Habilitationsschrift

Model-Based Mutation Testing:
Theory and Application

Bernhard K. Aichernig

January 5, 2012
Abstract

In this habilitation thesis we present our research on model-based mutation testing. Mutation testing is a way of assessing and improving a test suite by checking if its test cases can detect a number of injected faults in a program. The faults are introduced by syntactically changing the source code of a program.

In our work we generalise mutation testing from program testing to model-based testing. Like in model-based testing the aim is to use the model for both, generating test cases and as a test oracle. Hence, we automatically generate test cases from a test model and test if a system under test conforms to it. In contrast to classical model-based testing, only those test cases are generated that find the injected faults in a set of mutated models. The generated tests are then executed on the system under test and will detect if a mutated model has been implemented. Hence, model-based mutation testing rather tests against non-conformance than for conformance. In terms of epistemology, we are rather aiming for falsification than for verification. It is a complementary fault-centred testing approach.

Our research has led to a series of research results that range from theory via implementation to application. The theoretical investigations are based on formal semantics of models, conformance, and test cases. We showed that the search for mutation test cases can be generalised to a non-conformance problem. The result is a theory of mutation testing that also applies to non-deterministic models. We have presented this theory in the semantic styles of Back’s Refinement Calculus and in Hoare and He’s Unifying Theories of Programming.

We implemented these theoretical results for different modelling styles: relational specifications, process algebras, coordination languages, Action Systems, and Qualitative Reasoning models. The underlying techniques for implementing our test case generators for this variety of models include constraint solvers, SMT solvers, bi-simulation checkers, symbolic execution, concolic execution, term rewriting, input-output conformance checkers, and Qualitative Reasoning simulators.

The thesis is composed of an introductory chapter and 21 selected publications. The papers are organised according to the application domain of our model-based mutation testing technique: we generated tests for contracts, communication protocols, embedded systems, and hybrid systems.
Contents

1 Introduction ................................................. 1
   1.1 Thesis .............................................. 1
   1.2 Model-Based Testing ..................................... 3
   1.3 Mutation Testing ...................................... 5
   1.4 Formal Testing ..................................... 6
      1.4.1 Formal Models ................................ 7
      1.4.2 Conformance ................................ 10
      1.4.3 Testing Assumptions ......................... 15
   1.5 The Role of Test Cases as Specifications ................. 17
   1.6 Model-Based Mutation Testing ......................... 18
      1.6.1 Mutation of Models ........................... 18
      1.6.2 Contracts .................................. 18
      1.6.3 Communication Protocols .................... 20
      1.6.4 Embedded Systems ............................ 25
      1.6.5 Hybrid Systems .............................. 26
   1.7 Concluding Remarks .................................. 28
      1.7.1 Summary ................................... 28
      1.7.2 Discussion ................................. 29
      1.7.3 Future Work ................................ 30
   Bibliography ........................................... 32
      Publications Forming the Habilitation Thesis ....... 32
      Other Related Publications of the Author .......... 34
      Related Literature ................................ 38

2 Publications on the Theory of Testing .................. 45
   2.1 Mutation Testing in the Refinement Calculus 1 .......... 47
   2.2 Mutation Testing in UTP 2 ......................... 63
   2.3 Unifying Input Output Conformance 3 ................ 95
   2.4 When Model-Based Testing Fails 4 ................... 117

3 Publications on Contracts ................................. 131
   3.1 Test Case Generation by OCL Mutation and ........... 133
      Constraint Solving 5 ............................
   3.2 Test Case Generation by Contract Mutation in Spec# 6 ... 141
# 4 Publications on Communication Protocols

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>From Faults Via Test Purposes to Test Cases</td>
<td>159</td>
</tr>
<tr>
<td>Protocol Conformance Testing a SIP Registrar</td>
<td>175</td>
</tr>
<tr>
<td>Test Purpose Generation in an Industrial Application</td>
<td>185</td>
</tr>
<tr>
<td>Improving Fault-Based Conformance Testing</td>
<td>197</td>
</tr>
<tr>
<td>Fault-Based Conformance Testing in Practice</td>
<td>213</td>
</tr>
<tr>
<td>When BDDs Fail</td>
<td>253</td>
</tr>
<tr>
<td>Dynamic Symbolic Execution for Testing Distributed</td>
<td></td>
</tr>
<tr>
<td>Connectors as Designs</td>
<td></td>
</tr>
</tbody>
</table>

# 5 Publications on Embedded Systems

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping UML to Labeled Transition Systems for Test-Case Generation</td>
<td>305</td>
</tr>
<tr>
<td>UML in Action: A Two-Layered Interpretation for Testing</td>
<td>327</td>
</tr>
<tr>
<td>Efficient Mutation Killers in Action</td>
<td>335</td>
</tr>
</tbody>
</table>

# 6 Publications on Hybrid Systems

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conformance Testing of Hybrid Systems with Qualitative Reasoning</td>
<td>347</td>
</tr>
<tr>
<td>Qualitative Action Systems</td>
<td>365</td>
</tr>
<tr>
<td>Model-Based Mutation Testing of Hybrid Systems</td>
<td>385</td>
</tr>
<tr>
<td>Automated Conformance Verification of Hybrid Systems</td>
<td>407</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Thesis

Is testing able to show the absence of bugs? The most prominent negative answer was given by the late Edsger Dijkstra: “Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.” [Dij72]. Dijkstra was always motivating the need for formally verified software. Of course, in general Dijkstra is right, in the same way as Popper was right, when he stated that we can never verify that a theory is correct by a finite set of experiments. In principle, only refutation (falsification) is possible [Pop95]. However, this should not lead to an over-pessimistic judgement rejecting testing completely. This would be futile, since testing is the only way of building trust in a running system embedded in a complex environment. Testing is needed to check our assumptions. With wrong assumptions, even formally verified software may fail.

A famous example of such a rare and subtle software bug was found in the binary search algorithm implemented in the Java JDK 1.5 library in 2006 [Blo06]. For large arrays the binary search method threw an exception raised by accessing the array out of its boundaries. The fault was in the line responsible for calculating the next element in the search following the divide-and-conquer strategy: int mid = (lo + hi) / 2; From an algorithmic point of view this line is perfectly fine, assuming idealised infinite integers. However, this assumption is wrong in the case of a concrete computer with bounded integer ranges. For large lo and hi values the sum would create an overflow leading to a negative value of variable mid. The developer of this method reported that he actually took the algorithm including this line from the famous book Programming Pearls [Ben96]. The fault resided in Sun’s Java library for nine years, before it was found. This code line existed for two decades in the algorithm book without anybody noticing the problem. The algorithm was even proved correct. So what was the nature of this problem? The answer is that the fault was introduced by assuming a wrong background theory on numbers. In the domain of mathematics the algorithm works perfectly fine, assuming infinite integers. In the domain of Java with integer overflows it is wrong. The correctness proof relied on the wrong assumption and therefore could not detect the problem.
This example shows that we have to keep in mind that program proof is about proving a formula, model checking is about checking a model, but only testing is targeting the running system in its real environment.

Gaudel showed that testing can be formal too [Gau95], and even one of the most prominent figures in computer science, Tony Hoare, has changed his view: "I have radically changed my attitude towards program testing which I now understand to be entirely complementary to scientific design and verification methods, and makes an equal contribution to the development of reliable software on industrial scale." [Hoa85]

In this habilitation thesis we present our work on formal testing that reconsiders the basic question of testing, if testing can show the absence of bugs. Our central thesis of this habilitation is that testing is able to show the absence of specific faults — under certain (strong) assumptions. What kind of assumptions are needed will be discussed throughout this introduction. Fault models play an essential role. Based on these assumptions, we developed techniques for generating test cases that can show the absence of specific faults. The technique is known as mutation testing [Ham00, DLS08]. We will discuss our contributions to this fault-oriented form of testing.

This habilitation thesis comprises of 21 papers plus this introduction chapter. The bibliographic references to these papers can be found on Page 22. The papers have been selected from the set of our publications on a thematic basis. In all of them we developed the idea of mutation testing on the modelling level, but put into different contexts. In some we develop the foundations based on sound theory, in some we take the theory and turn them into tools for test case generation. In others, these tools or existing tools are applied in real case studies. The theory-to-application thematic line is one dimension of research. Another dimension that can be identified are the different semantic domains of the models: mutation testing of relational contracts, recursive programs, reactive systems, and hybrid systems.

Having made this selection, leads to an obvious consequence: many good papers the author is proud of having co-authored in recent years have been left out. Most of them are related though, since the author’s main research focus has been on formal testing in recent years. Whenever opportune, we will reference to them throughout this chapter. A list of these references of our own related work can be found on Page 24.

The purpose of this introductory chapter is to give a good overview of the central ideas, of the contributions made and of their relations among each other as well as to other related work. Therefore, we refrain from including technical formulae and refer to the papers for details. In order to facilitate a quick mapping between the habilitation papers and the corresponding overview presentation in this chapter, the references to the habilitation papers are also presented as margin notes.

Structure. The rest of this chapter is structured as follows. Section 1.2 provides a brief introduction to model-based testing. We give an overview of the modelling languages we have applied until now. At the end of this section we provide a high-level view of the challenges faced in our contributions to model-based testing. Section 1.3 introduces classical mutation testing. A recent survey on mutation testing shows that our work is relevant. Section 1.4 motivates
formal testing with formal models, formal semantics, and formal conformance relations. Different conformance relations relevant to our work are discussed. At the end of this section, we consider testing assumptions. Section 5 introduces our novel view on test cases: a test case or test suite may be viewed as a formal specification. Section 6 discusses our research papers forming this thesis. We organise the presentation according to the modelling paradigms for contracts, communication protocols, embedded systems and hybrid systems. Finally, Section 7 draws our conclusions, including a summary; a discussion and an outlook to future work.

Acknowledgement. The papers in this habilitation thesis have been co-authored with the following colleagues: Farhad Arbab, Lacramioara Astefanoaei, Frank S. de Boer, Harald Brandl, Carlo Corrales Delgado, Chris George, Jifeng He, Andreas Griesmayer, Einar Broch Johnsen, Elisabeth Jöbstl, Willibald Krenn, Sun Meng, Bernhard Peischl, Jan Rutten, Percy Antonio Pari Salas, Rudolf Schlatte, Rupert Schlickl, Martin Weighrofer and Franz Wotawa.

