination with feature models more complex than combining feature models and sequence diagrams. As another example, McGregor [11] shows the importance of a well-defined testing software product line process. Just like McGregor, the focus of our paper is only investigating the process of actually creating tests rather than defining the structural possible relations of feature models and state machines. Pohl and Metzger [17] emphasize the preservation of variability in test artifacts of software product line testing. As we derive test case design from models automatically, this variability is preserved. Lochau et al. [10] also focus on test design with feature models. In contrast to our work, they focus on defining and evaluating coverage criteria that can be applied to feature models. In the presented top-down approach, we strive for using such coverage criteria on feature models for the automation of test design. Cichos et al. [4] also worked on an approach similar to the presented bottom-up approach. Their approach, however, requires that the used test generator has to be provided a set of product variants to derive 100% models from the 150% model for automatic test generation. As a consequence, the test generator requires an additional input parameter and (as the authors state) no standard test generator can be applied for their approach. In contrast, both of our approaches allow for integrating commercial off-the-shelf test generators like in

our case, Conformiq [5]. One of the most important aspects in our work is the ability to integrate our approach into existing tool chains. In [23], we already addressed model-based test generation for product lines. However, back then we focused on reusing state machines in multi-variant environments instead of describing the different automatic test design approaches for product lines.

5 Summary, Discussion, and Future Work

In this paper, we presented different approaches to the automatic test design for product lines. We described the state of the art, presented the general idea of linking feature models to other system artifacts, and presented two approaches to use this linking for automatic test design. Our main contributions are the definition and comparison of the presented approaches using a small example.

The presented outcomes are theoretical considerations and first test generation results using the Conformiq Designer. Some steps of the proposed tool chains are still under construction and the corresponding intermediate results were, thus, partly created manually. If the missing parts of the tool chains will be developed, we will be able to run larger case studies for the comparison of both approaches automatically. Furthermore, there are more approaches than the presented ones of automatically designing tests for product lines that were not evaluated here. To name just one example, single steps in our automatic tool chain could also be replaced by manual experience-based steps. Another point to discuss is the degree of reusability in the source code. As mentioned in the beginning of this paper, reusing system components is an important aspect in managing product variants engineering for product lines. If the components, however, were not reused adequately and copy&paste was applied, instead, then covering the behavior at the source code level for one product variant does not necessarily imply covering the very same behavior at the source code level for another product variant. A solution to this issue would be to also integrate the relations from features in the feature model to variation points in the source code.

In the near future, we plan to finish the development of both proposed tool chains. The tool chains are intended to provide the glue between existing tools for feature modeling like, e.g., pure::variants [18] or the FeatureMapper [7], and automatic test generators like, e.g., the Conformiq Designer [5] or ParTeG [22]. Besides the comparison of the two approaches, the single approaches contain enough room for further investigations. For instance, one interesting question for the top-down approach is if it is advisable to apply strong coverage criteria on the feature model and weak ones on the 100% models or vice versa. For the bottom-up approach, an interesting task is to retrieve a minimal number of product variants from test cases generated from the 150% model. As stated above, we also plan to run further experiments to evaluate the pros and cons of the presented approaches.

References

[1] Paul Ammann & Jeff Offutt (2008): *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, doi:10.1017/CBO9780511809163.

[2] Robert V. Binder (1999): *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Manfred Broy, Bengt Jonsson & Joost P. Katoen (2005): *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer, doi:10.1007/b137241.

[4] H. Cichos, S. Oster, M. Lochau & A. Schürr (2011): *Model-based Coverage-Driven Test Suite Generation for Software Product Lines*. In: *Proceedings of the ACM/IEEE 14th International Conference on Model Driven*

Engineering Languages and Systems, *Lecture Notes in Computer Science (LNCS)* 6981, Springer Verlag, Heidelberg, pp. 425–439, doi:10.1007/978-3-642-24485-8_31.