1.2 Model-Based Testing

Most of our research presented in this thesis contributes to the area of model-based testing. Model-based testing is a black-box testing technique focusing on the external behaviour of a system under test (SUT). Hence, we assume that we have no access to the internals of the SUT, like e.g., the source code. The test stimuli are automatically generated from an abstract model of the SUT. This test model is usually derived from the requirements. The model serves also as a test oracle providing the verdict (pass or fail) of a test case execution. The models are expressed in special modelling languages that support the abstract specification of the central properties to be tested. A detailed introduction to model-based testing can be found in [UL07 UPL11].

Modelling Languages Used. Not all modelling languages serve all needs. In our work, we have applied the following modelling languages for expressing test models (listed in chronological order):

1. VDM [Jon90] for testing air-traffic communication systems [44] [45] [33],
2. RAISE [Cro95] for testing data type implementations [40] [34],
3. OCL [OMG10] for testing against UML contracts [5],
4. LOTOS [ISO89] for testing communication protocols [7] [8] [9] [10] [11],
5. NuSMV language [CCGR99] for testing automotive controllers [41],
6. Crel [JO07] for testing distributed object-oriented systems [49] [42] [13],
7. Spec# [BLS05] for testing C# programs [6],
8. Qualitative Reasoning models [Kui94] for testing continuous systems [18],
10. REO [Arb04] for testing coordinated networks of components [32, 14].

11. Symbolic Labelled Transition Systems for testing communication protocols [12] and embedded systems [50], and finally, most recently

12. UML state-machine diagrams for testing embedded systems [15, 16].

Why should practitioners accept the efforts to learn new modelling languages and create models along their implementations? The answer is cost reductions. Testing consumes up to 50% of the development costs in a mission-critical project. Once the models and adaptors are created, the test cases come for free; i.e., they are automatically generated. Furthermore, when requirements change, it is much easier to change an abstract model compared to updating hundreds of hand-written test cases. Similarly, when the interface changes, only the adaptors need to be updated. Hence, test automation to save costs is the major motivation from a practitioner’s point of view.

Furthermore, model-based testing offers new testing possibilities. The classical approach is offline testing, where the test stimuli and the expected outcomes are first generated from the model and then executed on the SUT. In online testing, the model and the SUT are executed in alternation. First, a test stimulus is generated from the model, then the actual output of the SUT is compared to the expected output defined in the model. If the correct output has been observed, the next stimulus is generated from the model. In the presented work, we focused on offline testing mostly due to industrial requirements. However, most of the techniques can be easily adapted to serve online testing.

In order to bridge the two levels of abstraction between the model and the implementation, a test adapter has to be implemented. The test adapter maps the abstract test cases generated from the model to the interface of the SUT. In offline testing, both the test stimuli and the expected outcomes are mapped to a concrete test case offline, i.e., before execution. When online testing is performed, the test stimuli are mapped online, i.e., during test execution, down to the concrete implementation level, and the actual outputs are mapped back to the abstract modelling level, also online. Online testing has advantages, when generating test cases from non-deterministic models. The search space stays smaller.

A non-deterministic model of a system allows several possible outputs for a given stimulus. This may capture a true non-determinism in the implementation or an underspecification due to abstraction. With non-determinism, the problem of test case generation and execution is of a different level of complexity: test cases are no longer sequential traces of input-output pairs, but have a tree-like shape, since the next input depends on the previous output. Such test cases are called adaptive. Furthermore, as explained above, the conformance relations are no longer equivalence relations, but order relations. For offline testing all possible output behaviours have to be explored in order to generate an adaptive test case. This is computationally costly. In online testing, the actual outputs are available and all other alternatives can be pruned away. There is no need to generate tree-like test cases.

However, not all test harnesses support online testing and when testing real-time systems, the delays due to the online test case generation and mappings may be unacceptable. Both reasons lead us to concentrate on offline testing in our work.
At this point, we want to emphasise that in our contributions we are targeting the most challenging problems in model-based testing: the automatic test case generation using

- offline testing,
- non-deterministic models,
- formal models,
- partial models

in order to cover specific faults. Formal models and partial models are discussed in Section 1.3. Most of the current model-based testing tools do not support this combination of challenges. None supports mutation testing.

1.3 Mutation Testing

Mutation testing is a way of assessing and improving a test suite by checking if its test cases can detect a number of injected faults in a program. The faults are introduced by syntactically changing the source code following patterns of typical programming errors. These deviations in the code are called mutations. The resulting faulty versions of the program are called mutants. Usually, each mutant includes only one mutation. Examples of typical mutations include renaming of variables, replacing operators, e.g., an assignment for an equivalence operator, and slightly changing Boolean and arithmetic expressions. The number and kind of mutations depend on the programming language and are defined as so-called mutation operators.

A mutation operator is a rewrite rule that defines how certain terms in the programming language are replaced by mutations. For every occurrence of the term the mutation operator rewrites the original program into a new mutant. After a set of mutants has been generated, the test cases are run both on the original and on each mutant. If a test case can distinguish a mutant from the original program, i.e., a different output behaviour can be observed, we say that this test case kills a mutant. The goal is to develop a test suite that kills all mutants. This technique of program testing has been invented by Hamlet [Ham77] and DeMillo et al. [DLS78] in the 1970s.

Mutation testing has three basic assumptions: (1) the competent programmer assumption, assumes that programmers make only small errors. This argument supports the use of small variations in the code to represent the fault models, i.e., the typical faults of programmers; (2) the chosen mutation operators are a representative set of those errors; (3) via a coupling effect, more subtle errors can be detected by testing against the simple errors only.

However, even with these assumptions there is a fundamental difficulty in this approach. Not all mutations represent actual faults producing observable failures. For example, a mutation in dead code, i.e., code that is not reachable, will not lead to failures. Mutants with such an equivalent behaviour are called equivalent mutants and can never be killed by any test case. This poses a serious limitation to the mutation testing technique: for a mutant surviving the tests, i.e., it is not killed, we do not know, if this is due to our inability to come up with
a proper test case, or due to the fact that there is no such test case, because it is an equivalent mutant.

Unfortunately, we cannot automatically exclude equivalent mutants from the process, since the problem is undecidable in general. Otherwise we would be able to distinguish terminating from non-terminating programs automatically, which would solve the undecidable halting problem. Hence, manual inspection of the surviving mutants was often needed in order to exclude the equivalent ones.

In recent years, with the advent of model checking techniques, the situation improved considerably: Today, the equivalence of two mutants can be decided for a growing class of programs assuming bounded (finite) data types. This may explain the returning interest in mutation testing. In a recent survey on mutation testing Jia and Harman point out the problem of equivalent mutants

“One barrier to wider application of Mutation Testing centres on the problems associated with Equivalent Mutants. As the survey shows, there has been a sustained interest in techniques for reducing the impact of equivalent mutants. This remains an unresolved problem.” \cite{YM10}

In this survey the authors also mention the need for test case generation techniques:

“Most work on Mutation Testing has been concerned with the generation of mutants. Comparatively less work has concentrated on the generation of test cases to kill mutants.” \cite{YM10}

The presented work in this thesis addresses this gap in research. As a sidetrack, we did investigate the generation of test vectors from mutated programs \cite{2,51}. However, our main contributions are based on the combination of model-based testing and mutation testing. Mutation testing provides the basic idea to support our thesis. If we can generate test cases for a specific fault, i.e. a mutation, then we will be able to show the absence of this fault in the SUT — at least in the deterministic case. Hence, part of our basic assumptions are the fault models represented as mutation operators as well as the other assumptions in mutation testing. Lifted on the modelling level, mutation testing can show that a wrong (mutated) model has not been implemented. What this precisely means, is discussed in the following section, where we argue for the importance of formal definitions in model-based testing in general, and in our contributions to mutation testing in particular.

1.4 Formal Testing

In order to support our thesis that testing can show the absence of failures under certain assumptions, we have to be precise about these assumptions. This means that we have to define the notion of failure that is so central to mutation testing. For simple program mutation this was quite clear. Whenever we observe a different behaviour of the mutant for the same input, this is defined as failure and a test case should kill this mutant.

However, the situation is not as simple as this. What if the mutated program is slower? Is this a failure? What if a program shows different behaviour for
undefined input? Is this wrong? When mutating models the situation becomes
even more complex. Models express abstractions of the real SUT and may
contain non-determinism. An abstract model may leave choices for the imple-
menter what to do (implementation freedom). In this case, simple equivalence is
not sufficient to define a failure, since the implementation may actually do less
than an abstract model. Therefore, more elaborate notions of conformance are
needed and they need to be precisely defined, based on a precise semantics of
the modelling languages. Thus, these conformance relations form an important
part of our assumptions.

Formal development methods have long been known to provide the theories
needed to express properties of software in a precise and unambiguous way,
including assumptions made. However, it was only when Gaudel prominently
suggested to apply these techniques in testing [Gau6], that formal testing slowly
became acceptable as a further formal technique in the scientific formal methods
community. The significance of her contribution was that she made the testing
assumptions explicit, for the first time. By doing so she developed a formal
testing theory. Hence, formal testing comprises formal models of the system-
under-test and a testing theory that captures the essential properties of a testing
technique.

In our work we focus on formal testing techniques contributing to a long tra-
dition of research: formal testing of software started in the late 1970s. Among
many other pioneering work we refer to [Cho78, GMH81, HCFG86, Bri88,
BGM91, DF93]. A number of surveys cover the whole development [LY96,
GG07, HBB+09, FWA09].

1.4.1 Formal Models
The basis of formal testing is a theory of the system under test. This theory
describes the expected behaviour of the SUT and serves as an oracle for test
verdicts, i.e. if a test passed or failed. Usually, this theory is established via
a modelling language with a formal semantics. It is this underlying precise
semantics that distinguishes formal testing from the more general model-based
testing approaches, where modelling languages lacking formal semantics, like
UML, are applied.

This does not necessarily mean that formal testing has to apply unorthodox
notations. In case industrial modelling notations are required, the needed subset
of the language is interpreted in a formal semantics. Of course, the price to pay
for this individual semantic interpretation is that it is non-standard and that
others might come up with different semantics.

For UML many semantics with different purposes exist. Some years ago
we studied co-algebraic interpretations of UML diagrams [CMR+95, L66,
Z66, C89]. Co-

algebras provide a very abstract form of semantics for state-oriented models.
More recently in our mutation testing tools for reactive systems, we have mapped
UML state machines to Back’s Action System formalism [B15, 17]. All of our other
research on model-based testing has been based on formal notations as listed in
Section 1.2.

The choice of a modelling language together with its semantics has a major
impact on the associated testing theory and techniques. For example, it is easy
to automatically extract test sequences from finite state machines, however, in
case of pre-postcondition contracts constraint-solving techniques are required.
In principle, three styles of semantics can be distinguished: operational, denotational and algebraic semantics. We have generated test cases for all of these and will briefly characterise them, going from the most concrete to the most abstract form of semantics.

**Operational Semantics** Operational semantics are very popular in concurrency theory and model checking. An operational semantics defines the meaning of a programming or modelling language in terms of abstract machines. It is operational in the sense that it explains the operational (execution) behaviour of a language. For example, an operational semantics of a Boolean expression, will define the evaluation order, e.g., from left to right. This is the oldest form of semantics and has been applied in compiler design since the 1960s. The virtual stack machine of Java is a prominent example of an operational semantics assigned to a programming language. However, for theory building usually more abstract machines, represented as (labelled) transition systems are used. Model checking is performed on various kinds of automata. For reasoning purposes, Plotkin has established the style of structured operational semantics [Plo61]. Here the behaviour of these transition systems is presented in the form of formal proof rules over the abstract syntax of the language.

LOTOS has an operational semantics defined via labelled transition systems. Hence, our testing work with LOTOS is based on operational semantics [7 8 9 10 11]. Similarly, we interpreted Action Systems as labelled transition systems by labelling the actions. The test cases are then generated with a similar technique as applied in the LOTOS approaches. This includes our extended form of Action Systems for testing hybrid systems [20 21]. This extension is based on our earlier work on testing continuous systems with Qualitative Reasoning models being mapped to transition systems [18]. Our mapping from UML to an Action System is an operational interpretation of UML state machines in terms of Action Systems, too [15 17]. Furthermore, symbolic labelled transition systems are themselves defined as labelled transition systems [12].

Test case generation with model checkers is also based on operational semantics, like our work with NuSMV [11]. Creol is another example of a language defined with an operational semantics. In contrast to our other work, Creol is defined in rewriting logic with the Maude system [CDE+02]. Modern tools, like Maude, can automatically generate an efficient interpreter from an operational semantics definition. We have extended such an operational semantics definition of the Creol modelling language in order to facilitate test case generation via dynamic symbolic execution, also known as *concolic execution* [13].

**Denotational Semantics** A denotational semantics is defined via a mapping from syntax to a semantic domain. For example in case of imperative languages, this interpretation function maps every statement to a theory over the observations before and after its execution. Here, only the effects of what is computed are defined and not, like in operational semantics, how this computation is actually realised. For example, the truth table of a Boolean operator represents the interpretation function mapping the operator and operands to their semantic
truth values.

The distinguishing feature of denotational semantics is the definition of recursion and iteration via fix-points. Hence, the semantic domain is designed to guarantee their existence (see Tarski’s theorem). The advantage of this form of semantics is that it is highly compositional, i.e. it supports the compositional reasoning over the language. Therefore, formal notations supported by proof systems usually have a denotational semantics, like e.g., VDM \cite{Jon90} and Action Systems \cite{BKS83}.

In our first scientific work we investigated the denotational semantics of VDM for developing a proof obligation generator for guaranteeing model consistency \cite{KS96,JS96}.

Another application of this semantic style is formal step-wise development via the definition of refinement rules. Such refinement rules help to decide if a given implementation is indeed correct (invent-and-verify) \cite{Jon90, Abr96}, or may even help in finding such implementations (program synthesis) \cite{BvW95, Mor90}. The proof rules themselves are consequences (theorems) of the given formal semantics. However, denotational semantics are not limited to proof systems: For deterministic subsets of the semantics the interpretation function is executable (like a functional program). The interpreter of VDM is an example of such an executable denotational semantics. More recently, with the advent of efficient SAT and constraint solvers, executability has been extended to the (bounded) non-deterministic language features as well.

In our early work on OCL contracts \cite{BS00}, we gave OCL a denotational semantics and developed a set-based constraint solver for test case generation. Similarly, the work on SpecC \cite{GHH04} and REO \cite{JS91} is based on denotational semantics of the specification languages.

In our theoretical work on mutation testing, we gave test cases a denotational semantics and related them via refinement to models and implementations \cite{JH02}. This enabled us to formally prove our mutation test case generation algorithms being correct. For reactive systems, we formalised processes and test cases and reformulated Tretmans’ well-known testing theory in a denotational predicative semantics \cite{JH03}.

Most recently we extended Action Systems with Qualitative Reasoning models for modelling hybrid systems. Before implementing a test case generator for such models \cite{JH06}, we have formally defined what this extension precisely means \cite{JH06}.

**Algebraic Semantics**  Gaudel defined her testing theory \cite{Gau95} in this semantic style. An algebraic semantics defines the meaning of syntax by enumerating its algebraic properties. The properties are given in the form of equational axioms, possibly with preconditions. Therefore, this style is also called axiomatic semantics. Boolean algebra is an example of an algebraic semantics for Boolean expressions or circuits. The style has been successfully applied to the definition of abstract data types (ADTs).

An abstract data type abstracts away from its internal data representation, but provides an interface with operations, a concept that became famous through object-orientated languages. An ADT’s behaviour is defined via the axioms over this operations. A canonical example is a stack, with the central axiom that a stack after a push-operation followed by a pop will behave as
The advantage of algebraic semantics is its full abstraction of any semantic model. The price to pay for this abstraction is that non-experts may find it hard to come up with a complete and sound set of axioms describing their intended behaviour. However, once the properties are fixed, formal proofs become easier due to equational reasoning with algebraic laws, rather than reasoning in the semantic model. Out of this motivation, the RAISE specification language (RSL) [Gro95], although being in the tradition of VDM, has an algebraic semantics. It is possible to combine algebraic laws with denotational models. We have exploited such a combination for automated test case generation [vt].

In our theoretical treatment of mutation testing, we have exploited the algebraic semantics of a programming language as rewriting rules. These rewriting rules transform a program into a normal form in which mutation-based test case generation is easier [2].

A case of incompleteness of a conformance relation and its relation to mutation testing was also discussed in the context of algebraic specifications [3]. In the next subsection, we take a closer look on such formal notions of conformance.

### 1.4.2 Conformance

Once the modelling language with a precise semantics is fixed, one can define what it means that a SUT conforms to a given reference model, i.e. if the observations of a SUT confirm the theory induced by a formal model. This is similar to scientific experiments in the natural sciences, where a scientific theory is confirmed against observations in nature. The main difference between scientific experiments and software testing is the interpretation of non-conformance: in science the theory is considered flawed, if it does not conform to the real observations made in nature, while in software testing non-conformance is usually interpreted as a failure in the SUT. However, many results from epistemology and the philosophy of science are related to software testing. In [3] we have discussed results from the philosophy of science in the context of fundamental issues in model-based testing. Details follow later in this section.

In formal testing, conformance is defined as a relation between a specification model and an implementation model. The latter represents the assumption that the SUT behaves according to the model semantics. If this relation does not hold, an observable failure has been detected. Hence, what is considered a failure is defined by the conformance relation. Consequently, the notion of conformance in a mutation technique forms an important part of the testing theory of mutation testing.

A trivial, widely used form of conformance is *observational equivalence*, which demands that a SUT produces exactly the same observations as the reference model. This is the principle of *regression testing*, where new software must behave exactly as its older versions. However, this conformance relation is rather strong and in general a specification model derived from the requirements will be incomplete and should leave implementation freedom. Hence, useful conformance relations are order relations rather than equivalence relations, the order going from abstract to more concrete models. In the following, we are going to discuss the conformance relations we have applied in our work and refer to alternatives.
Refinement

Conformance is closely related to the formal notion of program refinement. It stems from the area of program verification and answers the question, if a program can be safely replaced by a more efficient, refined version. The Vienna Development Method (VDM) has advocated refinement as a formal method for developing programs in a step-wise manner from abstract models down to code, with every refinement step being formally verified [Jon90]. Other formal methods supporting step-wise refinement are RAISE [Gro95] and the B-Method [Abr96]. In addition, refinement calculi have been developed that allow to derive refined programs by following a set of refinement laws [BvW98, Mor90]. All these refinement techniques are based on a refinement relation defined via the semantics of the language.

Refinement can be split into two kinds: operational refinement and data refinement. Operational refinement is refinement without changing the state space, e.g., implementing a pre-postcondition contract specified in VDM, RAISE, B, Eiffel, OCL, JML, or Spec# notation [Jon90, Gro95, Abr96, Mey97, LBR99, BLS95].

Data refinement maps between programs of different state spaces, e.g., a balanced binary tree implementing a set functionality. In the latter, a mapping between the abstract and concrete data is needed, e.g., mapping between binary trees and sets. For details on data refinement, we refer to [DE99]. Such mappings are also applied in model-based testing when abstract stimuli need to be converted to concrete data formats of the SUT and actual responses, being converted back in order to compare with the abstract expected responses. In our early work, we have shown how a VDM specification may serve as a test oracle by mapping concrete test values up to the specification level [24, 23, 25]. In the terminology of [DE99], this was an application of $L^{-1}$-simulation. In our work on hybrid systems, we used L-simulation to prove a refinement law linking qualitative and continuous Action System models [19].

Conformance, and hence refinement, depends on the observations that are defined in the theory of the formal semantics. Therefore, different refinement relations have been proposed for different semantics. In the following, we discuss the most relevant ones.