[5] Conformiq: *Designer 4.4*. http://www.conformiq.com/.

[6] Inc. Forrester Research (2012): *The Total Economic Impact of Conformiq Tool Suite*. http://www.conformiq.com/tei-conformiq.pdf.

[7] Florian Heidenreich (2012): *FeatureMapper*. `http://featuremapper.org/`.

[8] IBM: *Rational DOORS*. www.ibm.com/software/products/us/en/ratidoor.

[9] Hartmut Lackner, Jaroslav Svacina, Stephan Weißleder, Mirko Aigner & Marina Kresse (2010): *Introducing Model-Based Testing in Industrial Context - An Experience Report*. In: *MoTiP'10: Workshop on Model-Based Testing in Practice*.

[10] Malte Lochau, Sebastian Oster, Ursula Goltz & Andy Schürr (2012): *Model-based pairwise testing for feature interaction coverage in software product line engineering*. *Software Quality Journal* 20(3-4), pp. 567–604, doi:10.1007/s11219-011-9165-4.

[11] John D. McGregor (2001): *Testing a Software Product Line*. *Technical Report CMU/SEI-2001-TR-022*.

[12] Glenford J. Myers (1979): *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.

[13] Object Management Group (2011): *Unified Modeling Language (UML), version 2.4*. `http://www.uml.org`.

[14] Jeff Offutt & Aynur Abdurazik (1999): *Generating Tests from UML Specifications*. In Robert France & Bernhard Rumpe, editors: *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, 1723, Springer, pp. 416–429, doi:10.1007/3-540-46852-8_30.

[15] Erika Mir Olimpiew & Hassan Gomaa (2005): *Model-Based Testing for Applications Derived from Software Product Lines*. *ACM SIGSOFT Software Engineering Notes* 30(4), pp. 1–7, doi:10.1145/1082983.1083279.

[16] Sebastian Oster, Ivan Zorcic, Florian Markert & Malte Lochau (2011): *MoSo-PoLiTe: tool support for pairwise and model-based software product line testing*. In: *VaMoS*, pp. 79–82. Available at `http://doi.acm.org/10.1145/1944892.1944901`.

[17] Klaus Pohl & Andreas Metzger (2006): *Software Product Line Testing*. *Communications of the ACM* 49(12), pp. 78–81, doi:10.1145/1183236.1183271.

[18] pure systems (2012): *pure::variants*. `http://www.pure-systems.com`.

[19] Carnegie Mellon University (2012): *Software Product Lines*. `http://www.sei.cmu.edu/productlines/`.

[20] Mark Utting & Bruno Legeard (2006): *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[21] Mark Utting, Alexander Pretschner & Bruno Legeard (2012): *A Taxonomy of Model-Based Testing Approaches*. *Softw. Test. Verif. Reliab.* 22(5), pp. 297–312, doi:10.1002/stvr.456.

[22] Stephan Weißleder: *ParTeG (Partition Test Generator)*. http://parteg.sourceforge.net.

[23] Stephan Weißleder, Dehla Sokenou & Holger Schlingloff (2008): *Reusing State Machines for Automatic Test Generation in Product Lines*. In Axel Rennoch Thomas Bauer, Hajo Eichler, editor: *Model-Based Testing in Practice (MoTiP)*, Fraunhofer IRB Verlag.

[24] Stephan Weileder & Hartmut Lackner (2010): *System Models vs. Test Models -Distinguishing the Undistinguishable?* In Klaus-Peter Fhnrich & Bogdan Franczyk, editors: *GI Jahrestagung (2)*, *LNI* 176, GI, pp. 321–326. Available at `http://dblp.uni-trier.de/db/conf/gi/gi2010-2.html#WeisslederL10`.

[25] Editors Justyna Zander, Ina Schieferdecker & Pieter J. Mosterman (2011): *Model-Based Testing for Embedded Systems*. CRC Press.