Relational refinement. Relations between (before-after) states is a standard (denotational) semantics for imperative sequential programs and their abstract contracts. Here, refinement is defined as relation-inclusion, meaning that a refinement (implementation) must not reach states that are forbidden by the abstract specification. If the state-relations are defined as predicates, refinement is defined as implication from the refined to the abstract. However, this only holds for total relations (specifications). In order to allow for partial specifications, the relations are defined via pre- and postconditions. Then, refinement can be characterised directly via pre- and postconditions: under refinement preconditions are weakened and postconditions are strengthened. For example, VDM [Jon90] and Hoare [HH98] use this kind of refinement for sequential programs.

Relational refinement was explicitly used in our mutation testing work on OCL [5], Spec# [6], REO [32, 14] and also in our second formalisation of mutation testing in the Unifying Theories of Programming (UTP) [2].
**Weakest-Precondition Refinement.** An alternative denotational semantics are weakest-preconditions used by Dijkstra and Scholten [DS90], Back [BvW98] and Abrial [Abr96]. Here the syntax of a modelling or programming statement is interpreted as a predicate transformer mapping a given postcondition to the weakest precondition such that the statement will satisfy the postcondition. Refinement is given if and only if the weakest precondition of the refinement is implied by the weakest precondition of its abstract specification. Therefore, compared to the relational model, the implication order is reversed.

This notion of refinement was used in our first formulation of mutation testing in the refinement calculus [1] and in the semantic work on hybrid systems [19].

**Axiomatic Refinement.** For an algebraic semantics a refinement must satisfy all the axioms of the abstract specification. For example, all implementations of a stack must satisfy the axioms of a stack. The benefit of this form of refinement lies in the easier equational proofs, which was one of the motivations for adopting this style in RAISE [Gro95]. Axiomatic refinement is incomplete, which may have surprising consequences in testing as discussed in [4].

**Traces Refinement.** Reactive systems may be non-terminating. Hence, different semantic models are needed. Therefore, in CSP [Hoa85] additional points of observation are introduced: so called events mark the synchronisation points between communicating processes where data is exchanged. The semantic domain of such systems are event traces, the possible sequences of events of a process. Refinement is then defined as trace inclusion, the traces of the abstract including the ones of the refined process. A consequence of this refinement notion is that the abstract models must be complete: there is no notion of a partial model as expressed in the pre-postcondition style. For partial modelling input-output conformance is needed (see below).

In our work on mutation testing of REO circuits, a timed-trace semantics was given to REO connectors [32, 14]. Here, trace inclusion was the chosen conformance relation.

Other related work of ours involved the testing of distributed systems. In this work the logged traces of the SUT were checked against executable Creol models. Hence, trace inclusion was tested [48, 55, 64, 49, 42, 13].

**Failure Refinement.** Traces semantics is not strong enough to express all semantic nuances of a language. For example, it cannot distinguish between internal and external choice of CSP [Hoa85]. Therefore, additional refusal sets have been introduced representing the events that cannot be accepted in a certain state. The according refinement extends traces refinement with the additional requirement that an implementation must only block, if the specification allows blocking. Hence, refinement is extended to negative (blocking) behaviour, an idea that has been adopted by the following notion of input-output conformance.

In our work on formalising the following input-output conformance relation, we used a semantic model for failure refinement and proved that failure-refinement implies input-output conformance [3].

Refinement does not distinguish between input and output, but the following conformance relation does.
Input-Output Conformance

The conformance relation \textit{ioco} has been developed by Tretmans for testing the behaviour of communicating systems \cite{Tretmans}. Focusing on communication, it is defined over models with a labelled transition system (LTS) semantics, the labels representing communication events.

Informally, ioco conformance is given, if for all event traces possible in the specification model, the implementation does not produce output that is not allowed by the specification. For a formal definition based on labelled-transition systems and predicate logic see \cite{Tretmans}. The latter is our contribution.

Like the refinement of pre-postcondition specifications it supports incomplete (partial) specification models. This is realised by splitting the events into input and output events, i.e. controllable and observable events. Hence, a SUT may react to unspecified input events in an arbitrary way, like programs outside their specification precondition. This is an important feature for industrial application, where it is unrealistic to model the complete behaviour of a system under test.

A further advantage of ioco is that it allows quite general specification models. They may be non-deterministic or have arbitrary interleavings of inputs and outputs in the models. For example, a specification may prescribe that in a certain state the system may accept input but alternatively may also issue output.

Important in practical software testing is the detection of absence of (output) reaction. Therefore, ioco adds an additional observation called quiescence to its alphabet of output events. This observation of quiescence can be easily implemented via timeout watchdogs reporting the absence of response from the SUT in a given time limit. Hence, with ioco one can test global real-time response times.

For this reasons, ioco has been the choice in our later work on testing protocols with LOTOS \cite{LOTOS} \cite{LOTOS2} \cite{LOTOS3}, and our more recent work on testing embedded systems with Action Systems \cite{As} \cite{As2} \cite{As3} \cite{As4}.

In \cite{As4} we presented the details of our ioco checker that produces a test case as a counterexample for input-output conformance between two models.

Assumptions in IOCO. However, ioco is also based on assumptions of which some are only informally stated. Therefore, in \cite{Tretmans} we have reexamined ioco and reformulated it in a denotational, predicative semantics. The benefits of this new theory can be summarised as follows: (1) Instead of describing the assumptions of ioco informally, the new formalisation presents the underlying assumptions as unambiguous healthiness conditions and by adopted choice operators over reactive processes; (2) Our formalisation naturally relates ioco and refinement in one theory; (3) The denotational version of ioco enables formal, machine checkable proofs. (4) Due to the predicative semantics, test case generation based on the presented theory can be seen as a satisfiability problem. This facilitates the use of modern SAT modulo theory techniques (SMT) for test case generation. (5) Finally, this version of ioco broadens the scope of ioco to specification languages with similar semantics, e.g., to generate test cases from Action System specifications.
Alternatives

Further alternatives to ioco for defining conformance of reactive systems exist. In general, we can distinguish between two kinds of interpretations for behavioural conformance: global interpretations as sequences of events and local interpretations as successor events at individual states. Examples for global interpretations are traces refinement, failure refinement and ioco. Examples for local interpretations are all kinds of simulation preorders, including alternating simulation [AH91].

Ioco is global, since it refers to traces of the specification. For calculating a test verdict all non-deterministic choices along a specified trace have to be explored. This is less efficient than comparing the next events in a state as in local conformance relations. However, the advantage is that non-determinism in a model adequately represents underspecification and not the local choices of the SUT. An important property in the context of black-box testing.

Local simulation conformance relations are stronger than global conformance properties. Here, all the local choices are tested for conformance as well, i.e. one tests if taken local steps in the SUT are possible steps in the specification. The advantage is that conformance can be decided locally by forward simulation. This is faster. The disadvantage is that implementations may be rejected due to wrong non-deterministic choices in the specification. To overcome this, one has to determinise the models such that all non-deterministic choices are postponed to the end: Hoare and He have shown how to convert process models such that a simulation check is equivalent to traces inclusion refinement [HH06]. More recently it was shown that if models are determinised beforehand, then ioco and alternating simulation are equivalent [VB10].

However, such transformations are not possible in black-box testing without access to the implementation. Therefore, we mostly relied on global refinement relations in our work, with one exception:

Our first mutation testing work for reactive systems used the local property bi-simulation after some simplification of the models. The motivation was the availability of a ready bi-simulation checker. The checker was used to compare an original and mutated model. If the check failed, a test case was generated leading to the failure. Hence, in this setting access to both models is given.

However, bi-simulation is a too strong conformance relation that resulted in redundant test cases. Therefore, we later implemented our own ioco checkers, one for LOTOS [8] and one for Action Systems [21].

Incompleteness of Conformance

Conformance relations need to be complete, i.e. that all correct implementations of a specification can be identified by a conformance relation. However, from data refinement proofs it is well-known that not all conformance relations are complete. For example, downward simulation alone is not sufficient to prove all correct implementations as being refinements [HHS86].

For testing, a possible incompleteness of a conformance relation has severe consequences: a correct implementation under test might be rejected by the testing team although its observations cannot be distinguished from an accepted implementation. For example, this would be the case if we base our testing on local simulation relations. In certain cases of non-determinism our simulation
test would reject models that are correct from an external black-box view. The reason is that simulation is a local refinement property and hence not generally applicable to model-based black-box testing.

In [H] we have identified such a case of incompleteness in the context of algebraic specifications. Haeberer and Cengarle [CH00, HM01] had previously translated epistemological results from philosophy of science to software testing and criticised the classical approaches to testing from algebraic specifications. They presented several counterexamples where this testing method fails. In these examples the generated test cases report false negatives: some software modules get rejected, although observationally equivalent to others which pass the tests. Haeberer and Cengarle proposed Glymour’s bootstrap testing approach for confirming scientific theories as an alternative to overcome these problems.

Clarke Glymour had shown in [Gly80a, Gly80b] that the classical approach to derive the expected observations from the hypothesis and a background theory has fundamental problems and that all attempts to rescue this hypotheticoductive approach have failed. Identifying algebraic testing techniques as hypotheticoductive was the key criticism of Haeberer and Cengarle. In our work we have analysed their arguments from the conformance point of view. It turned out that the counterexamples to algebraic testing failed exactly due to the well-known incompleteness of axiomatic refinement. Furthermore, we could link our own testing theory to Glymour’s confirmation theory. This enabled us to show that some of Glymour’s attacks on the hypotheticoductive (HD) confirmation method are not that severe in software testing. One of the criticisms was that with HD any stronger theory would be confirmed as well. However, this is correct in model-based testing, since it means that any refinement would get confirmed as well. The general contribution of this branch of our research is a technical analysis of fundamental pitfalls in model-based testing inspired by epistemological arguments.

1.4.3 Testing Assumptions

Explicit assumptions are the key to a sound and complete testing technique. We have just seen in Section 1.4.2 that assumptions are necessary to ensure completeness of certain conformance relations. In general, every testing framework comes with a set of assumptions and it is vital for a scientifically defensible testing technique that these assumptions are precisely defined.

However, there is a second important class of assumptions. Assumptions on the system under test may allow to reduce the necessary test cases to be executed. Hence, testing assumptions may come in two forms: (1) as a precondition for a testing technique to work; (2) as a test hypothesis forming the basis for reducing the necessary test cases. In the following, we are going to discuss the two cases in more detail.

Preconditions for Testing Techniques

A common precondition for many model-based testing techniques is the assumption that the model and the implementation behave deterministically, i.e. that a given stimuli and set-up will always result in the same behaviour. As already discussed in Section 1.4.2, this is a precondition for local simulation techniques
applied to test case generation problems. Furthermore, many of the classical
test case generation approaches for finite state machine models make the
determinism assumption [Cho78]. More recently the growing work on test case
generation with model checkers [FWA09] usually relies on deterministic models,
too.

However, excluding non-determinism has disadvantages: A deterministic
model is less abstract and a test model should be more abstract than an
implementation. Assuming determinism in an implementation implies having full
control over the execution platform. The latter is often unrealistic in concurrent
systems with schedulers.

Another strong assumption is that the specification model is complete, i.e.
that it captures the full behaviour of the SUT. All forms of traces refinement
testing make this assumption, e.g. [CG10]. In our work on testing distributed
systems we also assumed complete models [15, 35, 34, 19, 42, 13]. However, for
larger systems complete models are often unrealistic. Hence, a major effort in
testing research has been to overcome such restricting assumptions.

Tretmans’ ioco conformance testing (Section 1.4.2) is designed for incomplete
models and non-determinism. However, other strong assumptions are made. In
our work we have developed a new denotational theory of ioco that is able to express
this assumptions as formal healthiness conditions. This is the first time that all
of ioco’s assumptions have been formalised. These assumptions are: exclusion
of live-locks in the models (no divergence); an additional observation in case
of absence of reaction after a time-out (quiescence); a SUT must always accept
every input (input-enabledness); a SUT has full control over its outputs (internal
choice); an implementation eventually shows all its possible non-deterministic
behaviours when it is re-executed with a particular set of inputs (fairness).

Test Hypotheses

The second class of test assumptions are test hypotheses. A test hypothesis is
a property that the tester believes that a SUT satisfies. Such an assumption
on the SUT may serve as an argument for reducing the number of test cases.
Hence, a test hypothesis forms the basis for a test selection strategy and must
be stated explicitly in any formal testing approach. There are two extreme
cases. One is the absence of a hypothesis which implies the need to test exhaustively, i.e.
with all possible combinations of input stimuli. The other extreme
test hypothesis would be that the SUT is already correct, hence no testing is
needed. The practise and challenge of testing is to come up with reasonable
hypotheses in between these extremes. Gaudel distinguished two classes of test
hypotheses that are commonly found in testing [Gau96]: uniformity hypotheses
and regularity hypotheses.

A uniformity hypothesis assumes that the SUT will behave equivalently for
certain classes of inputs. Based on this hypothesis, it is sufficient to test with
one representative out of each equivalence class of input stimuli. Forming such
equivalence classes is a common practise in testing. Dick and Faivre have shown
how this process of forming equivalence partitions can be automated by transq
forming a specification into a disjunctive normal form [DF93].

A regularity hypothesis assumes that it is sufficient to test with input data
up to a certain size. The program will behave similarly for larger inputs. This
hypothesis is applied when testing programs with data collections, like arrays
or lists and aims for testing recursion or loops. The idea is that after a sufficient number of iterations through a loop, one can safely assume that the loop will behave regularly for the rest of iterations. This, too, is commonly found in testing, when the number of paths in a program is limited by fixing the maximal number of allowed loop iterations.

Recently, it was shown that such test hypotheses together with a number of test cases can be used to formally verify the correctness of software with a theorem prover [BBW08]. This work shows that turning the test hypotheses into formal assumptions transforms the testing process into a formal verification process.

In the next section, we will discuss our contribution of integrating testing into program semantics and verification: we view test cases as a special form of specification.

### 1.5 The Role of Test Cases as Specifications

A novel contribution of our work to formal testing is that we give test cases a formal denotational semantics. This semantics facilitates the seamless integration of test cases and test suites into program semantics. This integration is achieved by viewing test cases as specification of the system under test, i.e. for a given test stimuli the system should behave with an expected test response. This natural view of test cases assigns to a test case the meaning of a partial specification, for which the behaviour is defined for the given test stimuli, but is undefined for other values.

In the case of simple input-output test cases for sequential programs, a test case is like a contract, where the input defines the precondition and the expected output the postcondition of the contract. This contract view of test cases corresponds to the role of test cases in acceptance testing. In acceptance testing the test cases form part of the contract: if the system fails these tests, the developers violated the contract.

We presented this view on test cases first in the weakest-precondition semantics of the refinement calculus [26, 27, 28, 29, 30, 1], and later reformulated it in the relational semantics of the Unifying Theory of Programming [2]. These papers concentrate on sequential programs. For concurrent processes, our notion of abstraction is based on the event-view of ioc. In [7] we used this formal abstraction relation between test purposes, test cases and formal models as basis for our theory of protocol testing. Here, a test purpose is a specification of one or more test cases to satisfy a certain test goal.

Although our view of test cases is very natural it is sometimes hard to accept, because it has a non-intuitive technical consequence: Test cases are more abstract than specifications of a system under test. Since test cases have a very weak precondition, i.e. a single test input, the behaviour of the system under test is undefined for all other inputs. Weakening the preconditions is the opposite of refinement and hence abstraction. However, the syntax of a test case appears very concrete, therefore this confuses people. The paradox can be solved by separating the syntax and semantics of test cases. The syntax is small and concrete, but the interpretation of a test case as a contract is a very abstract concept.

The fact that test cases are abstract can also be easily understood from a
verification point of view. Abstract specifications allow more behaviours of a
system and hence represent a greater level of implementation freedom. Thus, the
more abstract or weaker a specification, the more implementations will satisfy
this specification. Therefore, a small number of test cases will always form a
weaker specification than a detailed formal model. Hence, the test cases are a
more abstract contract. Furthermore, with a decreasing number of test cases
this description becomes more abstract up to the extreme with no test cases
allowing any implementation to satisfy the (empty) contract. Usually, testers
do the opposite of abstraction: refinement. They develop a test suite by adding
adequate test cases.

This view of test cases as partial specification is important to integrate a
testing theory with a theory of programming. It allows us to relate the imple-
mentation, its formal model and test cases via refinement. This facilitates the
proof of test case generation algorithms from formal models and led to an easy
and very general characterisation of mutation testing.

1.6 Model-Based Mutation Testing

In the following, we introduce the combination of model-based testing and muta-
tion analysis and discuss our own contributions in this growing area of research.

1.6.1 Mutation of Models

The common idea of the papers forming this thesis is to combine model-based
testing and mutation testing. We call this approach model-based mutation test-
ing. Like in model-based testing the aim is to use the model for both, generating
test vectors and as a test oracle. Hence, we generate test cases from a model
in order to test the conformance of a SUT. In contrast to classical model-based
testing, only those test cases are generated that would kill a set of mutated
models. The generated tests are then executed on the SUT and will detect if
a mutated model has been implemented. Hence, model-based mutation test-
ing rather tests against non-conformance, than for conformance. In terms of
epistemology, we are rather aiming for falsification than for verification. It is a
complementary fault-centred testing approach.

In the following subsections, we will discuss our published results on model-
based mutation testing. The common technique is to use counterexamples to
conformance between the model and its mutants as test cases. The variation in
the work stems from the choice of the following parameters leading to different
test case generation techniques: (1) the modelling paradigm; (2) deterministic
or non-deterministic models; (3) the conformance relation; (4) the supporting
technology; and (5) efficiency considerations. We organise the presentation
according to the first parameter, the modelling paradigm, and cover contracts,
communication protocols, embedded system models, and finally hybrid systems.

1.6.2 Contracts

Contracts are pre-postcondition specifications added to a program source code.
Semantically, they represent relations between the program’s state before and af-
ter execution. They are well-known in VDM [Jon90], RAISE [Gro95], B [Abr93].
Theoretical Results in the Refinement Calculus. Our first work on model-based mutation testing extended mutation testing to the general notion of contracts \cite{Mey97}, which included both pre-postcondition specifications as well as executable programs. The idea was to mutate the contracts and derive test cases that would kill implementations of the mutated contract. In this work, we used the refinement calculus of Back and von Wright for formulating our first testing theory of mutation testing. This theory formalised the notion of test cases and used the concept of refinement to relate test cases, models and implementations. This work introduced our specification-view of test cases as described in Section \ref{sec:specification-view}.

Here we also established the first formalisation of the criterion for all the tests that would kill a mutant based on non-conformance. The conformance relation used was weakest-precondition refinement (see Section \ref{sec:precondition-refinement}). The central idea was that one should generate test cases that are abstract specifications of the contract, but not of the mutated contract. Furthermore, the paper presents the V-diagram of the V-development process \cite{Thu96} as a commuting diagram with the arrows denoting refinement. This formalisation of the V-diagram relating step-wise development and testing was first published in \cite{Thu96}.

Further Theoretical Results in the Unifying Theories of Programming. We continued this theoretical work in \cite{2} and developed more specific conditions for generating mutation tests from contracts. Due to our close collaboration with He Jifeng we switched to the formal framework of his Unifying Theories of Programming (UTP) \cite{HH98}. The idea of this work is to exploit the negated refinement laws of UTP. The result was a condition for a mutation test case for non-deterministic contracts: the input should cover the case where the mutant allows behaviour that is forbidden by the specification. In addition, the tests should cover valid inputs with undefined behaviour in the mutated specification. We could formally prove our condition being correct in the sense that the resulting test cases conform to the original contract but do not conform to the mutant. This was the first testing theory added to UTP and the first mutation testing theory for non-deterministic contracts.

Since UTP generalises its theory from contracts to executable programs, we extended our testing theory to a small programming language as well. The language is small but non-trivial, since it contains general recursion and non-deterministic choice. Here, we developed laws to transform such programs into a normal form of guarded commands. Then, we presented a formula for mutation test cases derived from an original and mutated program in this normal form. We formally proved the correctness of this formula in the predicative semantics of UTP. Each step in the proof was checked by a SAT solver.

The first results of this work were presented in 2006 already \cite{2}. An extended version of this article appeared as a book chapter \cite{3}.

OCL Contracts. The first implementation of these theoretical results on contract mutation was presented in \cite{5}. In this work UML contracts in the object-constraint language OCL are translated to a constraint solver. We developed our own constraint solver for sets using a Java library for Constraint Handling
The result was a tool that would take an original and mutated contract as input and then generated a test case covering the fault in the mutant. The generated test case also included the set of expected outputs.

**Spec# Contracts.** Later we applied this concept also to contracts in the C# language [6]. This time we exploited Microsoft’s tool-set for C# and its associated contract language Spec#. We showed how to implement our mutation testing technique with a combination of the verification generator Boogie [BCD+06] and the SMT solver Z3 [AMR08]. Boogie simplifies the contract statements, generates first order formulae and forwards them to the solver Z3. A counterexample is generated when a formula cannot be proved. For mutation testing we linked the original and mutated contract in the following way: The original contract was added to an implementation that calls a faulty method, but is annotated with a mutated contract. Boogie only uses this mutated contract when checking the method call against the original specification. Then, a counterexample is generated from which a mutation test case can be extracted. The work shows how an existing verification tool for annotated programs can be exploited for generating model-based mutation tests.

**1.6.3 Communication Protocols**

In recent years we have successfully applied model-based mutation testing to several implementations of communication protocols. In this domain we are interested in sequences of observable communication events. Hence, the generated test cases have the form of event sequences in the deterministic case, or they have a tree-like shape in the non-deterministic case. This is in contrast to the work on contracts, where we only generated test cases as input-output vectors. Obviously the applied conformance relations are event-based, too, like traces refinement, failure refinement, input-output conformance and all simulation relations (see Section 1.4.2).

**Faults as Test Purposes.** Our first work in this domain was the model-based testing of the Apache web-server [7]. In this project we modelled parts of the HTTP-protocol in the process algebra LOTOS [ISO89]. LOTOS models represent the exchange of messages as events and describe the possible sequences of such events in form of recursive processes. The motivation for using LOTOS was the available CADP-toolbox [GMLS07] with its powerful test case generator TGV [LJ05]. This paper contributed to four problem areas.

First, the early work on conformance testing in the area of distributed systems was mainly concerned with the soundness and completeness of the testing theory. Emphasis was given to develop a realistic conformance relation and a test case generation algorithm that was sound (no false negatives) and complete (no false positives). Since the models were finite labelled transition systems (LTSs), the problem of how to select a manageable subset out of the exhaustive test set was not a major concern. Abstraction was used to cope with the complexity. This lack of a test selection strategy limited the application domain to highly abstract protocol specifications. In this work we proposed to use the mutation testing criterion to select test cases.
Second, to overcome the lack of a test selection criterion, TGV uses test purposes. A test purpose is a special LTS that specifies the subset of test cases to be generated. With test purposes, a tester can steer a test case generator according to his strategy. However, the problem remains, how many and which test purposes to select. Thus, the problem has been lifted, but not entirely solved. In our approach, we wanted to support the tester in formalising test purposes, by turning his focus on possible faults. The idea of this paper was to generate test purposes covering the model mutations.

Third, we approached the equivalent mutants problem by using CADP’s equivalence checkers. Here, we first simplified and determinised the model and then used a bi-simulation check to exclude equivalent mutants.

Fourth, we solved the problem of using counterexamples for non-deterministic models. Many use the counterexamples (or witnesses) produced by a model checker as test cases. However, a counterexample is not a test case in the traditional sense. A test case should provide the stimuli and the responses for a system. However, a counterexample exemplifies only one possible choice of computation (a path). In case of non-determinism involved this is not sufficient for a test case, since a test case should predict and take care of all possible responses, as well as reject wrong responses. Therefore, we proposed to use the counterexample as a test purpose. A test case generator, then, will generate a proper test case to cover the counterexample. Hence, our idea was to generate test purposes from injected faults, such that the generated test cases will discover this anticipated faults. To our knowledge this was the first work on generating test purposes via specification mutation.

For mutation we used 27 mutation operators, some of which had been newly developed for LOTOS. We produced 1491 mutants of which 342 were semantically equivalent. Since the whole process was not automated yet, we selected 100 interesting mutants for test case generation. The test execution of the 100 tests was done manually via Telnet. We did not expect to find major flaws since Apache had been widely used for years. However, we found some unexpected behaviour in the conditional requests caused by ambiguous requirements.

**Exploiting the Fault Location.** We continued this line of research with several improvements. First, we had to resolve scalability issues. The method above did not scale to large LOTOS models, because the technique required the construction of the complete labelled transition system (LTS) prior to CADP’s equivalence check. This was highlighted in a case study on testing a commercial and an open-source SIP registrar [v] [v]. The session initiation protocol (SIP) is used in voice-over-IP applications to register and to connect communication partners.

During the construction of the LTS for the SIP Registrar model, the CADP toolbox ran out of memory (2 GB) after 11 days. Hence, we invented a slicing technique exploiting the knowledge where the model has been mutated. We marked the places of mutation in the LOTOS model with special labels and extracted slices that include the relevant parts only. The slice could be calculated automatically with a special test purpose in TGV. However, the approach is restricted to mutations not affecting internal transitions.

Since this mutation approach was not yet automated we randomly selected three mutation operators and applied our approach manually. The result were
91 mutants of which 7 were equivalent. To 69 mutants our slicing technique was applicable generating 124 test cases in over 4 hours. In the case study we compared the mutation approach to the usual test case generation with handwritten test purposes. The six handwritten test purposes generated over 6000 test cases in about the same time.

The tests of the handwritten test purposes discovered nine different faults in the commercial SIP registrar. In the case of the OpenSER Registrar we found four discrepancies between the SUT and the specification. This shows the power of model-based testing. Encouraging was the fact that the small set of mutation tests uncovered one additional fault in the commercial implementation.

In [9] we evaluate different strategies for our slicing technique. The paper analyses three techniques where to place a special label that marks the transitions that should be excluded from the slice. If done wrongly, the mutation is not reachable and TGV cannot generate a valid slice. The simplest strategy only considers the process where the mutation is placed and cuts off alternative branches. The two advanced strategies incrementally enlarge the slice in a trial-and-error fashion. The paper demonstrates that a breadth-first widening of the slice is better than depth-first. The simple strategy performs poorly.

The breadth-first search worked but suffered in performance due to repeated invocations of TGV. Furthermore, the slices were in some cases still larger than necessary. In [10] we further improved this technique by using an on-the-fly model checker to search a path to the mutation label. This search is implemented by verifying that the mutation label is not reachable. The model checker produces a counterexample trace that contains all labels needed for reaching the mutation. From this a more accurate test purpose for slicing the model can be produced resulting in smaller slices. For the SIP protocol the reachability analysis took in average about 5 minutes. Additionally, it took in average 11 minutes for producing the slice of an original model and over 2 hours for the slice of a mutant. As is often the case, a small number of slices (23%) were complex and pushed the average up. The rest could be generated in about ten minutes each.

Using an IOCO Checker. The second important contribution of this paper [10] was the use of an input-output conformance checker instead of CADP’s bisimulation checker. The bisimulation check of CADP was fast but obviously too strong. The model was determinised and simplified before the bisimulation check, but bisimulation is an equivalence relation. However, abstract models need a conformance relation that is an order relation (see also Section 1.4.2). As a consequence of the stronger equivalence relation more checks failed generating more test cases than actually needed. Therefore, an input-output conformance checker was newly developed, producing a test purpose as a counterexample for ioco.

At this stage of research the whole test case generation approach was fully automated, including mutation, slicing, conformance checking and test case generation. The paper compares the previous bisimulation with the new ioco approach. The first observation is that the bi-simulation check is much faster than the ioco check (on average 4.35s vs. 138s). However, the ioco check resulted in one third fewer test cases. The real test execution showed that the reduced test suite detected the same faults as the bigger suite. Hence, higher generation efforts result in less execution efforts.
Two Case Studies. In [11] we summarised our previous results and presented the detailed analysis of testing the SIP and Conference protocols with our model-based mutation testing approach. This work discusses the modelling of the protocols, including the kind of abstractions and simplifications needed. In order to overcome the state space explosion problem, we introduced the notion of bounded ioco checking and present the technical details of our bounded on-the-fly ioco checker.

We chose the Conference Protocol as an additional case study, because it is a well-known benchmark example in the area of model-based testing. The specification is available in different specification languages and 27 erroneous implementations can be used to evaluate testing techniques.

The Conference Protocol is less complex than SIP: Our mutation tool produced 910 mutants for SIP, compared to 400 mutants for the Conference Protocol. The ioco check between two SIP models took on average over 6 minutes, for the Conference Protocol 23 seconds. Both models produce a high number of equivalent mutants: 52% of SIP’s and 60% of Conference Protocol’s mutants were ioco conform.

We tested two configurations of an open source implementation of SIP and the 27 faulty implementations of the Conference Protocol. The 428 test cases for SIP detected four differences between the SUT and our models. The 124 test cases for the Conference Protocol found only seven out of the 27 faulty implementations. The main reason was the low boundary of r’ for the ioco check. Longer test cases were needed for most of the existing faults. Hence, the choice of boundary value is critical.

We also compared the model-based mutation testing approach with random testing and two kinds of scenario-based testing, using TGV’s test purposes. In one scenario-based approach, we generated one test case per test purpose, in the other all possible test cases resulting in a huge test suite. Hence, the number of test cases varied significantly. For random testing 100 tests were selected.

For the SIP, mutation testing (4 faults detected) was better than scenario-based testing (2 faults detected) with one test case per scenario, but worse than random testing (5 faults detected). The highest number of faults (6 faults) was detected by the large test suite of the all-tests-per-scenario strategy. However, the latter has a high price: Over 6800 test cases were needed. Only this high number of tests revealed all the faults detected by the other strategies. The other strategies complemented each other.

For the conference protocol the other approaches performed much better than mutation testing: random testing detected 19, one-test-per-scenario 17 all-tests-per-scenario 23, compared to the low number of mutation testing 7. As mentioned above, the mutation tests were too short.

Our conclusion from these two case studies was that mutation testing should be combined with other selection strategies if a low number of test cases is required. In our work on embedded systems (see Section 1.6.4) we have incorporated this finding and combined random with mutation testing. Furthermore, we have experimented with test selection strategies generating all test cases killing a mutant.

Symbolic Techniques. The Session Initiation Protocol (SIP) and Conference Protocol also served as case studies for evaluating symbolic test case generation
techniques. The test case generation above had been based on explicit state enumeration. In [12] we evaluated the symbolic test case generator STG. To our surprise the tool based on Binary Decision Diagrams (BDDs) did not terminate on the SIP protocol, but only on the Conference Protocol. Hence, we implemented a symbolic execution [Kin76] engine to generate test cases, which performed well for the SIP, but badly for the conference protocol. This showed again that symbolic techniques are highly dependent on the shape of the models. Which symbolic techniques are best for model-based mutation testing is a topic of ongoing research.

Symbolic execution can be combined with concrete execution. This combination is known as dynamic symbolic execution or concolic execution [GKS05]. In [13] we used this technique for generating test sequences from sensor network models. In dynamic symbolic execution a concrete test execution runs in parallel to a symbolic execution along the same path. The concrete execution determines the path to be executed. The symbolic execution records the path condition that has been taken. After such a combined run, the next input value is selected from the negated path condition. This guarantees that a new path will be taken. The path exploration continues until all paths have been covered.

In this work we applied dynamic symbolic execution the first time to models of distributed systems involving asynchronous method calls and non-deterministic scheduling of interleaved processes. The technique has been formalised in terms of rewriting logic and implemented in the Maude rewriting system. We applied it to a peer-to-peer system, an industrial information system and to wireless sensor networks. Our testing technique forms part of a methodology for modelling and analysing highly reconfigurable distributed systems [7s1 7r].

Exogenous Coordination. An interesting alternative to process algebras like LOTOS is the coordination language REO [Arb04]. In process algebras the protocol is expressed rather implicitly in terms of interacting processes. These models are like abstract implementations of the protocol. The explicit view on possible interactions is only gained after mapping it to its labelled transition system semantics followed by simplification algorithms. As a consequence, parallelism is expressed via interleaving semantics.

The coordination language REO adopts an alternative approach. It is a visual modelling language for expressing a network coordinating the communication between a set of components. The coordination is exogenous, which means that the network is responsible for connecting and synchronising the communication. The network is modelled by composing a set of basic connectors. Various formal semantics for REO exist, but basically it is a data flow language with added synchronisation. This data flow view of a REO network expresses a protocol graphically by visually connecting the ports of the components with different connectors. As a result, the protocol is not hidden in the communicating components, but made explicit via the network representation.

This new language for protocols opens new opportunities for mutation. For example, exchanging one type of a connector by another, changes the coordination pattern of a network. Hence, new fault models can be expressed by single (first order) mutations.

In [32] we presented the first mutation testing approach for REO. The basis of this work was a new REO semantics expressed in the Unifying Theories of
Programming (UTP). The new semantics expressed the behaviour of a connector as a relation between input and output data streams. This formulation made it possible to adopt our earlier results on mutation testing in UTP \cite{2}. It was quite encouraging to see that our theory on mutation testing for non-deterministic contracts was general enough to be applied to a coordination language as well.

The technique has been implemented in a prototype tool for testing REO connectors using Maude \cite{CDE02}. In this tool one can search for test cases identifying faulty connectors.

We described the technique in more detail in \cite{14}. In this more recent work we changed our implementation language from Maude to Tom \cite{GMR04}. Maude was great in the early work when experimenting with the theory, but Tom is more flexible. It is a powerful and efficient pattern matching engine on top of conventional programming languages like C or Java. By design, it is meant to support pattern matching against native data structures like objects or records. A refinement checker has been implemented in JTom, the Java-version of Tom. In case of non-refinement, the tool produces a test case as a counterexample. The interesting point regarding implementation is that here a rewriting technique has been used to find the counterexamples, compared to the constraint solving technique reported in our work on contracts \cite{5}.

1.6.4 Embedded Systems

Another line of research was the model-based mutation testing of control software in embedded systems.

Mapping UML to Action Systems. In order to support the formal testing from informal UML diagrams we defined a precise semantics for UML state-machines by mapping them to formal intermediate models. As a formal target language Back’s Action Systems have been chosen. Action systems are a kind of guarded command language for modelling concurrent reactive systems. To our knowledge this was the first time that Action Systems have been used as test models in model-based testing. We exploited an object-oriented extension of Action Systems to support the object-orientation in UML and labelled the actions to interpret them as labelled transition systems. This semantic mapping from UML via Action Systems to labelled transition systems provided the link to the existing body-of-knowledge on formal testing that is mostly based on labelled transition systems. This translation is presented in \cite{15}. Unlike the existing UML tools for test case generation, we support parallel regions, nested states, non-deterministic models, inheritance and time-out events.

Consequently, we were able to develop a formal input-output conformance relation and testing technique for non-trivial UML State Transition Diagrams. An overview of this mutation-testing technique for UML models is given in \cite{16}. The tool chain comprises (1) a translator from UML to object-oriented Action Systems, (2) Argos, a translator from object-oriented Action Systems to simple Action Systems, and (3) Ulysses, a newly developed conformance checker for Action System models. In this work, we also compared model-based mutation testing against test case generation with the TGV tool. With our tools we found 97% of all known bugs in a car alarm system implementation. In contrast, the TGV technique detected 66% only.
**Test Case Generation with Ulysses.** Our test case generator Ulysses is an
ioco checker for Action Systems. It takes two Action Systems, an original and a
mutated one, and generates a test case that kills the mutant. To our knowledge,
this is the first mutation test case generator for Action Systems.

Ulysses expects the actions being labelled as input, output and internal
actions. For deterministic models, the generated test case is a sequence of
input and output events leading to the faulty behaviour in the mutant. For
non-deterministic models a tree-like adaptive test case is generated. Ulysses
explores both Action Systems, determinises them, and produces a synchronous
product modulo the ioco conformance relation. It is implemented in Sicstus
Prolog exploiting the backtracking facilities during the model explorations.

Different strategies for selecting the test cases from this product are sup-
ported: linear test cases to each fault, adaptive test cases to each fault, adaptive
test cases to one fault. Ulysses also checks if a given or previously generated
test case is able to kill a mutant. Only if none of the test cases in a directory
can kill a new mutant, a new test case is generated. Furthermore, Ulysses is
able to generate test cases randomly. Our experiments showed that for complex
models it is beneficial to generate first a number of long random tests for killing
the most trivial mutants. Only, when the randomly generated tests cannot kill
a mutant, the computationally more expensive product calculation is started.
The different strategies for generating test cases are reported in [7v]. In this
work, we also report the empirical results of testing a car alarm system with
the different strategies of Ulysses. The best strategy for this model was the
combination of random and mutation testing. One randomly selected test case
and 10 mutation tests were able to find all of the 38 known faults in a car-alarm
system implementation. Both, the pure random as well as the pure mutation
strategies missed one fault.

Ulysses is also able to generate tests for hybrid systems. This is discussed
in the next section.

### 1.6.5 Hybrid Systems

Hybrid systems involve discrete and continuous state updates as typically found
in controllers interacting with a physical environment. Many embedded systems
interact with a continuous environment and hence there is a strong interest in
applying model-based testing to such systems. The main contribution of this line
of research was an integration of model-based testing and qualitative reasoning.

**Qualitative Reasoning (QR)** is a technique from Artificial Intelligence to
abstract and simulate continuous systems qualitatively [Ku94]. The qualita-
tive models of QR represent basically cause-effect relationships and constraints
on model variables. They can be seen as an abstraction of the usually imple-
mented quantitative differential equation models when using Simulink or other
modelling languages. For example, in our first research on hybrid systems, we
combined VDM and differential equations in Mathematica [37].

In this new line of research the control of inter-connected water-tank systems
served as case study examples. The aim was to test the discrete controller in
the context of its continuous environment of water-flows. For this, adequate
test cases had to be generated automatically.
Conformance Testing of Continuous Behaviour. In [18] we show how Simulink models of control programs can be tested with respect to qualitative models. We modelled in the QR-tool Garp3 [BBJ16] which produces a state-graph of all the possible abstract behaviour for a given initial state. We defined an input-output conformance relation for these qualitative transition systems and implemented a corresponding test case generator. This tool was not mutation-based, but used test purposes to select meaningful test cases. However, we mutated a given Simulink implementation of a water-tank system and showed that we can detect the faulty behaviour with our generated test cases. The challenge here lies in the highly non-deterministic transition systems stemming from the abstract qualitative models. Consequently, the generated test cases are adaptive with respect to possible outputs, i.e. they may contain branching and looping output behaviour. The paper presents an algorithm for executing such adaptive test cases, with time being abstracted away, on a continuous SUT.

Combining Action Systems and Qualitative Reasoning. Garp3 is designed for qualitative reasoning and not for modelling hybrid systems. Hence, in the next phase of our research we combined Action Systems and QR models. The idea is to model the discrete controller as an Action System and its continuous environment as a QR model. The result was a new type of Action System, called Qualitative Action System, that was presented in [19].

This work was inspired by Hybrid Action Systems which add a continuous action to a classical Action System. This continuous action is a guarded differential equation that describes the evolution of the environment variables while the guard is satisfied. Instead of the differential equation, we added a QR model defining the qualitative evolution of the environment variables. The semantics of this new qualitative actions was formally worked out including a refinement law for qualitative actions. Again, a water-tank example served as demonstrating case study for this new modelling extension.

Mutation Testing of Qualitative Action Systems. Our model-based mutation testing approach for hybrid systems was presented in [20]. In this work we apply our mutation approach to qualitative action systems. For this the action labels have been extended with (controllable or observable) qualitative labels. Like in the purely discrete case, we interpret the labelled Qualitative Action System as a labelled transition system and check for input-output conformance. Here, the qualitative labels indicate that the environment is doing an update according to its qualitative action. Hence, we enforce an interleaving semantics between the actions of the controller and the environment assuming that the controller actions take zero-time compared to the environmental changes. This event view abstracts away from the internals of these environmental changes. We are only interested in the end-states, when the controller takes action again. Controllable qualitative actions carry a parameter with the initial state from which the qualitative simulation should start. Observable qualitative action’s parameters contain the end-states.

Consequently, the conformance check between the original and mutated qualitative Action Systems compares not only the discrete observables, but also the qualitative end-states of the environmental changes. Hence, the resulting test
cases contain expected values of the environment, e.g., that after certain actions of the controller the water tank should be empty.

For automated test case generation, the Ulysses tool has been extended with an existing simulation engine for qualitative models. The conformance checking algorithms are described in [21]. The paper also shows the empirical results of checking the conformance between a water tank model and 36 of its mutants: Ulysses found 8 mutants being equivalent and 22 being non-equivalent. Each conformance check took in average about 1.3 seconds. Once non-conformance has been detected, the test case extraction process is the same as for discrete action systems. Consequently, with Ulysses we are able to generate test cases for hybrid systems. To our knowledge, this is the first mutation-based test case generator for hybrid systems.

1.7 Concluding Remarks

1.7.1 Summary

In this thesis we have presented our research on mutation testing on the modelling level. Our aim to generate test cases that cover specific faults has led to a series of research results that range from theory via implementation to application.

The theoretical investigations are based on a formal semantics of models, conformance and test cases. In this setting we showed that the usual search for test cases that prove non-equivalence between a program and a mutant can be generalised to a non-conformance problem. By fixing the conformance relation and deriving the conditions for non-conformance, we obtained the constraints for our mutation test cases. The advantage of this theory is that it also works for non-deterministic models. We showed this in the Refinement Calculus and in the Unifying Theories of Programming. In addition, we formally proved that our so generated test cases correctly kill mutants. We mainly used refinement as a conformance relation in our theoretical work, but also presented a reformulation of Tretmans’ input-output conformance in the predicative semantics style of the Unifying Theories of Programming. We have also contributed to a fundamental dispute on false-negative test cases derived from axiomatic specifications. We could link the original arguments from philosophy of science to the problem of incompleteness of conformance relations.

The first application domain for our mutation testing technique were contracts. Contracts are program annotations in the form of pre-postcondition specifications. We developed a test case generator for OCL contracts using a constraint solver. The input to the solver were the non-refinement constraints developed in the theory. Later, we showed that for C# contracts the same can be achieved with its existing verification tools.

The next application domain were communication protocols. We first showed how existing tools can be used to support model-based mutation testing. Testing the HTTP-protocol implementation of Apache revealed surprising behaviour. Next, we showed several ways how to improve the test case generation process: we sliced the models by exploiting the knowledge of the fault location and used an input-output conformance checker. The SIP and Conference protocol served as case studies. We found faults in implementations of the SIP, but more system-
atic experiments showed that not all known faults in a system under test could be found with one testing technique. We also investigated symbolic test case generation techniques and found that a BDD-based tool failed for the SIP models. Therefore, we implemented a symbolic execution technique which worked. However, for the conference protocol the BDD-based approach excelled our tool. This confirmed again that the performance of symbolic techniques is highly dependent on the modelling style. An alternative style of modelling protocols was investigated with the REO coordination language. A new predicative semantics over timed traces was developed in Unifying Theories of Programming. A refinement checker showed that our original mutation testing theory that is based on refinement also works for protocols. The interesting point of this work is that new fault models can be expressed by mutating the data-flow oriented REO networks.

Embedded systems were our most recent application domain. Here, we had to deal with new challenges: the informal UML models had to be translated to formal models. We chose Action Systems as a suitable modelling framework. By interpreting them as labelled transition systems we could build on the existing testing theory. A new input-output conformance checker for Action Systems was implemented in Sicstus Prolog. The tool is able to generate adaptive test cases for non-deterministic models. Different test selection algorithms were studied. Experiments showed that a combination of the cheap-but-inaccurate random testing and the expensive-but-accurate mutation testing technique performed best.

The new test case generator is also able to handle hybrid system models. We abstracted from the continuous behaviour of a controller’s environment using qualitative reasoning techniques. First, action systems were extended with qualitative actions describing the qualitative behaviour of the environment. Then, we integrated an existing qualitative reasoning engine into our test case generator. The result is a test case generator that produces adaptive test cases including controllables and observables describing the qualitative changes of the environment. This was the first mutation-based test case generator for hybrid systems.

1.7.2 Discussion

In the following we discuss our work by answering frequently asked questions related to our work. We start with the very first question in this thesis:

Is model-based mutation testing able to show the absence of bugs?

Testing cannot show the absence of arbitrary bugs, but it can show that a specific bug has not been implemented. By a specific bug, we mean a fault that has been modelled in a behavioural model of the system under test (SUT). We do this fault modelling by mutating the models. We then generate a test case that covers this fault. Executed on the implementation, it will detect if this fault has been implemented.

To be more precise, this is only true for a deterministic SUT. For non-deterministic systems, we need an additional fairness assumption over the non-deterministic choices of the system. This means that after a certain number of repetitions of a test case, all of the involved non-deterministic branches are
taken. This assumption is common to all black-box testing approaches. Given access to the source code, we could measure this branch coverage.

Hence, if we know what kind of faults we are testing for, it is possible to derive tests that can prove their absence. This anticipated faults serve the role of negative requirements and the purpose of the mutation tests is to cover them. This is strongly related to the fault-tree analysis known from safety-analysis [LGTL8y].

The critical assumption in this methodology are the fault models. Faults that were not anticipated will not necessarily be detected. This means for our mutation approach that only faults expressible as simple mutations are detectable. Consequently, the modelling style plays a major role. The more abstract the models, the coarser the fault models will be. Hence, we conclude that model-based mutation testing can only guarantee the absence of bugs that are captured within the (mutated) models. However, if abstraction is a problem, one could question the usefulness of models:

**Why models?** The models provide the necessary redundancy in any verification effort. In formal correctness proofs this redundancy is provided by a specification, in model checking by a temporal property and in classical testing by a hand-written test case. In model-based testing the test model provides the redundancy and serves as a kind of script for generating the test cases. Furthermore, models can capture non-functional properties. For example, non-functional properties like time, energy consumption or security can be captured in a model.

**Is modelling not too costly?** Modelling is costly, but certainly cheaper than writing hundreds of high-quality tests. In the car-alarm-system case study the longest test cases had 14 interactions with the system. This is very hard to get right manually without a model. Furthermore, if the requirements change it is easier to change a model and generate fresh test cases from it, than adapting a large test suite.

**Does model-based mutation testing scale?** The barrier of this technique is the equivalent mutant problem. With an equivalent mutant we have to perform a full equivalence check up to a certain depth. Since the problem is NP-complete our exploration depth is limited. Here partial models help. Several smaller test models focusing on different aspects of a system can be developed. Furthermore, different symbolic analysis techniques may be applied to overcome scalability issues. This is similar to model checking research coping with state space explosion. Finally, we have combined classical test case generation techniques with our mutation approach. The easily detectable faults will be caught be the classical techniques. Only for the subtle faults the more expensive test case generation technique is applied.

### 1.7.3 Future Work

The presented work shows that model-based mutation testing involves a variety of research directions and is far from being a closed case. As of today, no
commercial tool has adopted this technique yet. Scalability is certainly an issue, but we firmly believe that advances are possible.

Symbolic Techniques. We currently investigate different symbolic analysis techniques to address state space explosion: constraint solving and SMT solving are promising candidates. However, the first experiments show that we cannot simply translate the non-conformance problem to one big formula and let the solvers do the job. A clever combination of normal form transformation, directed search and solving is necessary.

Parallelisation. Parallelisation and distribution is another line of research to tackle complex models. High-level parallelisation is straightforward, since each mutated model can be analysed in parallel. Low-level parallelisation targets the search for non-conformance itself. For example, the branching of symbolic execution leads to a natural form of parallel search. Another strategy would be to apply different search strategies and techniques in parallel.

Non-Functional Fault Models. Models may express non-functional properties, like energy or other resources. In [19] we have modelled biomedical sensor networks and simulated the impact of limiting their resource constraints, like memory consumption. With our mutation technique we could generate tests that show the difference between an unconstrained system and a system with limited resources. The result would be test cases that stress the system at its resource boundaries. We can also think of more elaborate fault models that express faulty assumptions with regard to non-functional requirements, like e.g., power consumption.

Semantic Mutation Testing. Another interesting application of mutation testing is semantic mutation testing. Instead of syntactically altering the mutation, we can also alter the semantic mapping of a given syntax. The result is a correct and a mutated semantic model exhibiting different behaviour. We could generate test cases that enforce one semantic interpretation. This would prevent modellers from wrongly interpreting a modelling language. Hierons et al. [CDH10] present a nice example of a cruise control modelled as a state machine with nested states. When braking while the cruise control increases speed, the semantics of the STATEMATE modelling tool correctly switches off the cruise control. However, in UML’s semantics the car would erroneously accelerate. A model-based mutation test could uncover such misunderstandings due to wrong semantics.

Here, we end our brief outlook on future research and applications. We hope that we could demonstrate that the topic is intellectually inspiring. In this research we constantly wandered between the worlds of theoretical and applied computer science and still await new discoveries. The future will show its impact. At least we hope that we gave some insight into the theory of testing in general and in our generalisation of mutation testing in particular. Since it is certainly true that in computer science

“We know less about the theory of testing, which we do often, than about the theory of program proving, which we do seldom.”

[GG75]
Bibliography

Publications Forming the Habilitation Thesis


Other Related Publications of the Author


[43] Immo Grabe, Mohammad Mahdi Jaghoori, Joachim Klein, Sascha Klüppelholz, Andries Stam, Christel Baier, Tobias Blechmann, Bernhard K. Aichernig, Frank S. de Boer, and Andreas Griesmayer. The credo methodology


Related Literature


extra-extra-read-all-about-it-nearly.html (last visit January 5, 2012).


42